ICT, Strategy & Policy
www.tno.nl
+31 88 866 50 00â€‹â€‹
info@tno.nl

TNO 2024 R11725 – 3 May 2024

Investigations on Probabilistic Programming Applications in Engineering

Author(s)	Alvaro Piedrafita Postigo Gert-Jan van den Braak Leonardo Barbini
Classification report	TNO Public
Title	TNO Public
Report text	TNO Public
Number of pages	45
Number of appendices	0
Project name	Probabilistic Programming Applications
Project number	060.60544/01.01
Contacts	alvaro.piedrafitapostigo@tno.nl gert-jan.vandenbraak@tno.nl leonardo.barbini@tno.nl

Contents

1 Introduction

Engineering companies employ a multitude of deterministic and stochastic simulation models, instantiated as computer code, to compute outputs from specified inputs. These models are, for instance, utilized to predict system performance under a range of parameter configurations.

A prevalent engineering challenge, however, is the inverse problem of determining the required inputs that yield desired outputs. This involves for instance, calibrating system parameters to achieve predefined performance targets. Due to the intrinsic complexity and randomness within these engineering simulation models, their direct mathematical or computational inversion is usually impractical, and often infeasible.

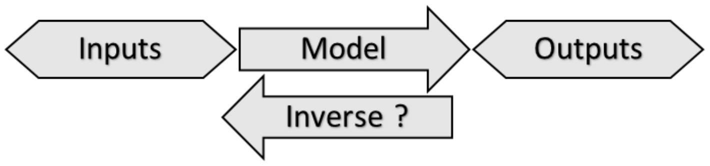

Figure 1-1 Inverting complex engineering models is needed but cumbersome

An alternative approach to perform such computations without model inversion is statistical inference. If the simulation model is specified as the joint probability distribution

$$P(Outputs|Inputs)P(Inputs)$$

then the quantity of interest $P(Inputs|Outputs)$ can be computed using Bayes' theorem as

$$\frac{P(Outputs|Inputs)P(Inputs)}{p(Outputs)}$$

This approach is in principle generic but not always numerically tractable, and cumbersome to implement from scratch. Probabilistic programming (PP) is a paradigm designed exactly to facilitate the specification and evaluation of such statistical inference problems. Thus making statistical inference generically applicable.

Furthermore, PP exhibits robustness against measurement uncertainty, data scarcity, and data incompleteness, which are common situations encountered in industrial engineering. These aspects, combined with recent advancements in PP libraries, result in a significant potential for the application of PP to the types of inverse engineering challenges described above.

This report explores the topics introduced above, offering an initial assessment study of the feasibility of PP for engineering applications and providing an understanding of its core principles. It also gives a literature review and shows several coding examples of PP.

1.1 Organization of the document

This document is organized as follows:

- Section 2 gives an overview of the literature on PP. It covers generic references to the mathematical and implementation foundations of PP, as well as references to available SW libraries for PP. It also contains available references to applications of PP in engineering inference tasks.
- Section 3 contains examples of PP applied to several inference tasks. In this section we implement small toy examples to make the reader familiar with the concepts of PP. We use different open-source SW libraries for PP. The code used in the examples is shared as Appendix to this document.
- Section 4 applies PP to diagnostics cases from current TNO-ESI projects. The first example focusses on centring a belt around two cylinders, the second example focusses on print quality in a professional printer. Relevant code snippets from these examples can also be found in the Appendix.
- Section 5 concludes this document, summarising the findings and paving the road towards further research.

How to read this document

This document summarizes the work done in the Kennisinvesteringsproject [1] (KIP) Feasibility study of probabilistic programming applications in industrial engineering. It has been written with the twin goals of serving as an accurate repository of activities and an informative report on probabilistic programming for those unfamiliar with the technology.

For the reader interested in a high-level or strategic overview of probabilistic programming, sections 2 and 5 contain all necessary information and conclusions. A superficial reading of section 3 will suffice, and the appendix can be omitted.

For the reader interested in the technical details, sections 3 and 4 contains detailed explanations of the examples studied and the rationale behind their probabilistic programming approach. The code provided in Appendix A should be sufficient to replicate the results presented. The reader is encouraged to contact the authors via email for any questions or comments regarding implementation.

[1] Knowledge investment project

2 State of the art

In order to capture the state of the art in probabilistic programming (PP) a literature review was conducted for which the results are summarized in Section 2.1. Also experts in the field of PP from the Eindhoven University of Technology and the University of Amsterdam were interviewed, as discussed in Section 2.2.

2.1 Literature review

The literature review into PP is split in three parts: Section 2.1.1 describes probabilistic programming and discusses different approaches to PP, Section 2.1.2 discusses various programming languages and software libraries for PP, and finally Section 2.1.3 highlights some applications in which PP is used.

2.1.1 Probabilistic programming: a very short introduction

Probabilistic programming is a programming paradigm specifically designed to facilitate statistical inference: *'[Probabilistic programming] is fundamentally about developing languages that allow the denotation of inference problems and evaluators that "solve" those inference problems'* from [1]. Probabilistic programming also enables the modelling and reasoning over complex relationships among variables and accomplishes tasks involving statistical analysis and the handling of uncertainty across diverse domains.

Several methods have been introduced to perform statistical inference, sampling-based methods like Markov Chain Monte Carlo [1], gradient-based methods like automatic differentiation variational inference [2] and analytic methods like message passing [3], or combinations thereof [3].

Probabilistic programming has been successfully applied to various areas of science and engineering such as particle physics [4], geological modelling [5], captcha solving [6] and constrained simulation [7]. There are libraries in different programming languages supporting probabilistic programming such as Stan in C++ [8], Pyro [9] in Python and RxInfer [10] in Julia.

The general picture

In industrial settings, processes are often tightly controlled, and their outputs generally well understood. A perfect industrial process is thus akin to a mathematical function that reliably transforms inputs, denoted as $\vec{x}$, into outputs, $\vec{y}$, for which we can explicit a (data) generation procedure, f, often in the form of computer code, such that $\vec{y} = f(\vec{x})$.

In reality, some of the variables affecting a process will be unobserved. These variables are denoted as $\vec{h}$, and often called *hidden* or *latent* variables. In such settings, the quantity and nature of these hidden variables, and the different ways in which $\vec{h}$ affects $\vec{y}$, are generally well understood. What remains unknown, then, is the *value* of the latent variables.

And so, it is possible to write an explicit procedure, which can be deterministic or probabilistic, known as a *generative model*, that can generate the output $\vec{y}$ as a function of $\vec{x}$ and $\vec{h}$.

Mathematically, we describe these as hidden random variables, and the outputs as observed random variables. This means that generative models are probabilistic processes:

$$\vec{y} \sim g\left(\vec{x}, \vec{h}\right),$$

Where the symbol ~ denotes that the probability distribution of the left-hand side is a function of the right-hand side.
Inference in the probabilistic programming setting does not assume a single set of hidden variables $\vec{h}$ that work for all $(\vec{x}, \vec{y})$ pairs, but rather it starts with a known model $g\left(\vec{x}, \vec{h}\right)$, and infers the probability distribution of $\vec{h}$ for a given set of observed inputs $\vec{x}$ and outputs $\vec{y}$. In concrete terms, a probabilistic program takes a generative model and some data and performs an operation Γ such that:

$$\begin{cases} \vec{x}, \vec{y} \\ \vec{y} \sim g(\vec{x}, \vec{h}) \end{cases} \rightarrow \ \vec{h}(\vec{x}, \vec{y}) \sim \Gamma(\vec{\mathrm{x}}, \vec{y})$$

This operation of computing an estimate for the hidden variables, be it a probability distribution or a single point, given a model and a pair of observed inputs $\vec{x}$ and outputs $\vec{y}$ is called *performing inference.*

By contrast, in most data-driven learning tasks a parametrized model $f_\theta : y = f_\theta(x)$ is proposed and many pairs of $(\vec{x}, \vec{y})$ are used to *infer* the best parameters θ for the model. This task is usually known as *learning* the function f.

Figure 2-1 illustrates the difference between probabilistic programming and standard data-driven machine learning (ML) approaches. The main difference is that in standard ML, the unknown quantities (model parameters θ) are assumed to be constant for all $(\vec{x}, \vec{y})$ pairs and learned from many instances of data, while in probabilistic programming, the unknown quantities (hidden variables $\vec{h}$) are allowed to depend on the given data and inferred on a case-by-case basis.

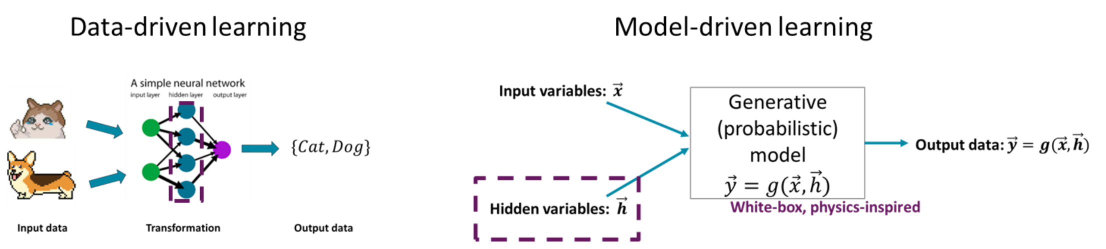
Data-driven learning
Model-driven learning
A simple neural network
input layer
hidden layer
output layer
$\{Cat, Dog\}$
Input data
Transformation
Output data
Input variables: $\vec{x}$
Generative (probabilistic) model
$\vec{y} = g(\vec{x}, \vec{h})$
Output data: $\vec{y} = g(\vec{x}, \vec{h})$
Hidden variables: $\vec{h}$
White-box, physics-inspired

Figure 2-1: Data-driven learning (left) versus model-driven learning. The purple box indicates the parts that are inferred (θ or $\vec{h}$). In the data-driven example, learning the parameters teaches little about the difference between cats and dogs. In model-driven learning, the learned parameters relate to a physical model, and thus inform us about the real world.

Architecture of a probabilistic program

The way a probabilistic program manages to define the inference operator Γ is to:

1. Use the generative model to *encode* a joint probability distribution. $P(\vec{y},\vec{h}|\vec{x}) = \underbrace{P(\vec{y}|\vec{h},\vec{x})}_{likelihood} \cdot \underbrace{P(\vec{h})}_{prior}$, where $P(\cdot)$ denotes the probability distribution, and $(y|x)$ denotes the conditioning of the probability distribution of y on a particular value of x.
2. Condition the joint probability distribution above on both the observed values of $\vec{x}$ and $\vec{y}$.
3. Compute the posterior distributions of $\vec{h}$ given $\vec{x},\vec{y}$, defined as $P(\vec{h}|\vec{x},\vec{y})$, by somehow computing the likelihood and applying Bayes rule. This is the step that requires extensive computation.

The workflow of a probabilistic program follows this scheme, splitting the specification of the model, the conditioning of the model, and the inference, automating the latter two. This is depicted in Figure 2-2.

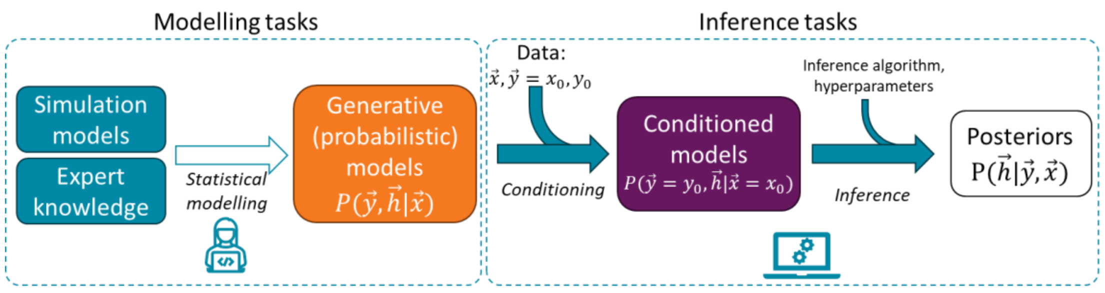
Modelling tasks
Inference tasks
Simulation models
Expert knowledge
Statistical modelling
Generative (probabilistic) models $P(\vec{y},\vec{h}|\vec{x})$
Data: $\vec{x},\vec{y} = x_0, y_0$
Conditioning
Conditioned models $P(\vec{y} = y_0,\vec{h}|\vec{x} = x_0)$
Inference algorithm, hyperparameters
Inference
Posteriors $P(\vec{h}|\vec{y},\vec{x})$

Figure 2-2: Workflow of probabilistic programming. The model creation is a manual process requiring expertise in statistical modelling. Inference tasks are mostly automated. Conditioning is usually automated and inference is a matter of choosing among a set of existing inference algorithms and hyperparameters.

Approaches to inference

The only step that requires extensive computation is the last one, since the computation of posteriors requires the computation of the likelihood function, which is often computationally intractable. An implementation of the architecture above is called a Probabilistic Programming Language (PPL). These are often distinguished on the basis of their approach to inference. There are roughly three ways to approach the problem of inference.

1. Use sampling techniques to approximate the likelihood function.
2. Use stochastic variational methods to approximate incomputable true posteriors with proposed computationally tractable surrogates.
3. Use analytic methods to simplify the intractable posterior.

These methods have pros and cons, which can be seen in Table 1.

Method	Pros	Cons
Sampling	Universal: Can infer posteriors of any shape. Precise: Can approximate posteriors to arbitrary precision. Mature: These methods have the longest history and are well understood.	Challenging to scale to high dimensions. Computationally expensive. No guarantee of convergence.
Stochastic Variational Inference	Scalable: Can handle high-dimensional problems. AI compatible: Variational methods can be merged with deep learning techniques.	No guarantee of convergence. Bounded precision: Can struggle to approximate difficult posteriors.
Analytic methods, message passing	Scalable and efficient.	Limited to the subset of analytic models (exponential families, discrete models).

Table 1: Pros and Cons of different approaches to inference.

2.1.2 Libraries, tools & software

Since the introduction of the first probabilistic programming software over 30 years ago [11], the space of probabilistic programming has seen the introduction of over 50 different libraries, languages and tools addressing different aspects of probabilistic programming and statistical modelling. A non-exhaustive list can be found on Wikipedia[2]

The introduction of the first universal probabilistic programming language in 2012, STAN [12], marked a turning point in the field, and development has accelerated since then. In Figure 2-3 we can see the growth in citations to a selection of the most modern PPL libraries. All the selected libraries are written for the Python programming language except Turing.jl [13] and Gen.jl [14], which are for Julia, and Stan, which is written in C++. One of the reviewed libraries, RxInfer [10], is not included in the figure since its introduction is too recent (2023) for a discussion of growth or trends.

[2] https://en.wikipedia.org/wiki/Probabilistic_programming

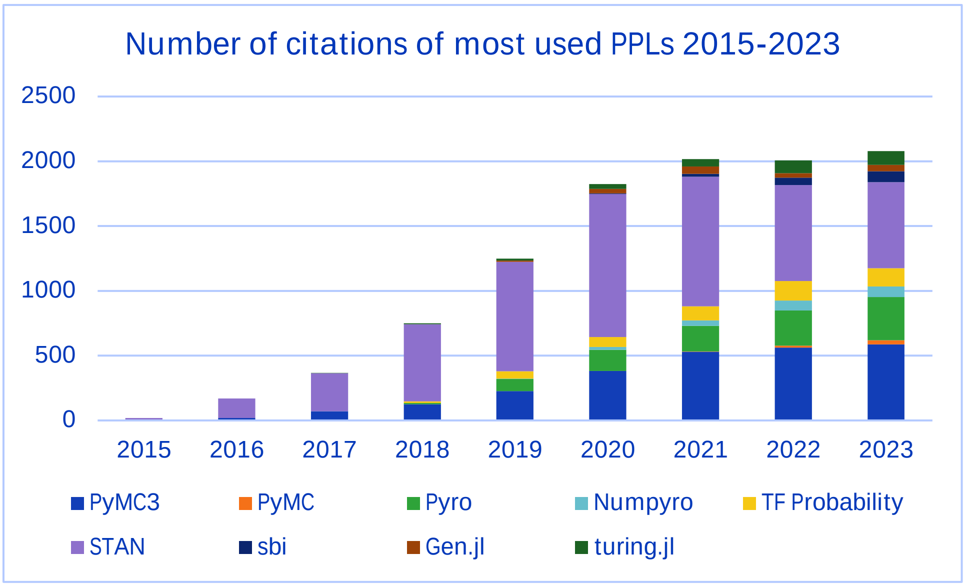

Figure 2-3 : Number of citations of the most commonly used PPLs in the years 2015-2023. Of special interest is the accelerating increase trend between 2015 and 2020. In the last three years, the number of citations has stabilized around 2000 per year.

All of the libraries in the figure support sampling methods as well as stochastic variational inference (SVI) in their inference engines, with the exceptions of Turing.jl, which is exclusively sampling-based, and SBI [15], which has a deep learning approach to inference.

In sections 3 and 4, we discuss example applications of probabilistic programming implemented in a subset of the libraries discussed in Table 2. Our selection is motivated by a balance between ease-of-use for the authors, performance, and breadth of approaches.

Library	Language	Inference engine	Year	Other features
NumPyro	Python	Sampling, supports SVI	2019	Fastest inference. Together with Pyro, preferred libraries of the ML/AI community.
PyMC	Python	Sampling, supports SVI	2016	Latest iteration of PyMC3. Most used and documented Python library. Some use in engineering areas. Lags in performance.
Turing.jl	Julia	Sampling	2019	Integration with Julia's automatic differentiation packages means it can readily be used for solving probabilistic differential equation models.
RxInfer	Julia	Message passing in factor graphs	2023	Still in early stage. Developed at TU/e. Established working relationship with the developers. Very scalable under certain conditions.

SBI	Python	Deep learning on surrogate neural networks	2020	Radically different approach. Almost entirely automated inference step means much lower statistics expertise required.
Gen.jl	Julia	Sampling, SVI	2019	Very flexible inference algorithms. Highest degree of customization.
Stan	C++	Sampling, SVI	2015	Preferred use in bioscience applications. Slow inference.
TF Probability	Python	Sampling, SVI	2019	Similar to NumPyro and PyMC, but built around TensorFlow.

Table 2: Overview of probabilistic programming libraries considered in this KIP

2.1.3 Applications

Probabilistic programming is used extensively in various fields of research. Especially in astrophysics and sub-atomic particle physics research, probabilistic programming is used to map observations to a model. Also in (computational) biology PP is used often. In engineering probabilistic programming is used less. When applied it is often used for reliability analysis, model fitting or defect detection. A couple of examples are mentioned in this section.

A common use of probabilistic programming is mapping results from a small experiment to a distribution in order to get the whole distribution characteristics. Lamont et al. employ a Bayesian reliability analysis to estimate reliability and expected life-time of encapsulated implanted electronics based on an accelerated aging experiment [16]. 36 samples were tested at three temperature levels for over 300 days after which only 4 samples had failed. Meng et al. analyse failures in a ship electromechanical system using a Bayesian method [17]. Based on the often incomplete error reports on ships a model is made which can be used for suggesting preventive maintenance.

Probabilistic programming can also be used to estimate parameters for a model. Tada uses Bayesian estimation to create an equivalent circuit of a solar cell [18]. The benefit over classical methods like nonlinear least-squares methods is that also estimation errors are computed, although the probabilistic programming approach requires a long computation time. Sun et al. use Bayesian computing to estimate the aircraft drag polar (the relationship between the drag on an aircraft and other variables, such as lift, angle-of-attack or speed) based on flight data for 20 common aircraft types [19]. Schön et al. model a nonlinear spring-damper system using differential equations and learn the damper’s and spring’s coefficients from observed (simulated) data [20]. This paper was used as inspiration for the Mass-spring-damper system example of section 3.3.

Steffelbauer et al. use probabilistic programming to model the acceleration of sea-level rise (SLR) in the North Sea [21]. Data acquired between 1919 and 2018 from seven tidal stations in the Netherlands and one in Germany are used to find the breakpoint where the sea-level rise increased from SLR_1 mm/year to SLR_2 mm/year. Various “short-term” effects are taken into account, such as seasonal changes (lunar cycle and yearly variation), atmospheric pressure and wind direction and wind stress. The model is implemented using PyMC3 [22] and consists of approx. 100 lines of Python code: https://github.com/steffelbauer/sea_level_rise_acceleration.

It is also possible to find defects using probabilistic programming. Tamaki et al. create a probabilistic model of cast iron parts [23]. By only sampling known correct parts broken parts can be identified when they fall outside the confidence interval. Wang et al. model the out-of-roundness (OOR) of metro wheels [24]. The probabilities resulting from the model are linked to the Sperling index, which can be used to evaluate vehicle comfort and consequently a maintenance threshold. Similarly, Boyali et al. use Simulation Based Inference (SBI) to identify vehicle (car) dynamics parameters [25].

2.2 Interviews

In the course of this study, we have reached out to academics in the Netherlands with expertise in the field of probabilistic programs, and interviewed research groups in the Netherlands who have developed or are actively developing probabilistic programming libraries. A summary of the interviews is given below.

On the strengths of probabilistic programming for industrial application, interviewees pointed the following:

- Probabilistic programs do best in settings where there is a clear model for data generation. In this regard, they seem well suited for industrial applications where processes can be modelled easily.
- Probabilistic programming is a favoured approach for parameter estimation problems with uncertain and/or missing data.
- “If one can write an ‘easy’ generative model, there exist in principle an inference model capable of computing it. Problems with inference are often caused by incorrect modelling.”

On the current state of practice in the field of probabilistic programming:

- Developers of probabilistic programs usually have data science/AI, biomedical, or applied mathematics backgrounds and haven’t paid attention to engineering problems. Most effort has been put to modern computer science problems such as image recognition, clustering, classification of noisy data, epidemiology, etc.
- Expertise requirements are still very high. Little effort has been made yet to lower them due to several factors:
    - *Novelty of the field*. Most progress in probabilistic programming is less than 10 years old.
    - *Inherent difficulty of inference problem.* Most practitioners find that complex models often require customized inference algorithms. No general inference algorithm exists.
    - *Unstructured nature of existing applications.* Universality and flexibility are the goal of most probabilistic programming languages. This usually comes at the cost of usability.
    - *High expertise among current users.* Most practitioners are academics or have deep expertise in statistics, inference and software engineering.
    - *Lack of incentives to tailor PPLs to specific applications*. Applications of probabilistic programming outside its original context are still scarce and there is no push from industry to make them more accessible.
- Lowering expertise requirements can potentially be done for sufficiently bounded domains. Additional research is needed in this regard. Not just a couple of research

papers but a comprehensive program requiring several researchers for a handful of years.

- There is a real slowdown of the field, due mainly to 3 things:
    - *Emergence of LLMs.* Modern LLMs have proven that black-box deep learning is more powerful than previously thought, and the limitations of that approach are still unknown, leading to many researchers to shift attention away from PPLs and towards LLMs.
    - *The community has done what it set out to do.* "Universal" PPLs exist now that allow specification of any model with continuous variables. Many inference algorithms are available and customization is possible if needed.
    - *Universal inference algorithms cannot exist*. It is impossible to make a universal, fully black-box PPL. Therefore, any attempt to automate inference must be to some extent problem-specific, which is contrary to the philosophy of PPL developers.

On the research directions that would facilitate adoption in an industrial setting:

- *Misalignment between simulation models and inference models.* Some things that can be easily expressed in a "forward" simulation model are hard to express in an probabilistic program in ways that make inference tractable. This is both a problem with PPL language specification as a problem with mathematical robustness.
- *Nesting of sub-models with different "theoretical" requirements for inference.* Models with many different kinds of variables and variable interactions may benefit from breaking the problem into sub-problems and using different inference subroutines. This requires:
    - Better understanding of strengths and limitations of different inference approaches.
    - Criteria to select the right tool for the (sub)-job.
    - A mathematical framework for fusing these subroutines into a single inference algorithm.
- *Hierarchies and coupling of processes at different scales.* The parameters determining the distribution of random variables can themselves depend on other random variables. This is known as variable hierarchy.
    - Selecting the right level of granularity to infer slow-moving variables from fast-moving data has a big impact on inference efficiency.
- *Hybrid models are particularly difficult.* Inference on models with many interacting continuous and discrete variables is harder than inference on fully continuous or fully discrete models. Additional research is needed to unify both types.

3 Example applications

The examples presented in this section showcase the capabilities of probabilistic programming in tackling inference tasks. We do not investigate the comparison of the PP approach to classical approaches to solve such tasks. This should be done in future research.

The following concepts and notations will be used in all the examples below:

Simulation model - is a computational model that given some inputs computes some outputs. In other words, the simulation model is the function which computes

$$function(inputs) \rightarrow \ outputs$$

This model is used to generate the data on which the inference task will be executed. This function can be deterministic or stochastic. In an engineering application, the simulation model is the modeler's best guess on how to mathematically and computationally model the phenomena of interest. In this way the data generated by such a model is assumed to be resembling *sufficiently* well the real world measurements, given the same values for the inputs. With data* we will refer to those outputs generated by the simulation model given the inputs*.

Inference model - similarly to the simulation model is a computational model that given input parameters computes outputs. The fundamental difference between the simulation and inference models is on the inputs. In the simulation model these are the actual values on which *function()* is executed. On the contrary, in the inference model those are parameters of probability distributions, e.g. mean and variance of a Normal distribution. These distributions are referred to as the priors. Some actual values are then sampled from this distributions and subsequently the *function()* is executed on those sampled values. In other words, the simulation model has two steps which read like:

$$prior\_distribution(inputs\_parameters) \rightarrow \ sampled\_inputs\_values$$
$$function(sampled\_inputs\_values) \rightarrow outputs$$

This means that the inference model is probabilistic, even if the function of the simulation model is deterministic. Every time it is executed it will sample different inputs values on which execute the function. Finally, additionally to the simulation model, the inference model also takes some data*. This data* is used in the inference routine, described next.

Inference routine - is the computational model that executes the statistical inference. That is to infer the inputs* from data*, without inverting the *function()*. The goal of probabilistic programming is to lower, as much as possible, the effort in defining mathematically and computationally such inference routine. With some degree of difference, the PP libraries described in Section 2.1.2 have inference routines which take as inputs the inference model, the data* and the prior distributions of the input parameters. While returning, the posteriors distributions on theses parameters, samples from such distributions and some metrics to describe the accuracy of such inference.

Below we will present some examples using these concepts and showcasing applications of PP. The code to reproduce the examples is shared in the Appendix Section of this document.

3.1 Biased coin

In this example we model the toss of a coin *n* times. We assume that it is already known that the coin is biased, but that we do not exactly know to what extent. Our goal is to infer the actual amount of bias from the *n* tosses. We implement this example in the Julia programming language using, the RxInfer [10] library for the inference.

The simulation model consists of sampling *n* times the discrete Bernoulli distribution with parameter *p,* i.e. a sample is 1 with probability *p* and 0 with probability *1-p*. This simulation model is stochastic since every time we execute this model, for the same *p* and *n*, the *n* values will be different.

The inference model is very similar to the simulation model, with additional information on the prior distribution of the parameter *p*. We assume that this prior is a Beta distribution, which is parametrised by the two parameters *p1* and *p2*. The values of these parameters are assumed known as part of the prior information. We also assume that the two parameters are independent. The inference model then proceeds in two steps:

1. take sample *p* from a Beta with parameters *p1* and *p2.*
2. take *n* samples from a Bernoulli with parameter *p* from step 1.

Notice that step 2 above is equivalent to the simulation model.

The inference routine will then use the inference model, together with a set of n samples from the simulation model and return the posterior distribution for the parameter *p* of the Bernoulli.

In the figures below we show some inference results. The parameters used in each figure are shown in the title, where *p** is the actual parameter of the Bernoulli used to generate the data in the simulation model; *p_est* is the mean of the posterior distribution; *p_naive* is the mean of the *n* samples. The posterior is the result of the inference. The results depend on the number of samples *n*, and by the 'distance' between the *p** and the prior of the Beta. This dependence is shown in the different figures below. Increasing the number of samples and minimising the 'distance' between the *p** and the prior of the Beta allows to perform a more accurate inference.

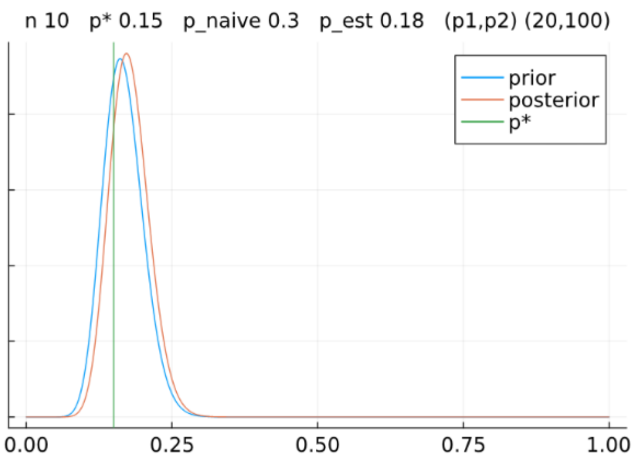

Figure 3-1 Biased coin. Inference based on 10 tosses. Correct prior. Correct inference.

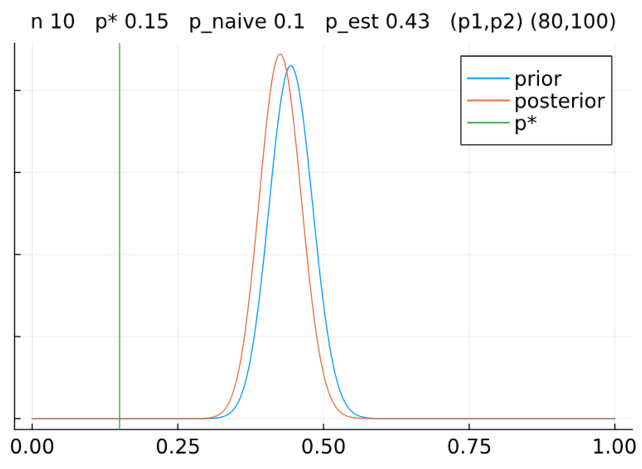

Figure 3-2 Biased coin. Inference based on 10 tosses. Wrong prior.

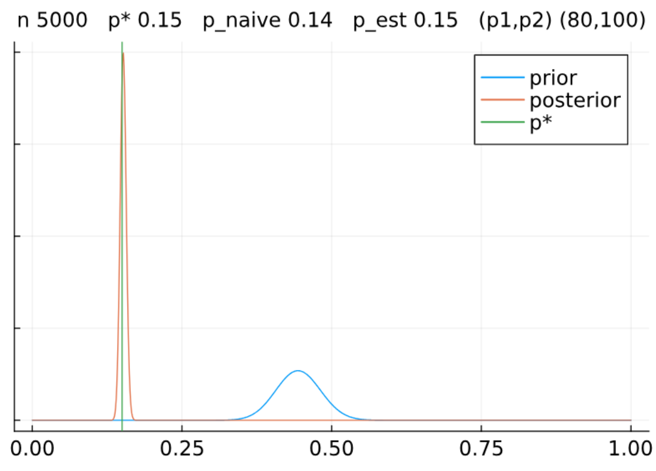

Figure 3-3 Biased coin. Inference based on 5000 tosses. Wrong prior.

Specifically, Figure 3-1 shows an ideal situation, where we have a 'correct' prior for the Beta. Here correct means that the sample p^* (green vertical line) has a high probability to be sampled from the prior (blue curve). The variance of the posterior distribution (red curve) is dependent on the number of samples n=10, increasing the number of samples will decrease the variance of the posteriors. Figure 3-2 shows a less ideal situation in which the prior on the Beta is off, i.e. the p^* is very unlikely to be sampled from this distribution. With a low number of samples n=10 we see that the PP approach is 'reluctant' on changing its prior beliefs on this coin. Increasing the number of samples to n=5000 as in Figure 3-3 will change this situation, with the posterior being estimated and peaked around p^*.

3.2 If-else example

In this example we model a data-generating process containing logical if-else statements. This results in discontinuities in the execution trace, meaning that two contiguous datapoints can be arbitrarily far apart, and input and output distribution of data that exhibits complex behaviour.

Our simulation model accepts five parameters, n, m_0, m_1, σ, and p. Building on the previous example, we begin by sampling n Bernoulli distributions with parameter p, each taking values either 0 or 1. However, rather than simply recording the output, we make a second sample from a different distribution, a gaussian distribution with mean m_0 (respectively m_1) if the value of the Bernoulli is 0 (respectively 1) and standard deviation σ for both gaussians. This kind of model, where the generated data is the result of a random mixture of two or more distinct processes, is known as a mixture model.

Figure 3-4 shows a depiction of a dataset generated following this process.

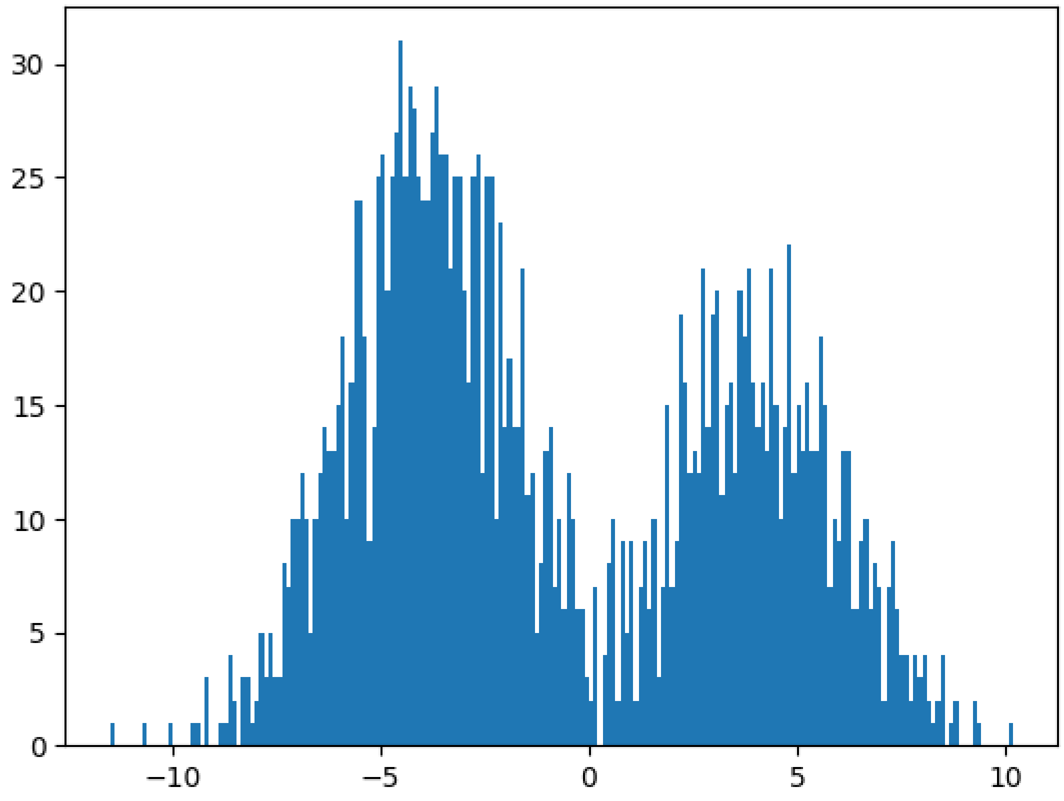

Figure 3-4: Dataset generated by a mixture process with $n = 2000$, $m_0 = -4$, $m_1 = 4$, $\sigma = 2$, and $p = 0.4$.

The task of the inference engine is to identify the two different gaussian modes and the proportion of samples that correspond to each mode. In other words, the task is to identify the parameters m_0, m_1, σ, and p.

The inference model proceeds similarly to the simulation model with some slight modifications. First, we give priors for all inferred parameters. We use rather uninformative priors for all, with $locs = (m_0, m_1)$ being the vector formed by m_0 and m_1, both drawn from a gaussian distribution with mean 0 and standard deviation 15. The prior for σ must only allow for positive numbers, so we choose to be a half-normal with width 5. The prior for p is uniform. Then we say that each step an $assignment$ is generated according to a Bernoulli distribution with parameter p and the data is drawn from a gaussian distribution with $std = \sigma$ and $mean = locs[assignment]$. This is the way we express in the inference model the condition that a sample is drawn from one or the other Gaussian according to the value of the Bernoulli sample.

Table 3 shows the computed posteriors for the parameters of the simulation. Inspecting the table it seems that the inference overestimates the values of m_0 and m_1 by about 0.1.

Interestingly, this is not because the inference engine is wrong but because the data is slightly skewed to the positive numbers.

This becomes clear when one computes the sample average of the data in Figure 3-4 and finds it to be 0.699, rather than the theoretical value of 0.8. This betrays an important property of simulation models with internal randomness, namely that the data they generate can be affected by random fluctuations. In reality, no random process is perfectly unbiased, and random fluctuations in the simulation data will be considered intrinsic by the inference engine because it only has access to the data.

Table 3 Parameter posteriors for mixture models.

Parameter	Est. Mean	Est. 5%	Est. 95%	Real value
m_0	-3.91	-4.01	-3.81	-4
m_1	4.11	3.98	4.23	4
σ	2.00	1.94	2.06	2
p	0.4	0.38	0.42	0.4

Another important property of most probabilistic programs is that they generate posterior distributions, not simply point estimates for mean and variance of variables. Moreover, they can generate the *joint* posterior distribution over *all* variables, which can be used to discover correlation (or independence) between variables.

3.3 Mass-spring-damper system

In this example we consider a 1-D mass spring damper system schematically represented in Figure 3-5 below and mathematically modelled by the following second order differential equation:

$$m\frac{d^2x(t)}{dt^2} + c\frac{dx(t)}{dt} + kx(t) = 0$$

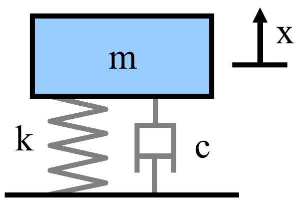

Figure 3-5 Schema of the mass spring damper system. From Wikipedia[3].

The inference task is to infer the posterior of the parameters *k* and *c* given their priors and *n* samples from the measured position at *dt* time intervals.

$y[i] = x[i \cdot dt] + \eta[i].$

Where $\eta[i]$ are measurement noise terms sampled independently from a Normal distribution. The mass *m* is assumed to be known and equal to 1. Here we implement this inference task in Julia using the probabilistic programming library Turing.jl [13]. The inference task is executed using a Markov chain Monte Carlo (MCMC) [13] sampling approach.

[3] https://en.wikipedia.org/wiki/Mass-spring-damper_model

Figure 3-6 shows the simulated data used in the inference task. Blue line is the continuous time series representing the position of the mass m. The red dots represent the noisy measurements of such position, every dt.

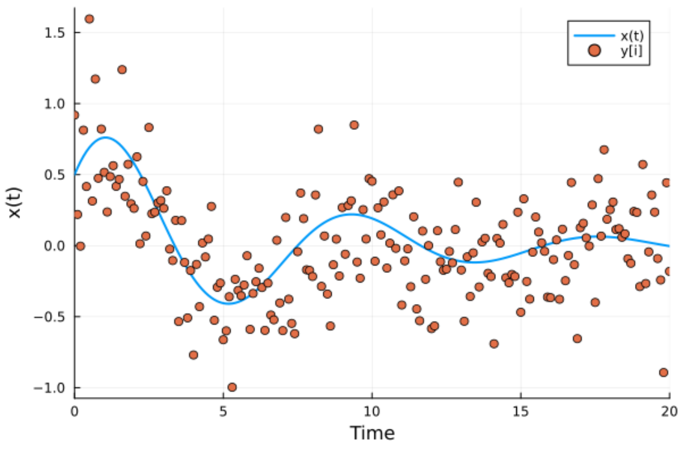

Figure 3-6 Mass spring damper system. Example simulated data for the position x(t) and its noisy measurement y[i]

Figure 3-7 and Figure 3-8 show the inference results for the k and c parameters respectively. Where k^* and c^* are shown as a vertical green line, the uniform prior distribution is shown as a horizontal blue line and the posterior as a red line. The uniform prior distributions for k and c are both in the range 0 and 2. These priors are the most uninformative as possible, to represent a situation when little knowledge is known on the system. Notice how the estimate of the parameter k is more accurate than the c parameter. This is because we only use measurements of the position of the mass. Using the velocity would results in opposite precision.

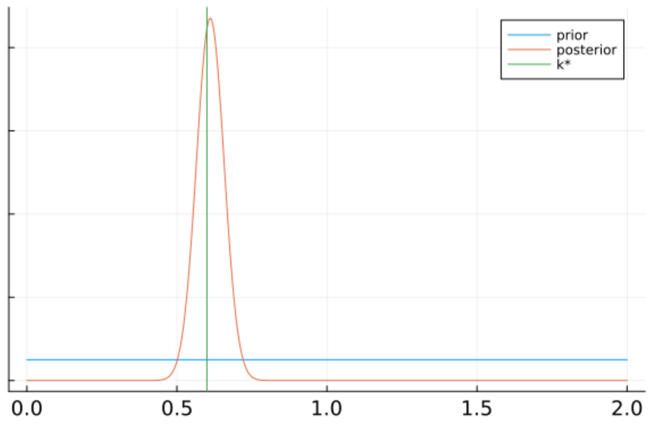

Figure 3-7 Mass spring damper system. Inference result for the k parameter. k^* = 0.60 k_est = 0.59. 201 Data samples. 100 MCMC samples.

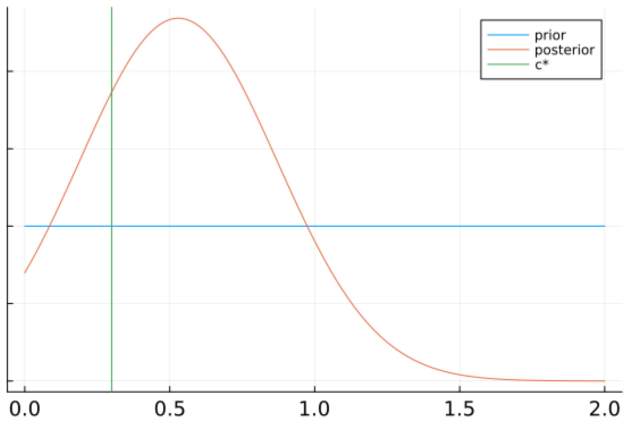

Figure 3-8 Mass spring damper system. Inference results for the *c* parameter. 201 Data samples. 100 MC samples.

Finally, Figure 3-9 show the results for the inference of the parameters *c* by increasing the number of samples used in the Markov chain in the MCMC inference algorithm from 100 to 1000 which results in a more accurate estimation of the parameter.

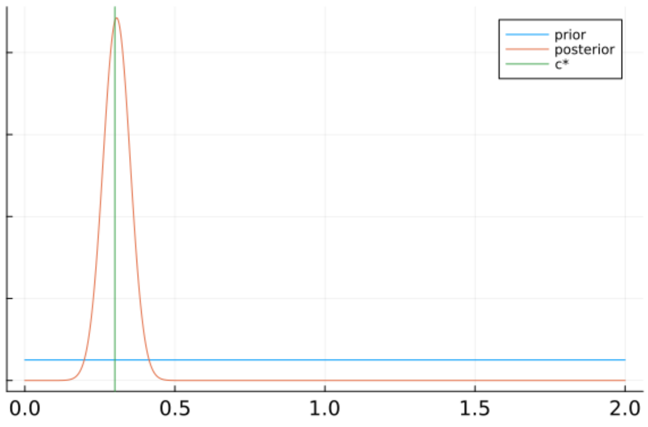

Figure 3-9 Mass spring damper system. Inference results for the *c* parameter. 201 Data samples. 1000 MC samples.

3.4 RC filter

An RC-filter is an electric circuit which consists of resistors and capacitors. The simplest RC-circuit is a first order RC-filter consisting of one resistor and one capacitor as shown in Figure 3-10. The output voltage V_c can be computed as a function of the input voltage V_{in} and the values of the resistor R and capacitor C using the following differential equation:

$$V_C = V_{in} - RC\frac{dV_C}{dt}$$

The goal of this example is to find the values for the resistor R and the capacitor C given a time series of input signal V_{in} and output signal V_c. In practice this can be done by applying a known signal as V_{in} and measuring V_c and computing the values of R and C. In the previous Mass-spring-damper system example a single input value was provided: the start position while in this example the input is a time series of V_{in} and V_c.

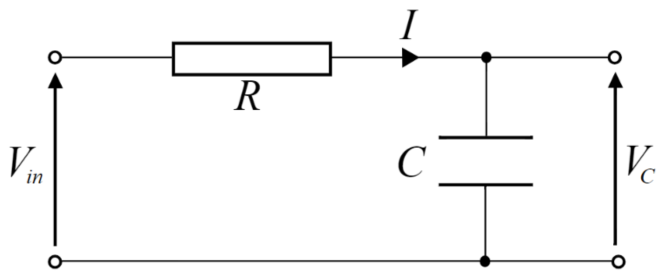

Figure 3-10: RC circuit configured as a low-pass filter. From Wikipedia[4].

The RC-circuit is modelled in Julia using DifferentialEquations.jl [26] as an Ordinary Differential Equations (ODE). Given a differential equation, an input signal and a start- and stop time a time series of results is computed. The differential equation for V_c can be written in Julia as follows:

```
dVc .= (Vin(t) .- Vc) / (R * C)
```

Note that dotted operators (.= and .-) are used to indicate these operations are broadcasted, meaning they operate on time-series which are called vectors in Julia.

The selected input signal V_{in} is a (single) block wave as shown Figure 3-11. The output signal is the typical shark-fin created by low-pass filtering a block signal, also shown in Figure 3-11. Random measurement noise is added to the filtered signal as indicated by the blue circles.

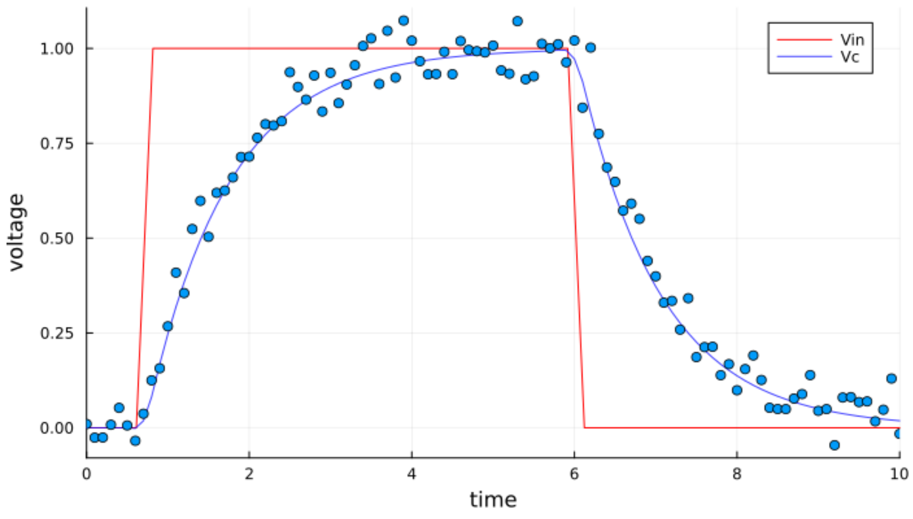

Figure 3-11 input and output signal for RC filter

The data points with measurements noise are used in the inference step where the values of the resistor (*R*) and capacitor (*C*) are estimated. This is done using Turing.jl [13], a Julia library for general-purpose probabilistic programming. Their website has an example dedicated to parameter estimation using Bayesian inference of differential equations: https://turinglang.org/dev/tutorials/10-bayesian-differential-equations/

During the inference potential values for R and C are sampled from a Normal distribution. The tolerances (deviations from the nominal value, calculated as σ/μ, where σ is the standard deviation and μ is the mean) of the resistor and capacitor are known to be 1% and

[4] https://en.wikipedia.org/wiki/RC_circuit

10% respectively. However, specifying the variance in the Normal distribution to match the tolerance gives a narrow sample range. This causes the inference to be very slow. A better approach is to allow the Normal distribution to sample from a (much) larger range than can be expected from the tolerance. This speeds-up the inference significantly and also improves the end result (closer to the original value for R and C in the simulation).

Since the values for the resistor and capacitor (R and C) only appear as a product in the differential equal, the inference only finds a value for this product, and not for the individual values of R and C. However, because the variance of the Normal distribution of the capacitor is larger than of the resistor (to mimic the larger tolerances of capacitors), the results of the inference does favour the capacitor to deviate from the designed value.

The results of an inference on a low-pass RC-filter as shown in Figure 3-10 can be found in Figure 3-12. The design values of resistor and capacitor are 22 kΩ and 47 μF respectively, which we use to create our priors for these parameters. The prior distributions of the design values are shown in blue in Figure 3-12. One instance of this filter is simulated in with value R^*= 21 kΩ and C^*= 45 μF. Based on the simulation results the values for R and C are estimated using inference, which results in R_{est} = 21.3 kΩ (+1.4% overestimation) and C_{est} = 44.3 μF (-1.5% underestimation). Observe that the estimated value of both R and C are closer to the true values than the design values, and the posterior tolerance for C is now much smaller (1.5%) than the prior tolerance (10%), while the posterior tolerance of the resistor is marginally bigger than the design value, due to its interaction with the far more imprecise capacitor in the RC circuit.

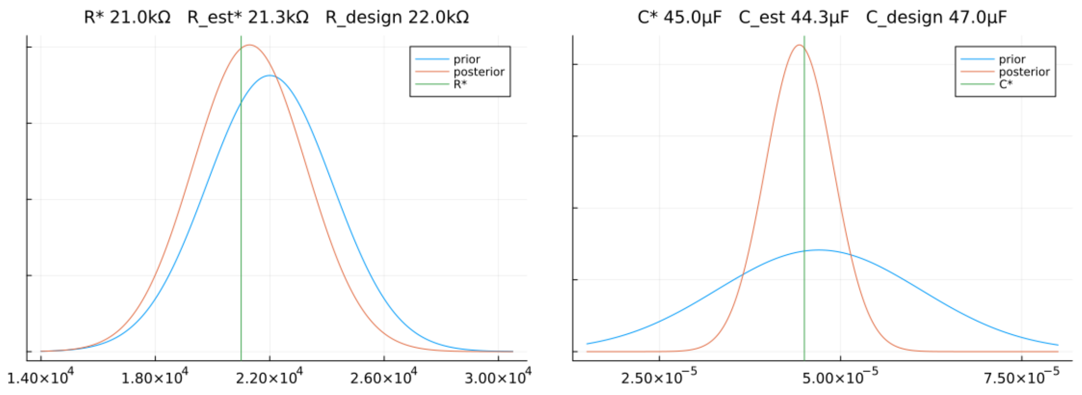

Figure 3-12 inference results of estimating the values for R and C in the RC-filter.
The actual values of the resistor and capacitor are given by R^* and C^*. The prior distribution around the designed value is shown in blue, the posterior distribution around the estimated value is plotted in red.

3.5 State machine

In this example we consider a simple probabilistic state machine with two states s_1 and s_2 and transition probabilities $p_1 = prob.transition\ s_1 \to s_2$ and $p_2 = prob.transition\ s_2 \to s_1$. With the probability of staying in a state given as $1 - p_1, p_2$ respectively for s_1, s_2. Given a sequence of N states, starting from the initial state s_1, the inference task is to infer the transition probabilities p_1, p_2. We solve this inference task using the Python library Simulation Based Inference (SBI) [15]. An advantage of SBI in respect to other probabilistic programming libraries is that inference model consists of a wrapper around the simulation model. Therefore, no additional coding is required to define the inference model.

Figure 3-13 show the results for the SBI inference for a sequence of N = 100 states with $p_1^* = 0.6$ and $p_2^* = 0.4$. The priors for the transition probabilities are uniform distributions in the range [0.3, 0.8]. The posterior for p_1 is shown in blue, on the top left plot, with in red a vertical line for p_1^*. Similarly for p_2 on the bottom plot. The top right plot shows the joint posterior distribution, with an orange dot at (p_2^*, p_1^*).

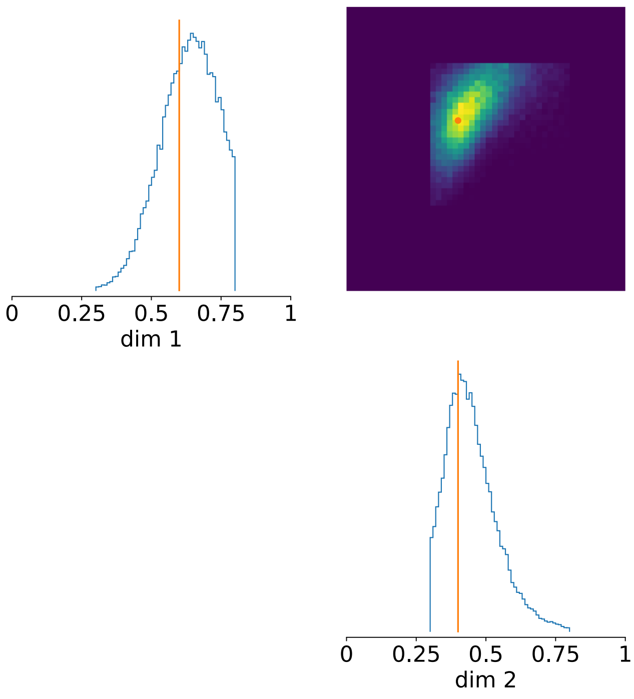

Figure 3-13 SBI inference results for transition probabilities of a probabilistic state machine, where dim 1, dim 2 correspond to p_1, p_2 respectively.

3.6 2D convolution

Image filtering, like with a blur-, sharpening-, or edge-detection filter, is usually done using a 2D convolution. In this application the input image and convolution kernel are known, and the output image is computed. Another application is to determine the convolution kernel from a known input and a measured output image, for example during the calibration of an imaging system.

In this example probabilistic programming is used to reconstruct the convolution kernel from a known input pattern and an acquired output image. The input pattern and the output image can be found in Figure 3-14. In this example the output image is simulated by applying a Gaussian blurring kernel on a slightly noisy input pattern and adding measurement noise.

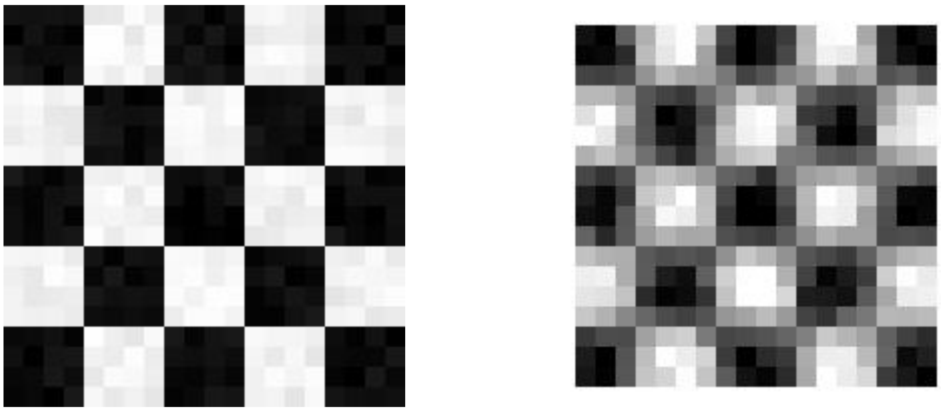
Figure 3-14 input pattern and output image used in determining the convolution kernel

NumPyro [27], a probabilistic programming library which provides a NumPy [28] backend for Pyro [9], is used to compute the convolution kernel from the kernel's shape, the input pattern and the output image. The results of the inference are shown in Figure 3-15. Figure 3-15a shows the convolution kernel coefficients used to transform the input pattern to the output image. A Uniform distribution is used as the input estimation to the inference to not give a bias to the kernel parameters, hence each coefficient is set to 1/9 as shown in Figure 3-15b. The coefficients resulting from the inference can be found in Figure 3-15c.

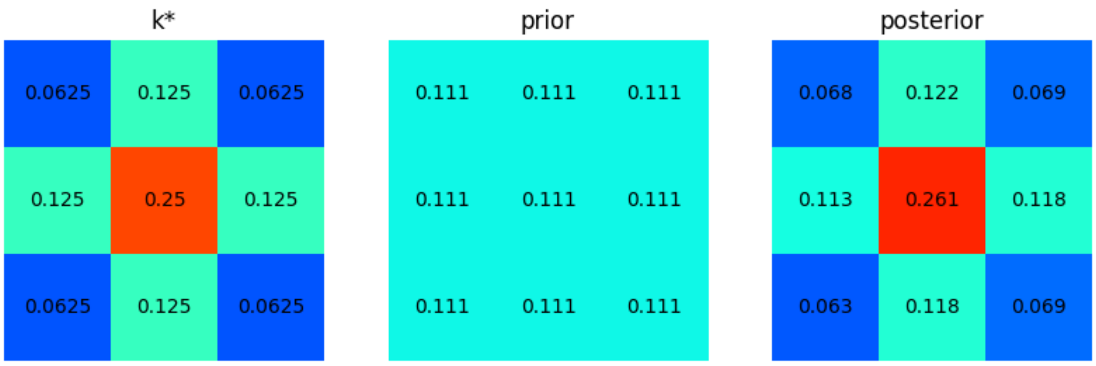

Figure 3-15 convolution kernel coefficients
[a] used in the simulation to generate the output image from the test pattern (k*)
[b] as the initial estimate for inference (prior)
[c] results after inference (posterior)

4 Diagnostic cases

4.1 Conveyor belt

In the context of the Carefree project, we have outlined a methodology for using probabilistic programs together with simulation models, and applied the methodology to a an industrial printer. For more detail, see[29]. This subsystem contains a conveyor belt that rests horizontally on four cylinders. The cylinders rotate at a variable speed and transmit this movement to the belt. In order to keep the belt at the centre of the cylinders, one of the cylinders can be tilted by raising or lowering it, see Figure 4-1.

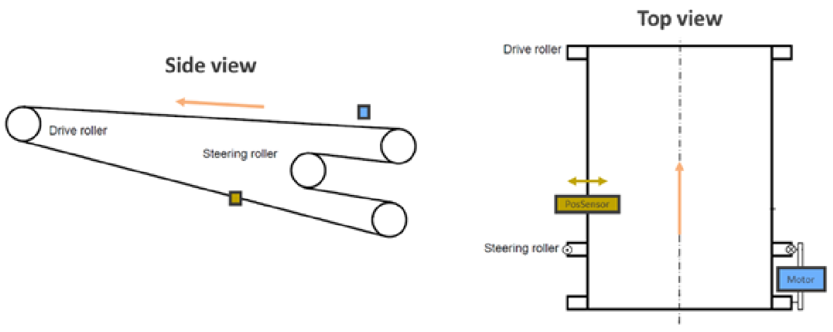

Figure 4-1: The belt lies on a horizontal plane. By raising the steering roller, the plane is tilted, and so the belt slides slightly to the side with every revolution due to gravity.

The mechanism responsible for this tilting is driven by a Z-position motor. This tilting causes the belt to slide up or down the cylinder each revolution by an amount proportional to the Z-motor position. Every few revolutions the position of the belt is measured and a correction is computed by a Proportional Integral (PI) controller, resulting in an adjustment of the Z-motor position. This steering action is necessary to counter the various causes that make the belt drift away from its intended position.

Our goal here is to discern the unknown causes of this drift and to infer their strength, given the available data on the belt and motor positions over time.

4.1.1 Simulation model

Every step of the PI-controller begins with a measurement of the belt position. This belt position must be a function of the previous belt position, the previous motor correction, and the drift incurred between the current measurement and the previous one. Based on the current positions of both the belt and Z-motor, the position of the latter is updated by a PI controller with the goal of returning the belt to its intended position. The equations modelling this behaviour are:

$$\begin{cases} belt_k = belt_{k-1} - \alpha \cdot motor_{k-1} + drift_k \\ integral_k = c_{int}(belt_k + belt_{k-1}) + integral_{k-1} \\ motor_k = c_{prop} \cdot belt_k + integral_k \end{cases}$$

Where α, c_{int}, and c_{prop} are known proportionality constants and subscripts (·)k corresponds to the value at sample k.
We conjecture that the drift results from the linear combination of five causes:

- Calibration: the belt might not be completely horizontal when the Z-motor is at position 0. This results in a constant calibration error c.
- Misalignment: the belt might not be well aligned with the previous component of the printer. This results in a constant misalignment error m that is present only when the machine is printing.
- Degradation: the belt material might wear out and deform over time, resulting in a time-dependent drift D_k. We conjecture this degradation to be exponential and with an unknown deformation direction $s \in \{-1, +1\}$ and degradation exponent δ.
- Sheets: when the pages make contact with the belt, they might cause a perturbation to its position, depending on the properties of the pages. This would result in a train of pulses P_k with varying amplitude and width, present only when the machine is printing.
- Noise: we finally conjecture that all other sources of error add up to a Gaussian term $\varepsilon_k \sim N(0, \sigma)$ with unknown variance and zero mean.

These causes are described by the following equations:

$$\begin{aligned} drift_k &= c + print_k \cdot (m + P_k) + D_k + \varepsilon_k \\ D_k &= s(4^{\delta k} - 1), \end{aligned}$$

where $print_k$ is a Boolean variable denoting whether the machine is printing at time k. In D_k, the sign parameter s determines the direction of degradation (positive or negative), and the -1 ensures that $D_{k=0} = 0$.

4.1.2 Inference model

Considering the temporal nature of our data and the controlled stepwise nature of the system, we propose a Bayesian state-space model as the probabilistic description. A Bayesian state-space model is a dynamical system of equations relating random variables. The system is determined by the observability and update equations. These equations describe how the unobserved dynamic variables (degradation D_k, perturbation P_k) evolve over time as a function of their previous state, the static variables (calibration c, misalignment m) and the observed external variables ($print_k$), and how the measured variable $drift_k$ depends on the above.

Table 4 inference results on simulated data

Parameter	Real value	Inferred value
c	15.00	15.02 ± 0.18
m	-20.00	-20.02 ± 0.20
δ	$8.00\,e^{-4}$	$8.00e^{-4} \pm 7e^{-6}$
s	1.00	1.00 ± 0.00

The inference algorithms use MCMC sampling to infer both the continuous variables $c, m, D_k, \delta, \sigma$ as well as the discrete variable $sign$. Moreover, it is possible to make future predictions. The quality of those predictions, however, depend on the quality of the underlying assumptions.

We tested the inference model against the simulation model and on real data (see Table 4 for results of inference on simulated data and Figure 4-2 for the results of inference on the real data).

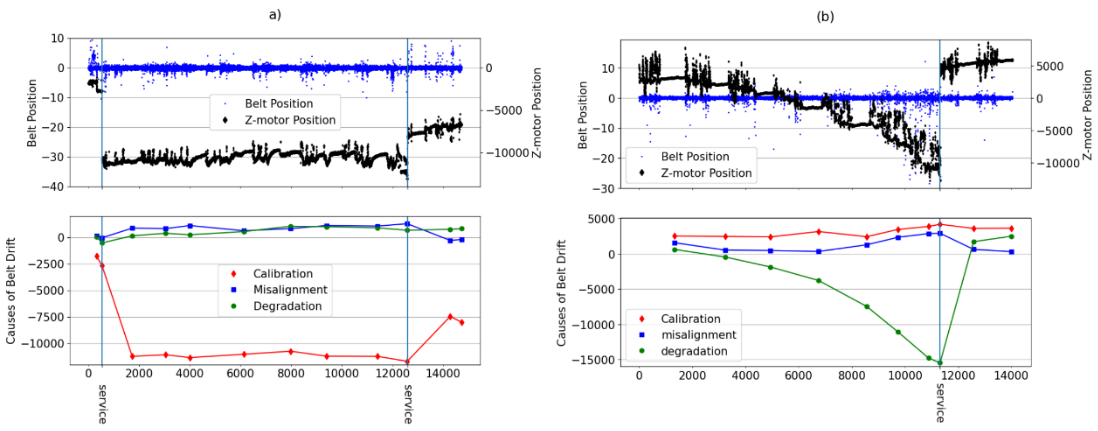

Figure 4-2 Example of measured data where miscalibration (a), and degradation (b) are the main causes of a belt position error. The Belt and Z-motor positions are measured, while the causes of belt drift in the bottom plots are inferred. The sources of drift are shown here in the units of the Z-position motor rather than the belt for comparison with the former

In this example, the simulation model is compiled using already available knowledge on failure mechanisms, together with control models, and serves a dual function. On the one hand, it helps validate the expert knowledge on failures, by comparing the results of simulations to data from incidents in the field. On the other hand, it is used to validate the inference models by providing us with a controlled test bench in which to test the ability of the inference model to distinguish the different causes of errors. The inference model is derived from the generative model and is used with field data from real incidents to perform root-cause diagnosis.

4.2 Print quality

In this case study, the printhead array of an industrial printer has a tendency to require too many service actions, which incurs unscheduled downtime costs and material replacements costs.

A printhead array, consists of four printheads for four colours: yellow, magenta, cyan, and black, laid out respectively in the direction the paper moves. Each printhead has ~10k nozzles which can be in either of three states: $\{healty, stuck, broken\}$.

At every print, each nozzle can become stuck because of dust, or become stuck as part of a cluster that appears when ink dries on the nozzle plate. Additionally, nozzles can be permanently damaged through usual wear and tear.

Every nozzle is *indirectly* measured by printing specific markers on (test) pages and scanning those pages. The result of the test is an $\{Ok, Nok\}$ value that is logged for each nozzle and each test print. Healthy nozzles are always reported as $\{Ok\}$, while stuck and broken nozzles are reported as $\{Nok\}$.

This is a dynamic system with discrete timesteps. At Every new print the state of the nozzles can potentially change. When the total number of Nok nozzles reaches a threshold (e.g. 100) the system cleans the printhead array, returning the state of $stuck$ nozzles to $healthy$, and leaving the $broken$ nozzles in the same state. Therefore, stuck nozzles can be distinguished from broken nozzles because the latter do not reset after a cleaning action.

While there are many effects that can make a nozzle Nok, for the purpose of this KIP we will focus on distinguishing $healthy$, $stuck$ and $broken$ nozzles in one printhead from their measured states and the cleaning information.

4.2.1 Simulation model

The simulation uses a sequence of hidden Markov chains to simulate the behaviour of the ~10k nozzles. The basic block for each nozzle at each time step is depicted in Figure 4-3. The parameters in the figure are defined below.

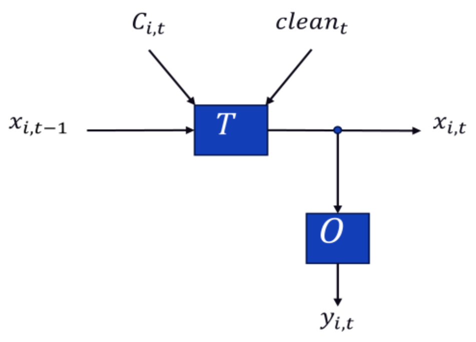

Figure 4-3: Basic building block of the hidden Markov chain used for simulating the system. The variables $clean_t$ and $y_{i,t}$ are observed, the rest are hidden. Missing is the connection of the clustering variables across nozzles for a fixed t.

The sequence of operations is as follows:

1. At the beginning of the simulation, parameters $p_s,\ p_b,\ p_u$(denoting the probabilities of transitioning from $healthy$ to $stuck$, $healthy$ or $stuck$ to $broken$ and $stuck$ to $healthy$ at any point) are sampled from prior distributions.

2. The hidden state$x_{i,t}$ of nozzle i at time t is represented by a vector of probabilities of size 3. The cluster variables $C_{i,t}$ are sampled using a Markov chain.
Every timestep, the hidden state is update by multiplying it with a 3x3 transition matrix $T_{i,t}$ that depends on the cluster variable $C_{i,t}$ and the cleaning $clean_t$.

$$T_{i,t} = \begin{cases} \begin{bmatrix} 1-p_b-p_{i,s} & p_u & 0 \\ p_{i,s} & 1-p_u-p_b & 0 \\ p_b & p_b & 1 \end{bmatrix}, & if\ clean_t = 0; \\ \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, & if\ clean_t = 1. \end{cases} \qquad \text{Equation 1}$$

Where the probability of becoming stuck $p_{i,s}$ depends on the value of the cluster variable $C_{i,t}$ at a particular nozzle location and time step as:

$$p_{i,s} = p_{i,s}(t) = \begin{cases} p_s & if\ C_{i,t} = 0 \\ 1 - p_b & if\ C_{i,t} = 1 \end{cases} .$$

These matrices capture the evolution of the nozzles between cleanings (first case) and the action of said cleanings (second case). The implicit encoding is that rows and columns correspond to $(healthy, stuck, broken)$ respectively.

It is also possible to express this operation as a single linear transformation using an order 4 tensor, rather than a matrix with nested cases.

3. The hidden state $x_{i,t}$ is multiplied by a fixed 3x2 observation matrix $O = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix}$. The result is a vector of probabilities that is sampled to obtain the observations $y_{i,t}$. Columns represent the states $(healthy, stuck, broken)$ and rows (Ok, Nok), respectively.

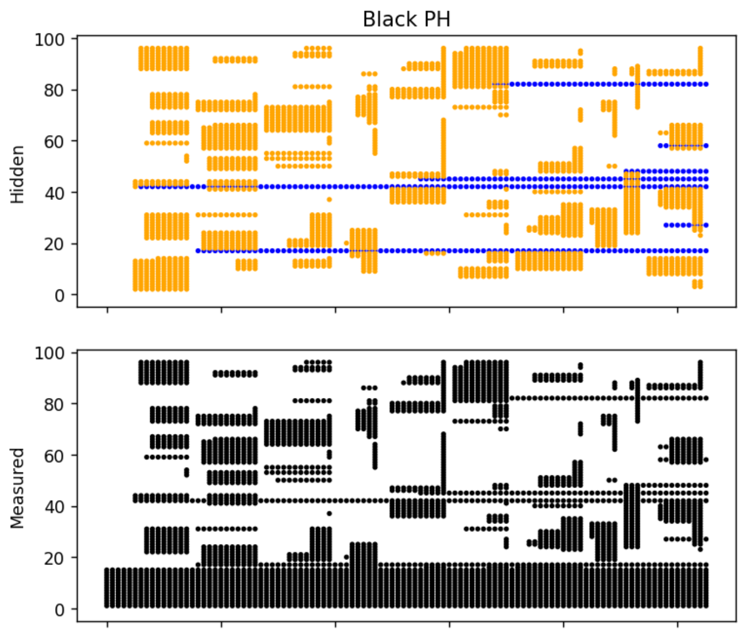

Figure 4-4: Output of one simulation with only 100 nozzles. On the vertical axis we have nozzle number. On the horizontal print number. The top plot displays the hidden states of 100 nozzles. No color for $healthy$, orange for $stuck$, blue for $broken$. The measured states are black for Nok and no color for Ok.

4.2.2 Inference model

The inference model follows closely the simulation model with only two modifications.

The model contains:

1. A set of hidden three-state categorical random variables $\{x_{i,t}, i = 1, \dots, N, t = 0, \dots, M\}$. All nozzles are initialized to be $healty$, i.e. $x_{i,0} = healthy\ \forall i$.
2. A random transition matrix T drawn from a MatrixDirichlet distribution.
3. A set of observable two-state categorical variables $\{y_{i,t},\ i = 1, \dots, N, t = 1, \dots, M\}$
4. Fixed matrices T_c and O determining the logic of cleaning and observing hidden states.
5. The logical structure of the hidden Markov chain, i.e. the identities

$$x_{i,t} = T \cdot x_{i,t-1} \quad if\ clean_t = 0$$
$$x_{i,t} = T_c \cdot x_{i,t-1} \quad if\ clean_t = 1$$
$$y_{i,t} = O \cdot x_{i,t}$$

The inference model differs from the simulation model in that there is no notion of clustering, its effects being imputed to the transition matrix T, and that there is no assumption on the internal structure of T such as in Equation 1.

The results of the inference on 500 nozzles for 100 consecutive prints can be seen in Figure 4-5. The quantities inferred here are the hidden states of the nozzles and the transition probability.
The transition matrix is not directly comparable to the transition matrix in the simulation, since it absorbs the effects of dust and clustering into one single process, but its results are consistent with those of a simulation with parameters $p_s = 0.0002$, $p_b = 0.0001$, $p_u = 0.0001$, and a probability of being in a cluster of roughly 0.004 ± 0.0045.

$$T_{est} = \begin{bmatrix} 0.9936 & 0.0003 & 0.0 \\ 0.0062 & 0.9984 & 0.0 \\ 0.0001 & 0.0013 & 1.0 \end{bmatrix}$$

The hidden nozzle states are inferred correctly with probability 99.95%.

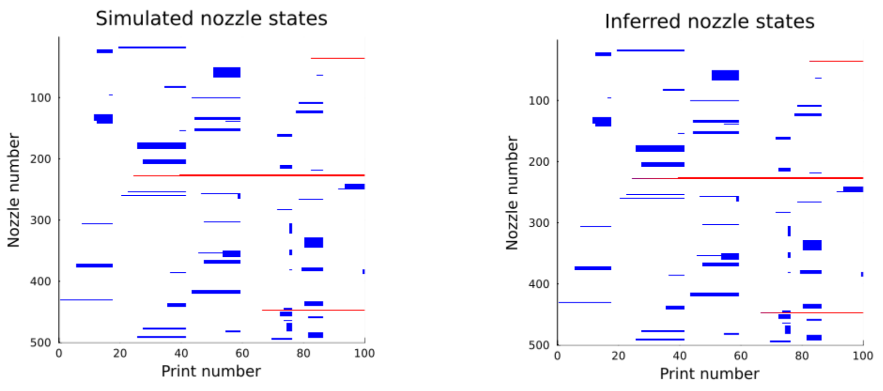

Figure 4-5: Right: simulated data, left: Result of inference on the hidden nozzle states. The accuracy is above 99.95%.

5 Summary and conclusions

Conclusions

We confirm that probabilistic programming (PP) holds potential for solving inverse problems relevant in industry, especially when simulation models are already in place. Most promising is the possibility of using PP for *quality control*, *performance diagnostics*, and *predictive maintenance*.
However, it remains unclear how PP compares to other methods for solving inverse problems (see Future Work).

PP libraries are developing rapidly, but they are not yet fully ready to be used by non-experts for engineering applications. Currently, most applications of PP are found in medicine and physics/astrophysics, with its use in engineering being relatively scarce.

There is a clear knowledge gap in industry with regards to the skills necessary for successfully integrating PP into an industrial setting. A strong collaboration with academia is advised as a means to bridge the gap.

Connecting with the previous point, there is strong interest from academia in collaborating with industry and TNO to enhance these libraries and explore their applications further. Dutch academia, in particular, is strong in probabilistic programming.

Connections to industry and future work

During this study we have identified several avenues for future work.

- *Comparison to other methods.* As stated in the introduction, comparing probabilistic programming to other methods of solving inverse problems was outside the scope of this study, yet we believe it is highly recommended to do so in order to fully assess the utility of PP for industrial applications.

- *Investigate industry needs in more detail.* In this study we have identified *predictive maintenance*, *quality control* and *performance diagnostics* of degrading and/or dynamic systems as promising applications with clear business value.

- *Connecting academia and industry on PP.* Considering the knowledge gap in industry and the relative novelty of PP libraries, industrial-academic collaboration is essential to further mature PP tools and transfer them to industry.

- *Investigate application of PP for design space exploration.* Design space exploration is usually characterized as an optimization problem, but can be also described as an inference problem when the desired properties are assumed as observed and the design parameters are treated as unknown variables connected to the former by a model. This has been considered in the literature, see [30], but not explored in this study

6 Appendix

This appendix contains the source code of the examples discussed in sections 3 and 4.

6.1 Biased coin

```
using Plots, RxInfer, Random, Distributions, LinearAlgebra
rng = MersenneTwister(42)# set seed

function simulation_model(p,n)
    data = float.(rand(rng, Bernoulli(p),n))
    return data
end

@model function inference_model(p1,p2,n)
    y = datavar(Float64, n)
    p ~ Beta(p1, p2)
    for i in 1:n
        y[i] ~ Bernoulli(p)
    end
end

function inference_routine(p,n,p1,p2)
    dataset = simulation_model(p,n)
    results = infer(model = inference_model(p1,p2,n),data = (y = dataset,))
    return results, dataset
end

n = 10
p = 0.15
p1 = 20
p2 = 100
res, data = inference_routine(p,n,p1,p2);
```

6.2 If-else models

```
from jax import random
import jax.numpy as jnp
import torch
import numpyro
import pyro

import numpyro.distributions as dist
import pyro.distributions as pyrodist
from numpyro.infer import NUTS, MCMC

def simulate(n, m0, m1, sigma, p):
    mean = torch.empty(n)
    y = torch.empty(n)
    c = torch.empty(n)
    for i in range(n):
        c[i] = pyro.sample('c_{}'.format(i), pyrodist.Bernoulli(p))
        if c[i]==0:
            mean[i] = m0
        else:
            mean[i] = m1
        y[i]=pyro.sample('y_{}'.format(i),pyrodist.Normal(mean[i],sigma))

    return {'m0': m1, 'm1': m1, 'sigma': sigma, 'c':c, 'p':p,
            'observed': y.numpy()}

# Run simulation.
sim_data = simulate(2000, -4, 4, 2, 0.4)

def model(data):
    # Global variables.
    p = numpyro.sample('p',dist.Beta(1,1))
    sigma = numpyro.sample("sigma", dist.HalfNormal(5.0))
    with numpyro.plate("components", 2):
        locs = numpyro.sample("locs", dist.Normal(0.0, 15.0))

    with numpyro.plate("data", len(data)):
        # Local variables.
        assignment = numpyro.sample("assignment", dist.Bernoulli(p))
        numpyro.sample("obs", dist.Normal(locs[assignment], sigma),
                       obs=data)
## Inference
Kernel = NUTS(model)
mcmc = MCMC(kernel,num_warmup=5000, num_samples=5000, num_chains=2)
mcmc.run(random.PRNGKey(1), jnp.array(sim_data['observed']))
mcmc.print_summary()
```

6.3 Mass-spring-damper system

```
using DifferentialEquations
using Plots
using Turing
using LinearAlgebra

function simulation_model(ddu, du, u, p, t)
    p1, p2 = p
    ddu .= -p1*u -p2*du
end

@model function inference_model(data::AbstractVector, prob, priors)
    # Prior distributions.
    σ ~ InverseGamma(priors[1], priors[2])
    p1 ~ Uniform(priors[3], priors[4])
    p2 ~ Uniform(priors[5], priors[6])

    # Simulate model.
    p = [p1,p2]
    predicted = solve(prob, Tsit5(); p=p,
                          saveat=delta_t, save_idxs=dimension_interest)
    # Observations.
    data ~ MvNormal(predicted.u, σ^2 * I)
    return nothing
end

function main_routine(dx0, x0, tspan, p, sigma,priors)
    # Generate data.
    prob = SecondOrderODEProblem(simulation_model, dx0, x0, tspan, p)
    sol = solve(prob; saveat=delta_t)
    data = Array(sol[dimension_interest,:]) +
                 sigma * randn(size(Array(sol[dimension_interest,:])))
    # we only measure the position

    # Inference
    model = inference_model(data, prob, priors)
    # Sample 3 independent chains.
    results = sample(model, NUTS(0.35), MCMCSerial(),
                                                 length_MC, 3; progress=false)
    return results,data
end

# Only consider position
dimension_interest = 2
# Parameters [k,c]
p_star = [0.6,0.3]
sigma_star = 0.3
# Initial Conditions
x0 = [.5]
dx0 = [0.5]
# Time
tspan = (0.0, 20)
delta_t = 0.1

# Priors
sigma_p1 = 2
sigma_p2 = 3
p1_min = 0
p1_max = 2
p2_min = p1_min
p2_max = p1_max
priors = [sigma_p1,sigma_p2,p1_min,p1_max,p2_min,p2_max]

# Markov chain parameters
length_MC = 100
results,data = main_routine(dx0, x0, tspan, p_star, sigma_star,priors)
```

6.4 RC filter

```
using DifferentialEquations
using Interpolations
using LinearAlgebra
using Turing
```

```
# Initial conditions and input signal
Vout0 = [0.0]
tspan = (0.0, 10);
input_a = zeros(50)
input_a[5:30] .= 1.0
xs = 0:(10.0/49):10
input_f = Interpolations.scale(
                            interpolate(input_a, BSpline(Linear())), xs);

# Design parameters, i.e. typical values
R = 22E3   # 22 kΩ
C = 47E-6   # 47 μF
p = (R, C, input_f)

# Function to compute output of RC-filter using a differential equation
function simulation_model(dVout, Vout, p, t)
    R, C, Vin = p
    dVout .= (1.0*Vin(t) .- Vout) / (R * C)
end

# Function to fit a model for R and C
@model function inference_model(data, prob)
    σ = 0.1;
    p1 ~ truncated(Normal(R, 0.10*R); lower=0.01*R, upper=10*R)
    p2 ~ truncated(Normal(C, 0.30*C); lower=0.01*C, upper=10*C)

    # Simulate model.
    p = (p1, p2, input_f)
    predicted = solve(prob; p=p, saveat=0.1)

    # Observations.
    for i in 1:length(predicted)
        data[:, i] ~ MvNormal(predicted[i], σ^2 * I)
    end

    return nothing
end

function inference_routine(data)
    p = (R, C, input_f)
    prob = ODEProblem(simulation_model, Vout0, tspan, p)
    model = inference_model(data, prob)
    chain = sample(model, NUTS(0.65), MCMCSerial(), 200, 3);

    R_est = mean(get(chain, :p1)[:p1])
    R_std = std(get(chain, :p1)[:p1])
    C_est = mean(get(chain, :p2)[:p2])
    C_std = std(get(chain, :p2)[:p2])
    return (R_est, R_std, C_est, C_std)
end

# Create an ODE of the RC-filter, use it to simulate output with noise
r_star = 21E3 # actual value of the resistor
c_star = 45E-6 # actual value of the capacitor
p_star = (r_star, c_star, input_f)
```

```
prob = ODEProblem(simulation_model, Vout0, tspan, p_star)
sol = solve(prob, Tsit5(); saveat=0.1)
simdata = Array(sol) + 0.05 * randn(size(Array(sol)))

# Use the output of the simulation to estimate the value of R* and C*
(R_est, R_std, C_est, C_std) = inference_routine(simdata)
```

6.5 2D convolution

```
import numpyro as npr
import numpy as np
import jax.numpy as jnp
from jax import random
import random as rnd

def simulation_model(image: np.ndarray, kernel: np.ndarray, var_noise=0):
    image_size = image.shape
    kernel_size = kernel.shape
    kernel_length = kernel_size[0] * kernel_size[1]
    output_size = tuple(image_size[n] - kernel_size[n] + 1 \
                                            for n in range(image.ndim))
    output_length = output_size[0] * output_size[1]
    proc_image = np.zeros([output_size[0], output_size[1],
                           kernel_size[0], kernel_size[1]])
    conv_image = np.zeros([output_size[0], output_size[1]])
    for y in range(output_size[0]):
        for x in range(output_size[1]):
            proc_image[y][x] = image[y:y+kernel_size[0],x:x+kernel_size[1]]
            conv_image[y][x] = np.sum(proc_image[y][x] * kernel + \
                                [[rnd.normalvariate(0,var_noise)
                                    for xx in range(kernel_size[1])]
                                    for yy in range(kernel_size[0])])

    proc_image = proc_image.reshape([output_length, kernel_length])
    conv_image = conv_image.reshape([output_length])
    return proc_image, conv_image

def inference_model(data):
    signal,convolved_signal,kernel_size = data[0],data[1],data[2]
    k = [npr.sample('param_' + chr(ord('a') + i),
    npr.distributions.Uniform(low=-3, high=3)) for i in range(kernel_size)]
    kernel = jnp.array(k, dtype=jnp.float32)
    with npr.plate('data', len(signal), dim=-2):
        return npr.sample('obs', npr.distributions.Normal( \
                jnp.sum(signal*kernel, axis=1), 0.1), obs=convolved_signal)

def inference_routine(data, warmup=100, samples=1500, chains=2):
    kernel_shape = data[3]
    npr.set_host_device_count(chains)
    kernel = npr.infer.NUTS(inference_model)
    mcmc = npr.infer.MCMC(kernel, num_warmup=warmup, num_samples=samples, \
                          num_chains=chains)
    mcmc.run(key, data)
    samples = mcmc.get_samples()
    mu = np.asarray([np.mean(samples[k]) for k in samples])
    mu = mu.reshape(kernel_shape)
    var = np.asarray([np.var(samples[k]) for k in samples])
    var = var.reshape(kernel_shape)
    return mu, var

# Create a checkerboard pattern
L=4
N=5 * L
image = np.random.rand(N,N) / 10.0
for i in range(0, N-L+1, 2*L):
    image[i:i+L, :] = 1 - image[i:i+L, :]
    image[:, i:i+L] = 1 - image[:, i:i+L]

# Create a Gaussian blurring kernel for simulation
img_kernel = np.ones([3,3])
img_kernel[0,1] = 2
img_kernel[2,1] = 2
img_kernel[1,0] = 2
img_kernel[1,2] = 2
img_kernel[1,1] = 4
img_kernel = img_kernel/np.sum(img_kernel)

# Create simulated output, including measurement noise
proc_image, conv_image = simulation_model(image, img_kernel,
var_noise=0.02)

# Run inference to determine convolution kernel coefficients
data = [jnp.array(proc_image, dtype=jnp.float32),\
        jnp.array(conv_image, dtype=jnp.float32), \
        img_kernel.size, \
        img_kernel.shape]
kernel_mean, kernel_var = inference_routine(data)
```

6.6 State machine

```
import torch
import numpy as np
from sbi import analysis as analysis
from sbi import utils as utils
from sbi.inference import SNPE, simulate_for_sbi, infer
from sbi.utils import MultipleIndependent
from torch.distributions import Uniform

def simulation_model(params):
    p1 = params[0].item()
    p2 = params[1].item()

    p_transition = np.array([[1-p1, p1], [p2, 1-p2]])
    states = [initial_state]

    for _ in range(markov_chain_length - 1):
        current_state = states[-1]
        probs = p_transition[current_state]
```

```
            new_state = np.random.choice(
                        np.linspace(0,np.shape(probs)[0]-1,np.shape(probs)[0]),
                        size=1, p= probs ).astype(int)
            states.append(new_state[0])
        return torch.as_tensor(states, dtype = torch.float32)

markov_chain_length = 100
initial_state = 0

#real parameters for simulation
p1_star = 0.6
p2_star = 0.4
p_star = torch.as_tensor([p1_star,p2_star])

#definition of prior
prior = utils.BoxUniform(low=0.3 * torch.ones(2), high=.8 * torch.ones(2))

#inference model
inference_model = infer(simulation_model, prior,
                                      method="SNPE", num_simulations=1000)
#main routine
posteriors_samples = 40000
data_star = simulation_model(p_star)
samples = inference_model.sample((posteriors_samples,), x=data_star)
```

6.7 Conveyor belt

```
def Conveyor_belt_1(y, printing, T0, T1, future=0):
  # Standard deviation of the noise introduced at each step.
  sigma = numpyro.sample("sigma", dist.HalfNormal(5))
  sign_param = numpyro.sample("sign_param", dist.Bernoulli(0.5))
  sign = (2*sign_param)-1
  decay_exp = numpyro.sample("decay_exp", dist.Uniform(0.000, 0.5))
  calibration = numpyro.sample("calibration", dist.Uniform(-30, 30))
  misalign_ampl = numpyro.sample("misalignment", dist.Uniform(-30, 30))

  def transition_fn(carry, t):
    # Update equation
    degradation = carry  # Designate hidden vars
    degradation = ((degradation + sign) * (4 ** (decay_exp / 100)) - sign)

    # Observability equation
    mu = calibration + degradation + printing[t] * (misalign_ampl)
    y_ = numpyro.sample("y", dist.Normal(mu, sigma))

    return degradation, y_

  with numpyro.handlers.condition(data={"y": y[T0+1:T1]}):
    _, ys = scan(transition_fn,
                 (sign * (4 ** (decay_exp * T0 / 100) - 1)),
                 jnp.arange(T0 + 1, T1+future), )

    if future > 0:
        numpyro.deterministic("y_forecast", ys[-future:])
```

6.8 Print quality

6.8.1 Simulation

```python
import matplotlib
matplotlib.use('Qt5Agg')
import numpy as np
import matplotlib.pyplot as plt
import warnings
warnings.simplefilter(action='ignore', category=DeprecationWarning)

prob_start_clust = 0.001
prob_stop_cluster = 0.2
prob_stuck = 0.0002
prob_break = 0.0001
prob_unstuck = 0.0001

initial_cluster_state = 0

OK = 0
stuck = 1
broken = 2

working = 0
not_working = 1

N = 1000
P = 100
cleaning_treshold = 100

def get_transition_matrix_2_states(p11,p22):
    p_stay = np.array([[1-p11, p11], [p22, 1-p22]])
    return p_stay

def markov_sequence(p_transition, sequence_length, initial_state):
    states = [initial_state]
    for _ in range(sequence_length - 1):
        current_state = states[-1]
        probs = p_transition[current_state]
        new_state = np.random.choice(np.linspace(0,np.shape(probs)[0]-1,
                         np.shape(probs)[0]), size=1, p=probs).astype(int)
        states.append(new_state[0])
    return states

def cluster(p1,p2,sequence_length, initial_state):
    return markov_sequence(get_transition_matrix_2_states(p1, p2),
                                           sequence_length, initial_state)

def measuring_nozzles(hidden_states):
    measured_states = hidden_states.copy()
    for idx,el in enumerate(hidden_states):
        measured_states[idx] = working if el==OK else not_working
    return measured_states
```

```
def single_print(prev_hidden_states,prev_clean):
    current_hidden_states = prev_hidden_states.copy()
    c = cluster(prob_start_clust,prob_stop_cluster,N,initial_cluster_state)
    c_probs = [prob_stuck if x == 0 else 1-prob_break for x in c]
    if prev_clean:
        for idx,el in enumerate(prev_hidden_states):
            if el == broken:
                current_hidden_states[idx] = broken
            else: #OK or stuck goes back to OK
                current_hidden_states[idx] = OK
    if not prev_clean:
        for idx,el in enumerate(prev_hidden_states):
            c_prob=c_probs[idx]
            if el == OK:
                current_hidden_states[idx] = np.random.choice(
[OK, stuck, broken], size=1, p=[1 - prob_break - c_prob, c_prob,
prob_break])
            if el == stuck:
                current_hidden_states[idx] = np.random.choice([OK, stuck,
broken], size=1, p=[prob_unstuck, 1 - prob_unstuck - prob_break,
prob_break])
            if el == broken:
                current_hidden_states[idx] = broken

    measured_hidden_states = measuring_nozzles(current_hidden_states)
    if sum(measured_hidden_states)> cleaning_treshold:
        current_clean = True
    else:
        current_clean = False
    return current_hidden_states,current_clean,measured_hidden_states,c


def prints():
    hidden_states = np.zeros((P,N))
    measured_states = np.zeros((P,N))
    cleanings = np.zeros(P)

    prev_hidden_states = np.zeros(N)
    prev_clean = False

    for p in range(P):
        current_hidden_states, current_clean, current_measured_states,_ =
single_print(prev_hidden_states,prev_clean)
        hidden_states[p] = current_hidden_states
        measured_states[p] = current_measured_states
        cleanings[p] = current_clean

        prev_hidden_states = current_hidden_states
        prev_clean = current_clean

    return hidden_states,measured_states,cleanings
```

6.8.2 Inference

```julia
using Pkg
Pkg.activate(".")
Pkg.instantiate()
using RxInfer

# Transition matrix prior statistics:
# Assumes that broken nozzles can't spontaneously repair;
# Transitions are assumed unknown otherwise.
A_T = [1.0 1.0 0.01;
       1.0 1.0 0.01;
       1.0 1.0 100.0]

# Known transition matrix when cleaning:
# Assumes that stuck nozzles always become operational after cleaning;
# Broken nozzles always remain broken.
T_c = [1.0 1.0 0.0;
       0.0 0.0 0.0;
       0.0 0.0 1.0]

# Observation matrix:
# Assumes that stuck and broken nozzles are always detected as failures.
O = [1.0 0.0 0.0; 0.0 1.0 1.0]

@model function batch(M, N)
    h_0 = randomvar(N) # Initial state per nozzle
    h   = randomvar(M, N) # Hidden states per nozzle over time
    w   = datavar(Vector{Float64}, M, N) # Obs. states per nozzle over time

    T ~ MatrixDirichlet(A_T) # Transition matrix prior
    for n = 1:N # For each nozzle
        h_0[n] ~ Categorical([1.0, 0.0, 0.0]) # initially operational

        h_min = h_0[n]
        for m = 1:M # For each timepoint
            if cleaning[m]
                # Transition model is known when cleaning
                h[m, n] ~ Transition(h_min, T_c)
            else
                # Transition model under matrix T
                h[m, n] ~ Transition(h_min, T)
            end
            w[m, n] ~ Transition(h[m, n], O) # Observation model

            h_min = h[m, n] # Reset previous state
        end
    end
end

# Assume a structured factorization of the free energy
constraints = @constraints begin
    q(h_0, h, T) = q(h_0, h)q(T)
end
```

```
# Initialization for the iterative variational Bayes algorithm
initmarginals = (T=MatrixDirichlet(A_T),)

# Keep only the posteriors at the last iteration
returnvars = (h = KeepLast(),
              T = KeepLast())

n_iterations = 5; # Number of iterations of the variational algorithm

result = inference(
    model         = batch(M, N),
    data          = (w = m_one_hot,),
    constraints   = constraints,
    initmarginals = initmarginals,
    returnvars    = returnvars,
    iterations    = n_iterations,
    free_energy   = true
);
```

7 Acknowledgements

The research is carried out as part of the Kennisinvesteringsproject (KIP) Feasibility study of probabilistic programming applications in industrial engineering, under the responsibility of TNO-ESI. The research is supported by the Netherlands Organisation for Applied Scientific Research TNO.

8 References

[1] J.-W. van de Meent, B. Paige, H. Yang, and F. Wood, “An Introduction to Probabilistic Programming,” 2018, [Online]. Available: http://arxiv.org/abs/1809.10756

[2] A. Kucukelbir, D. M. Blei, A. Gelman, R. Ranganath, and D. Tran, “Automatic Differentiation Variational Inference,” *J. Mach. Learn. Res.*, vol. 18, pp. 1–45, 2017.

[3] M. Cox, T. van de Laar, and B. de Vries, “A factor graph approach to automated design of Bayesian signal processing algorithms,” *Int. J. Approx. Reason.*, vol. 104, pp. 185–204, 2019, doi: 10.1016/j.ijar.2018.11.002.

[4] A. G. Baydin *et al.*, “Etalumis: Bringing probabilistic programming to scientific simulators at scale,” *Int. Conf. High Perform. Comput. Networking, Storage Anal. SC*, 2019, doi: 10.1145/3295500.3356180.

[5] D. Wingate, A. Stuhlmüller, and N. D. Goodman, “Lightweight implementations of probabilistic programming languages via transformational compilation,” *J. Mach. Learn. Res.*, vol. 15, pp. 770–778, 2011.

[6] T. A. Le, A. G. Baydin, and F. Wood, “Inference compilation and universal probabilistic programming,” *Proc. 20th Int. Conf. Artif. Intell. Stat. AISTATS 2017*, 2017.

[7] D. Ritchie, B. Mildenhall, N. D. Goodman, and P. Hanrahan, “Controlling procedural modeling programs with stochastically-ordered sequential Monte Carlo,” *ACM Trans. Graph.*, vol. 34, no. 4, 2015, doi: 10.1145/2766895.

[8] Stan Development Team, “Stan modeling language users guide and reference manual, version 2.34.” 2024. [Online]. Available: https://mc-stan.org/

[9] E. Bingham *et al.*, “Pyro: Deep universal probabilistic programming,” *J. Mach. Learn. Res.*, vol. 20, no. Xxxx, pp. 0–5, 2019.

[10] D. Bagaev, A. Podusenko, and B. de Vries, “RxInfer: A Julia package for reactive real-time Bayesian inference,” *J. Open Source Softw.*, vol. 8, no. 84, p. 5161, 2023, doi: 10.21105/joss.05161.

[11] W. R. Gilks, A. Thomas, D. J. Spiegelhalter, and W. R. Gilkst, “A Language and Program for Complex Bayesian Modelling,” 1992.

[12] B. Carpenter *et al.*, “Stan: A probabilistic programming language,” *J. Stat. Softw.*, vol. 76, no. 1, 2017, doi: 10.18637/jss.v076.i01.

[13] H. Ge, K. Xu, M. Trapp, M. Tarek, C. Pfiffer, and T. Fjelde, “Turing.jl.” 2024. [Online]. Available: https://turinglang.org/

[14] M. F. Cusumano-Towner, A. K. Lew, F. A. Saad, and V. K. Mansinghka, “Gen: A general-purpose probabilistic programming system with programmable inference,” *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implement.*, pp. 221–236, 2019, doi: 10.1145/3314221.3314642.

[15] A. Tejero-Cantero *et al.*, “SBI -- A toolkit for simulation-based inference,” Jul. 2020, [Online]. Available: http://arxiv.org/abs/2007.09114

[16] C. Lamont, F. Mazza, and N. Donaldson, “A Bayesian Demonstration of Reliability for Encapsulated Implanted Electronics,” *Int. IEEE/EMBS Conf. Neural Eng. NER*, vol. 2019-March, pp. 730–733, 2019, doi: 10.1109/NER.2019.8717034.

[17] Q. Meng, Y. Qian, L. Li, and L. Wang, “Data analysis on incomplete failure of ship electromechanical system based on Bayesian method,” *2017 Progn. Syst. Heal. Manag. Conf. PHM-Harbin 2017 - Proc.*, pp. 3–9, 2017, doi: 10.1109/PHM.2017.8079220.

[18] K. Tada, “Accelerating Bayesian Estimation of Solar Cell Equivalent Circuit Parameters Using JAX-Based Sampling,” *Electron.*, vol. 12, no. 17, 2023, doi: 10.3390/electronics12173631.

[19] J. Sun, J. M. Hoekstra, and J. Ellerbroek, “Aircraft drag polar estimation based on a stochastic hierarchical model,” in *SESAR Innovation Days*, 2018.

[20] T. B. Schön, A. Svensson, L. Murray, and F. Lindsten, “Probabilistic learning of nonlinear dynamical systems using sequential Monte Carlo,” *Mech. Syst. Signal Process.*, vol. 104, pp. 866–883, 2018, doi: 10.1016/j.ymssp.2017.10.033.

[21] D. B. Steffelbauer, R. E. M. Riva, J. S. Timmermans, J. H. Kwakkel, and M. Bakker, “Evidence of regional sea-level rise acceleration for the North Sea,” *Environ. Res. Lett.*, vol. 17, no. 7, 2022, doi: 10.1088/1748-9326/ac753a.

[22] O. Abril-Pla *et al.*, “PyMC: A Modern and Comprehensive Probabilistic Programming Framework in Python.” 2023. [Online]. Available: https://www.pymc.io/

[23] K. Tamaki, S. Kawabe, T. Takahashi, and H. Matsue, “A Probabilistic Method for Constructing an Empirical Discrimination Model for Hammering Inspection of Cast-Iron Parts,” *SICE J. Control. Meas. Syst. Integr.*, vol. 12, no. 6, pp. 228–236, 2019, doi: 10.9746/jcmsi.12.228.

[24] T. Wang *et al.*, “A DPIM-based probability analysis framework to obtain railway vehicle vibration characteristics considering the randomness of OOR wheel,” *Probabilistic Eng. Mech.*, vol. 75, no. July 2023, p. 103587, 2024, doi: 10.1016/j.probengmech.2024.103587.

[25] A. Boyali, S. Thompson, and D. R. Wong, “Identification of Vehicle Dynamics Parameters Using Simulation-based Inference,” *IEEE Intell. Veh. Symp. Proc.*, pp. 306–312, 2021, doi: 10.1109/IVWorkshops54471.2021.9669252.

[26] C. Rackauckas and Q. Nie, “DifferentialEquations.jl – A Performant and Feature-Rich Ecosystem for Solving Differential Equations in Julia,” *J. Open Res. Softw.*, vol. 5, no. 1, p. 15, 2017, doi: 10.5334/jors.151.

[27] D. Phan, N. Pradhan, and M. Jankowiak, “Composable Effects for Flexible and Accelerated Probabilistic Programming in NumPyro,” pp. 1–10, 2019, [Online]. Available: http://arxiv.org/abs/1912.11554

[28] C. R. Harris *et al.*, “Array programming with NumPy,” *Nature*, vol. 585, no. 7825, pp. 357–362, 2020, doi: 10.1038/s41586-020-2649-2.

[29] A. Piedrafita and L. Barbini, “Leveraging Generative and Probabilistic Models for Diagnostics of Cyber-Physical Systems,” PHM Society European Conference, 2024, pp. 605–611. [Online]. Available: https://doi.org/10.36001/phme.2024.v8i1.4055

[30] T. Rainforth, “Automating Inference, Learning, and Design using Probabilistic Programming,” 2017.