PLOS ONE

RESEARCH ARTICLE

Evaluating probabilistic programming languages for simulating quantum correlations

Abdul Karim Obeid[1☯]*, **Peter D. Bruza**[1☯], **Peter Wittek**[2,3,4,5☯]

1 School of Information Systems, Queensland University of Technology, Brisbane, Queensland, Australia, **2** Rotman School of Management, University of Toronto, Toronto, Ontario, Canada, **3** Creative Destruction Lab, Toronto, Ontario, Canada, **4** Vector Institute for Artificial Intelligence, Toronto, Ontario, Canada, **5** Perimeter Institute for Theoretical Physics, Toronto, Ontario, Canada

☯ These authors contributed equally to this work.
* abdul.obeid@hdr.qut.edu.au

Citation: Obeid AK, Bruza PD, Wittek P (2019) Evaluating probabilistic programming languages for simulating quantum correlations. PLoS ONE 14(1): e0208555. https://doi.org/10.1371/journal.pone.0208555

Editor: Marco Barbieri, Universita degli Studi Roma Tre, ITALY

Received: September 1, 2018

Accepted: November 17, 2018

Published: January 4, 2019

Data Availability Statement: Source code implemented for the publication of the manuscript can be found at the following open-source repository: https://github.com/askoj/bell-ppls.

Funding: This research was supported by the Asian Office of Aerospace Research and Development (AOARD) grant: FA2386-17-1-4016. This research was supported by Perimeter Institute for Theoretical Physics. Research at Perimeter Institute is supported by the Government of Canada through Industry Canada and by the Province of Ontario through the Ministry of

Abstract

This article explores how probabilistic programming can be used to simulate quantum correlations in an EPR experimental setting. Probabilistic programs are based on standard probability which cannot produce quantum correlations. In order to address this limitation, a hypergraph formalism was programmed which both expresses the measurement contexts of the EPR experimental design as well as associated constraints. Four contemporary open source probabilistic programming frameworks were used to simulate an EPR experiment in order to shed light on their relative effectiveness from both qualitative and quantitative dimensions. We found that all four probabilistic languages successfully simulated quantum correlations. Detailed analysis revealed that no language was clearly superior across all dimensions, however, the comparison does highlight aspects that can be considered when using probabilistic programs to simulate experiments in quantum physics.

Introduction

Probabilistic models are used in a broad swathe of disciplines ranging from the social and behavioural sciences, biology, the physical and computational sciences, to name but a few. At their very core, probabilistic models are defined in terms of random variables, which range over a set of outcomes that are subject to chance. For example, a measurement on a quantum system is a random variable. By performing the measurement, we record the outcome as the value of the random variable. Repeated measurements on the same preparation allow determining the probability of each outcome. Probabilistic programming offers a convenient way to express probabilistic models by unifying techniques from conventional programming such as modularity, imperative or functional specification, as well as the representation and use of uncertain knowledge. A variety of probabilistic programming languages (PPLs) have been proposed (see [1] for references), which have attracted interest from artificial intelligence, programming languages, cognitive science, and the natural language processing communities [2].

Economic Development and Innovation. The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript.

Competing interests: The authors have declared that no competing interests exist.

However, as far as the authors can tell, there has been little interest in PPLs from the physics research community. The aim of this article is raise awareness of PPLs to this community by showing how quantum correlations can be simulated by probabilistic programming. The core purpose of a PPL is to specify a model in terms of random variables and probability distributions [1]. As a consequence, a PPL is restricted to computing statistical correlations between variables which are a mathematical consequence of the underlying event space. Quantum theory, on the the hand, has a different underlying event space. This in turn allows correlations between variables to emerge that go beyond those governed by standard probability theory. In particular, local hidden variables are straightforward to represent in a PPL, since they correspond to what classical probabilities can express. Nonlocal correlations, however, cannot be described by a local hidden variable model [3]. The question arises as to how to simulate such correlations using probabilistic programming. This article addresses this question by using a hypergraph formalism that has recently emerged in quantum information [4]. The advantage of the hypergraph formalism is that it provides a flexible, abstract representation for rendering into the syntax of a PPL. In addition, constraints inherent to the experimental design being simulated can be structurally expressed within the hypergraphs. We will show that by embedding this hypergraph formalism in a PPL, an EPR experiment can be simulated where quantum correlations are produced (the acronym EPR describes Einstein, Rosen and Podolky's famous paper which subsequently led to experimental protocols being developed to investigate quantum entanglement [5]). In addition, we provide qualitative and quantitative comparisons between several implementations in contemporary PPLs under an open source license. This opens the door to the possibility of reliably and meaningfully simulating experiments in quantum contextuality by means of probabilistic programs.

Probabilistic Programming and the EPR experiment

The basis of the EPR experiment is two systems A and B which are represented as bivalent variables ranging of $\{0, 1\}$. Variables A and B are respectively conditioned by bivalent variables X and Y, with both ranging over $\{0, 1\}$. Four experiments are performed by means of joint measurements on A and B depending on the value of the respective conditioning variables. As a consequence, the experiments produce four pairwise distributions over the four possible outcomes from the joint measurements:

$$p(A, B|X = 0, Y = 0)$$
$$p(A, B|X = 0, Y = 1)$$
$$p(A, B|X = 1, Y = 0)$$
$$p(A, B|X = 1, Y = 1)$$

In order to simplify the notation, variable A_i is distributed as $p(A|X = i)$, $i \in \{0, 1\}$. In a similar way, variables B_0 and B_1 are introduced. Therefore, the preceding four pairwise distributions can be represented as the the grid of sixteen probabilities depicted in Fig 1. The EPR experiment is subject to constraint know as the "no-signalling" condition. No-signalling entails that the marginal probabilities observed in relation to one variable do not vary according to how the other variable is conditioned:

$$p_1 + p_2 = p_5 + p_6 \tag{1}$$

$$p_9 + p_{10} = p_{13} + p_{14} \tag{2}$$

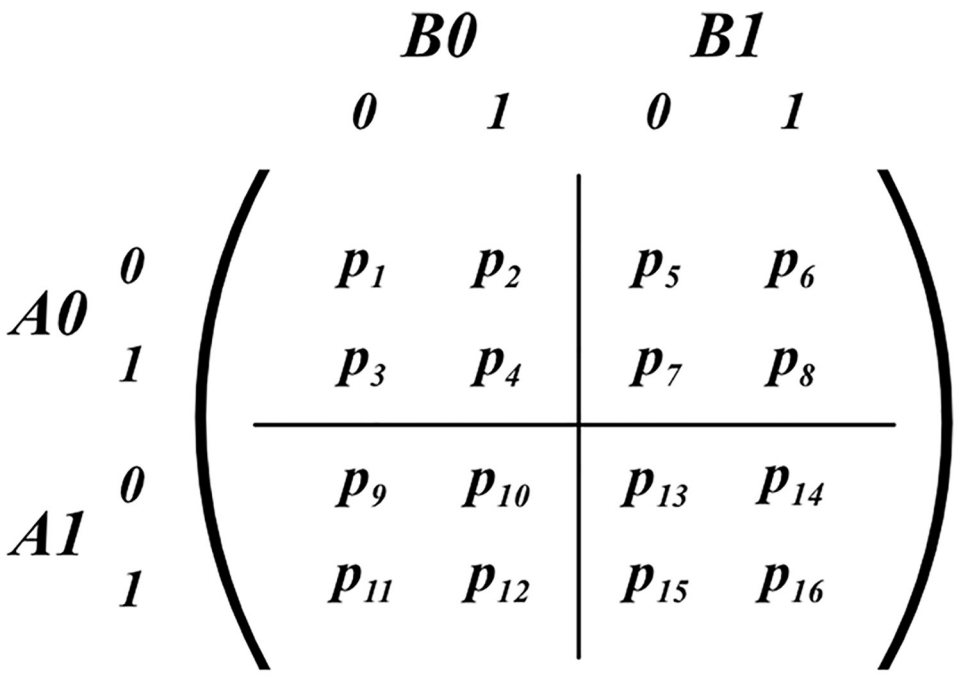

Fig 1. Four pairwise distributions in an EPR experiment.

https://doi.org/10.1371/journal.pone.0208555.g001

$$p_1 + p_3 = p_9 + p_{11} \tag{3}$$

$$p_5 + p_7 = p_{13} + p_{15} \tag{4}$$

The goal of an EPR experiment is to empirically determine whether quantum particles are entangled. We will not go into the details of what entanglement is, but rather focus on showing how statistical correlations between variables determine the presence of entanglement. Entanglement is determined if any of the following inequalities is violated.

$$|\langle A_0B_0\rangle + \langle A_0B_1\rangle + \langle A_1B_0\rangle - \langle A_1B_1\rangle| \leq 2 \tag{5}$$

$$|\langle A_0B_0\rangle + \langle A_0B_1\rangle - \langle A_1B_0\rangle + \langle A_1B_1\rangle| \leq 2 \tag{6}$$

$$|\langle A_0B_0\rangle - \langle A_0B_1\rangle + \langle A_1B_0\rangle + \langle A_1B_1\rangle| \leq 2 \tag{7}$$

$$|-\langle A_0B_0\rangle + \langle A_0B_1\rangle + \langle A_1B_0\rangle + \langle A_0B_0\rangle| \leq 2 \tag{8}$$

where the correlations are defined as follows:

$$\langle A_0B_0\rangle = (p_1 + p_4) - (p_2 + p_3) \tag{9}$$

$$\langle A_0B_1\rangle = (p_5 + p_8) - (p_6 + p_7) \tag{10}$$

$$\langle A_1B_0\rangle = (p_9 + p_{12}) - (p_{10} + p_{11}) \tag{11}$$

$$\langle A_1B_1\rangle = (p_{13} + p_{16}) - (p_{14} + p_{15}) \tag{12}$$

For historical reasons, the set of four inequalities have become known as the Clauser-Horn-Shimony-Holt (CHSH) inequalities [6]. The data is collected from the four experiments defined above by subjecting a large number of pairs (A, B) of quantum particles to joint measurements. More specifically, each such pair is measured in one of the four measurement conditions represented by the grid of probabilities depicted in Fig 1.

The maximum possible violation of the CHSH inequalities is 4, i.e., three pairs of variables are maximally correlated (= 1) and the fourth is maximally anti-correlated (= -1). However, if the experiment is modelled by a joint probability distribution across the four variables A_0, A_1, B_0, B_1, the maximum value that can be computed by any of the inequalities happens to be 2. This is why the boundary of violation in the inequalities is 2 as it demarcates the boundary which standard statistical correlations cannot transcend. This fact presents a challenge for a PPL, which is based on standard probability theory. How can a PPL be developed to simulate non-classical quantum correlations?

Design of an EPR Simulation Experiment using PPLs

Fig 2 depicts the framework for how a PPL can be used to simulate EPR experiments. A phenomenon P, e.g., entangled quantum particles, is to be studied. An experimental design is devised in which P is examined in the four experimental conditions called "measurement contexts". A measurement context M_i, $1 \leq i \leq 4$ is designed to study P from a particular experimental perspective. For example, one measurement context corresponds to $X = 0$ and $Y = 1$

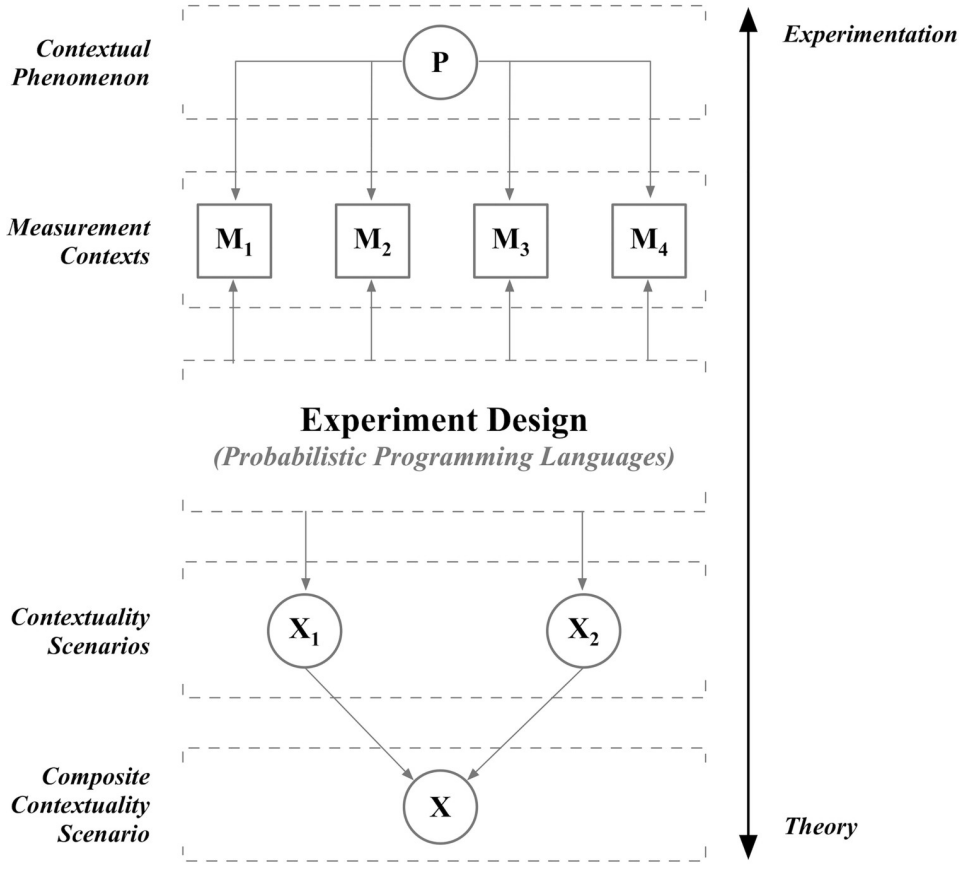

Fig 2. Framework for the EPR experiment.

https://doi.org/10.1371/journal.pone.0208555.g002

which yields probabilities over the four possible outcomes of joint measurements of A and B. We will denote these outcomes as {00|01, 01|01, 10|01, 11|01}. For example, 00|01 denotes the outcome $A = 0, B = 0$ in the measurement context $M_2 = \{X = 0, Y = 1\}$.

Measurement contexts are formally defined as hyperedges in a hypergraph called a "contextuality scenario". Contextuality scenarios $\mathcal{X}_i$, $1 \leq i \leq 2$ are composed into a composite contextuality scenario $\mathcal{X}$, which is a hypergraph describing the phenomenon P. Composition offers the distinct advantage of allowing experimental designs to be theoretically underpinned by hypergraphs in a modular way [7]. More formally, a *contextuality scenario* is a hypergraph $X = (V, E)$ such that:

- $v \in V$ denotes an outcome which can occur in a measurement context
- $e \in E$ is the set of all possible outcomes given a particular measurement context

See Definition 2.2.1 in Ref. [4].

It is important to note that the PPL functions as both a means to simulate an EPR experiment as well as determine whether quantum correlations are present. As we will see below, each hyperedge of $\mathcal{X}$ is a probability distribution over outcomes in a given measurement context. In EPR experiments these distributions are computed by a sampling process which ensures that the no-signaling constraint is adhered to. In order to achieve this, the hypergraphs $\mathcal{X}_i$ are composed using the Foulis—Randall (FR) product [4] (see the next section). As a consequence, the PPL must implement this product for a valid simulation of an EPR experiment. Much of the technical detail to follow describes how this can be achieved. To our knowledge the FR product has never been implemented before in a PPL. Several such implementations will be specified below in various PPLs and then compared.

At the conclusion of the simulation, the CHSH equalities can be applied to correlations computed from relevant hyperedges in the composite contextuality scenario $\mathcal{X}$ to determine whether quantum correlations are present. If so, the PPL has successfully simulated phenomenon P as exhibiting quantum, rather than, classical statistics.

Foulis—Randall product

The FR product is used to compose contextuality scenarios as its product ensures no signalling between systems represented by the variables A and B [4].

The *Foulis—Randall product* is the scenario $H_A \otimes H_B$ with

$$V(H_A \otimes H_B) = V(H_A) \times V(H_B), \quad E(H_A \otimes H_B) = E_{A \to B} \cup E_{B \to A}$$

where

$$E_{A \to B} \coloneqq \left\{ \bigcup_{a \in e_A} \{a\} \times f(a) : e_A \in E_A, f : e_A \to E_B \right\}$$

$$E_{A \leftarrow B} \coloneqq \left\{ \bigcup_{b \in e_B} \{b\} \times f(b) : e_B \in E_B, f : e_B \to E_A \right\}$$

The preceding definition formalizes the simultaneous measurements of the two systems A and B such that no-signalling occurs between these systems [4]. The no-signalling constraint is imposed by means of a set of specific hyperedges which are a consequence of the FR product.

We now turn to the issue of modularity which was mentioned previously. There are two systems A and B. System A has two measurement contexts: 1) $A|X = 0$ and 2) $A|X = 1$, where both measurements yield an outcome $A = 0$ or $A = 1$. In the hypergraph formalism, a measurement context is formalized by a hyperdge. The hypergraph H_A therefore has two hyperedges,

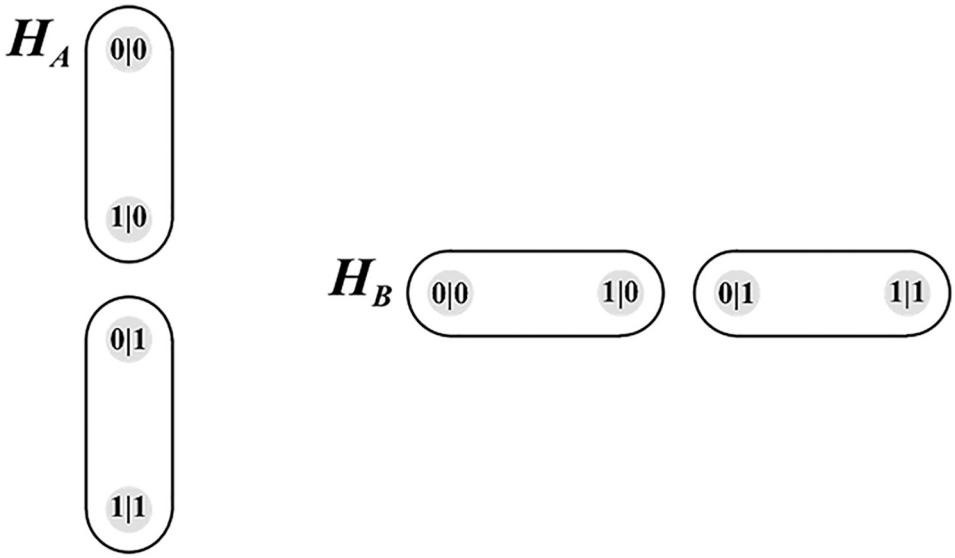

Fig 3. Hypergraph Representation Of EPR Systems A & B.

https://doi.org/10.1371/journal.pone.0208555.g003

one for each measurement contexts. These two hyperedges are visually represented on the LHS of Fig 3. Similarly, hypergraph H_B comprises two edges. H_A and H_B can be viewed as modules which can be composed in various ways to suit the requirements of a particular experimental design. In the EPR experiment, four measurement contexts are required in which A are B are jointly measured subject to the no-signalling condition.

In order to achieve this, the hypergraphs H_A and H_B are composed using the FR product to produce a composite hypergraph. The corresponding hypergraph contains 12 edges. Four of these edges correspond to the four pairwise distributions depicted in Fig 1 and 8 additional edges which ensure that no-signalling can occur.

The FR product produces the hypergraph depicted in Fig 4. This hypergraph corresponds to composite contextuality scenario $\mathcal{X}$ depicted in Fig 2.

To assist with understanding this formalism, one single hyperedge's calculation is considered.

Let e_A be equivalent to edge {0|0, 1|0} of hypergraph H_A.

The relevant calculation associated with the instance may then be one of two combinations:

$$f(0|0) \cup f(1|0) \text{ or } f(1|0) \cup f(0|0)$$

The first of the two combinations is selected, expanding to the following expression:

$$\{00|00, 01|00\} \cup \{10|01, 11|01\}$$

The hyperedge is isolated in Fig 5.

In what follows, we implement this hypergraph formalism in several probabilistic programming languages and evaluate the advantages of each.

Implementations

In this section, four commonly available PPLs illustrate a simulation of the same EPR experiment. The goal of this comparison is to judge their relative effectiveness for this purpose.

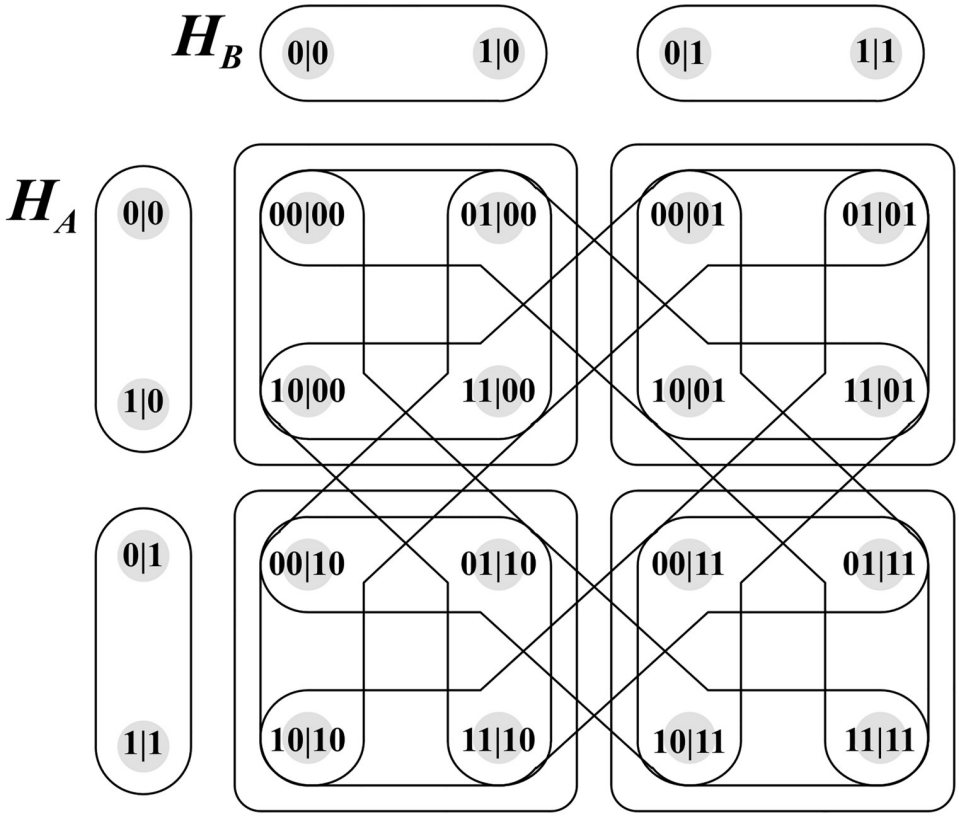

Fig 4. Hyperedges Of Foulis—Randall Product.

https://doi.org/10.1371/journal.pone.0208555.g004

Scope Of investigation

Four PPLs were chosen for both qualitative and quantitaive comparison and are listed in Table 1. While other PPLs such as Stan [8], Church [9], or WebPPL [2] were considered for the investigation, we decided to exclude such domain-specific languages on the basis of limited applications in quantum physics. Probabilistic programming frameworks that only focus on directed graphs, such as Edward [10], were also excluded, since this feature is not relevant to the EPR experiment in the hypergraph formalism.

Qualitative comparison Of PPLs

The qualitative comparison highlights important pragmatic aspects of probabilistic programs, and is defined by the following criteria.

Criteria Of Comparison.

- **Extensibility**: The PPL accommodates for simulation of complex experimental settings. This may be inherent in the PPL's means of extension i.e., is open-source, or whether its syntactic constructs provide flexibility in specifying data structures and the flow of control.
- **Accessibility**: The PPL is intuitive and coherent. Possibly by means of expressive constructs, or comprehensive supporting documentation, accessibility may also be demonstrated by the PPL's community base, or degree of application.

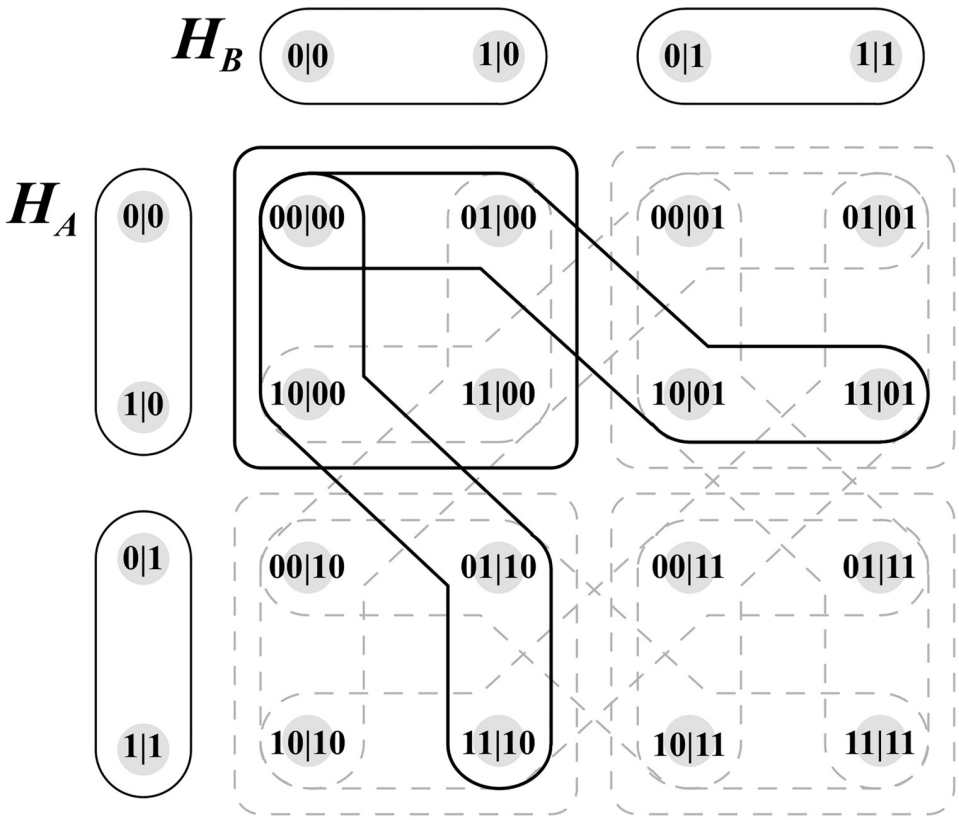

Fig 5. Single Hyperedge Of Foulis—Randall Product.

https://doi.org/10.1371/journal.pone.0208555.g005

- **Acceleration**: The PPL implements methods of optimization for its execution, reflected in the speed and resource-utilization of its compilation. Acceleration may also be demonstrated in the PPL's scalability.

These criteria are derived from criteria commonly used to judge programming languages.

Extensibility Of PPLs. Regarding extensibility, all PPLs are supplemented with repositories containing source code that can be (with respect to licenses) modified and re-compiled. While all PPLs offer containment systems for configuration of probabilistic models, only PyMC3 and Figaro provide tools for the diagnosis and validation of models. In considering Turing.jl's dependence on the Distributions.jl [15] package, it can be said that all PPLs provide a number of distribution configurations. In contrast, not all PPLs offer flexibility in step methods used for sampling. This can be overlooked considering that beyond common sampling methods, excess of configurations are typically specialised. Figaro and Pyro are the only PPLs

Table 1. Basic characteristics of probabilistic programming languages.

Name	Programming language	License	Supported OS
PyMC3 [11]	Python	Apache-2.0	Windows, Mac, Linux
Figaro [12]	Scala	Custom	Windows, Mac, Linux
Turing.jl [13]	Julia	MIT	Windows, Mac, Linux
Pyro [14]	Python	Custom	Windows, Mac, Linux

https://doi.org/10.1371/journal.pone.0208555.t001

Table 2. Comparison Of extensibility Of PPLs.

Criteria	PyMC3	Figaro	Turing.jl	Pyro
Control-Flow Independence	✗	✓	✗	✓
Open Universe Simulation	✗	✓	✗	✓
Distribution Configurations	∼60	∼36	?	∼39
Step Methods	∼8	∼41	∼13	∼4
Algorithm Manipulation	✓	✓	✗	✓
Online Repository	✓	✓	✓	✓
Model Configuration	✓	✓	✓	✓
Model Validation	✓	✓	✗	✗

https://doi.org/10.1371/journal.pone.0208555.t002

to offer control-flow independence in probabistic inference; both Figaro's inference algorithms and Pyro's strong integration with Python allow for atomic inference processing. Both are also the only PPLs to offer comprehensive open universe simulation [16]. All PPLs except Turing.jl provide constructs for the manipulation of the underlying inference algorithms. Table 2 demonstrates the articulated features in comparison.

Accessibility Of PPLs. While PyMC3 and Turing.jl have seen a wealth of research projects conducted since their conception, the later debut of Figaro and Pyro has perpetuated fewer examples of application. In light of this, both PPLs provide tutorial literature, and have more comprehensive API reference documentation than the former two. In contrast, Turing.jl has limited tutorial content to support its usage, and does not provide a complete API reference. For ease of use, Pyro advertises its design for agile development, however its syntactic conventions do not warrant any significant differences compared to PyMC3. Nevertheless, both are more easily applied than Figaro or Turing.jl.

Acceleration Of PPLs. Concerning acceleration, PyMC3 bases its optimization on Theano's architecture, which is an open-sourced project originally produced at the Université de Montréal [17]. Correctly applying the Theano architecture with respect to the GPU on which the PPL is running is a multi-staged process. As Theano depends on the NVIDIA CUDA Developer Toolkit, the GPU's compatibility with the toolkit's contained drivers must be verified before installation can occur. Thereafter, the software 'self-validates', and PyMC3 configuration settings must be altered to recognize the GPU support. Only in the instance that the toolkit is correctly installed can PyMC3 take full advantage of its GPU acceleration capabilities. For contrast, no other PPLs evaluated require manual extension of acceleration. For current experimentation, Theano may be suitable, however its discontinuation as of 2017 [18] poses a threat to using it as a stable basis for future development. For comparison, Figaro was designed specifically for usage within demanding experimental designs. The development team has stressed the library's capability with its various capabilities e.g. open universe models, spatio-temporal models, recursive models, or infinite models [16].

Similarly, Uber AI Labs stresses that Pyro can be easily scaled to projects of demanding size [19]; it should be noted that Pyro is based on Pytorch framework, and as a result takes advantage of Pytorch's strong GPU accelerated machine learning tools [20].

Illustration of the EPR experiment in the four PPLs

In specifying the EPR experiment in different PPLs varying syntactic constructs can be are highlighted and contrasted, as well as the differing approaches to the simulation.

PyMC3. A PyMC3 model defines a context management system used to isolate operations undertaken on stochastic random variables, and thus Python's `with` keyword is applied to

automate the release of resources after implementation. Inside the model, the `Bernoulli` method specifies that a distribution of Bernoulli values will be simulated for a given random variable. A probability is also given to direct the sampler towards a bias when generating the distribution. To assist with randomizing results, a `Uniform` distribution is also declared. Then the `sample` method invokes a number of iterations over the specified model. PYMC3 designates tuning of results prior to sampling, as well as indication of a sampling method for which a number of algorithms are offered. In the example, `Metropolis` implies the Metropolis-Hastings algorithm will be used to obtain random results. Upon execution, the model generates a trace containing distributions reflecting the earlier declared random variables.

As this is the first example of code for the experimentation, annotations expressing the meaning of the code are included throughout the implementation.

```
from numpy import zeros, array, fliplr, sum
from itertools import product
import pymc3 as pm
```

The first block of the implementation declares assistant methods used for value conversions. The first, being the `get_vertex` method linearises the binary variables X, Y, A, and B used to express probabilities of the global distribution into an index of an array.

$$get\_vertex(x,\ y,\ a,\ b) \coloneqq ((x \times 8) + (y \times 4)) + (b + (a \times 2))$$

```
def get_vertex(a, b, x, y):
  return ((x*8)+(y*4))+(b+(a*2))
```

The second method, `get_hyperedges` leverages enumeration techniques to retrieve hyperedges for contained vertices.

```
def get_hyperedges(H, n):
  l = []
  for idx, e in enumerate(H):
    if n in e:
      l.append(idx)
  return l
```

The `foulis_randall_product` method generates the binary coordinates for all hyperedges in the FR product.

```
def foulis_randall_product():
  fr_edges = []
```

The first step involves declaring the hypergraphs for both EPR systems.

$$((((0,\ 0),\ (1,\ 0)),\ ((0,\ 1),\ (1,\ 1))),\ (((0,\ 0),\ (1,\ 0)),\ ((0,\ 1),\ (1,\ 1))))$$

```
H = [[[[0,  0], [1,  0]], [[0,  1], [1,  1]]],
     [[[0,  0], [1,  0]], [[0,  1], [1,  1]]]]
```

The next step involves producing four hyperedges to represent the four explicit joint measurement contexts on both systems. Two variables are given to assist with computing this result.

$$g \in E_A,\ \ h \in E_B$$

```
for edge_a in H[0]:
  for edge_b in H[1]:
```

Thereafter, each hyperedge is defined as the combined sets produced by the following expression.

$$(\forall i \in g,\ \ \forall j \in h : \forall w \in i,\ \ \forall x \in j : (w_1,\ x_1,\ w_2,\ x_2))$$

```
fr_edge = []
for vertex_a in edge_a:
  for vertex_b in edge_b:
    fr_edge.append([
      vertex_a[0], vertex_b[0],
      vertex_a[1], vertex_b[1]])
fr_edges.append(fr_edge)
```

The last step involves calculating the hyperedges of both systems as dependent on the other. To achieve this programmatically, three variables are declared. The first (m) are all members of set M, where M are the possible measurement choices for the scenario (in this case two), the

second (n) being the other possible measurement choice, and the last variable o being all edges from the hypergraph associated with the measurement choice.

$$\forall m \in M : n = m', \ \forall o \in E(H_m)$$

```
for mc in range (0,2):
  mc_i = abs(1-mc)
  for edge in H[mc]:
```

For each o, in some selected hypergraph, a second variable j is declared as two possible values. For each possibility, a hyperedge is then defined as the variable k, being the combination of all vertices in o given to a function that produces the hyperedge.

$$\forall j \in (1, \ 2) : k = (\forall l \in E(H_n) : f(l, \ j, \ m, \ n, \ o))$$

```
for j in range (0,2):
  fr_edge = []
  for i in range (0, len (edge)):
```

The mentioned function computes the hyperedge by declaring single-use variables s, q, r, u, t, and v.

$$E(H_n)_s = l', \ q = o_{|s-j|+1}, \ r = l_1, \ u = l_2, \ t = (q_1, \ r_1, \ q_2, \ r_2), \ v = (q_1, \ u_1, \ q_2, \ u_2)$$

```
edge_b = H[mc_i][i]
vertex_a = edge[abs(i-j)]
vertex_b = edge_b[0]
vertex_c = edge_b[1]
vertices_a = [
  vertex_a[0], vertex_b[0],
  vertex_a[1], vertex_b[1]]
vertices_b = [
  vertex_a[0], vertex_c[0],
  vertex_a[1], vertex_c[1]]
```

Thereafter, a set is constructed by use of its variables, and portions of the desired hyperedges are iteratively returned.

$$((t_m,\ t_n,\ t_{m+2},\ t_{n+2}),\ (v_m,\ v_n,\ v_{m+2},\ v_{n+2}))$$

Upon calculation of the last step of the process, the hyperedges corresponding to the measurement choices of both EPR systems as dependent on the other are produced, totaling the necessary constraints described in binary format.

```
    fr_edge.append([
      vertices_a[mc], vertices_a[mc_i],
      vertices_a[mc+2], vertices_a[mc_i+2]]
      )
    fr_edge.append([
      vertices_b[mc], vertices_b[mc_i],
      vertices_b[mc+2], vertices_b[mc_i+2]])
  fr_edges.append(fr_edge)
return fr_edges
```

To compute the four pairwise distributions at the basis of the EPR experiment (See Fig 1, an iterative sampling process is undertaken for the four variables *a*, *b*, *x*, and *y* that were previously mentioned as specifying one of the 16 possible vertices. These values are restricted to binary outcomes by means of specifying Bernoulli distributions for which the sampler runs the process. The experiment is fixed such that each vertex has an equal possibility of being sampled as any other vertex, and results may only be discounted if they do not comply with specified input correlations. Upon selecting a vertex at a step in the iterative process, the array index associated with the binary representation is incremented by one, via the use of the vertex mapping function. Simultaneously, another array representing the hyperedges of the FR product is also incremented by one at all indexes associated with the hyperedges containing the said vertex. The iterative process only exits when the sum of the global distribution is equivalent to the desired number of iterations. Thereafter, each vertex is normalised by the sum of the values in the corresponding array of hyperedges in which its associated vertex is contained, and is multiplied by 3 (for reflection of the number of associated hyperedges). A visualisation of the hyperedges associated with the vertex at index 00|00 of the global distribution can be seen in Fig 6.

If the said vertex sustains a weight of 10, and the combined weight of its associated hyperedges is 40, the normalised weight of the vertex will equate to 0.75.

```
def generate_global_distribution(constraints,N):
  hyperedges = foulis_randall_product()
```

```
  hyperedges_tallies = zeros(12)
  global_distribution = zeros(16)
  while sum (global_distribution) < N:
    with pm.Model():
      pm.Uniform('C',0.0,1.0)
      pm.Bernoulli('A',0.5)
      pm.Bernoulli('B',0.5)
      pm.Bernoulli('X',0.5)
      pm.Bernoulli('Y',0.5)
      S = pm.sample(N,tune = 0, step = pm.Metropolis())
      c = S.get_values('C')
      a = S.get_values('A')
      b = S.get_values('B')
      x = S.get_values('X')
      y = S.get_values('Y')
    for i in range(0, N):
      if (c[i] < constraints[x[i]][y[i]][a[i],b[i]]):
        for edge in get_hyperedges(hyperedges,
          [a[i], b[i], x[i], y[i]]):
          hyperedges_tallies[edge] += 1
        global_distribution[
          get_vertex(a[i], b[i], x[i], y[i])] += 1
  z = [0, 1]
  for a, b, x, y in product(z,z,z,z):
    summed_tally = (sum(hyperedges_tallies[e]
      for e in get_hyperedges(hyperedges, [a, b, x, y])))
    global_distribution[get_vertex(a, b, x, y)] /=
summed_tally
  global_distribution *= 3
  return global_distribution
```

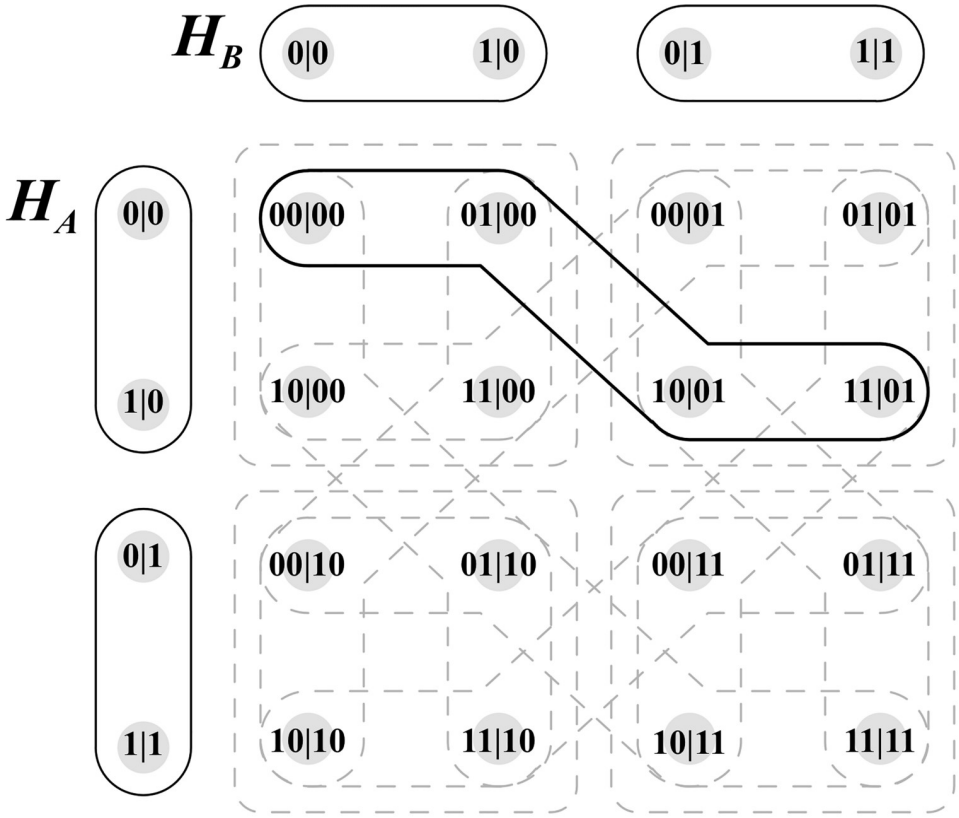

Fig 6. Visualisation Of hyperedges associated with vertex 00|00.

https://doi.org/10.1371/journal.pone.0208555.g006

Given below in Listing 1 is the complete undivided implementation of the EPR experimentation in PyMC3.

Listing 1. PyMC3 Implementation Of EPR Simulation

```
from numpy import zeros, array, fliplr, sum
from itertools import product
import pymc3 as pm
def get_vertex(a, b, x, y):
  return ((x*8)+(y*4))+(b+(a*2))
def get_hyperedges(H, n):
  l = []
  for idx, e in enumerate(H):
    if n in e:
      l.append(idx)
  return l
def foulis_randall_product():
  fr_edges = []
```

```
H = [[[[0,  0], [1,  0]], [[0,  1], [1,  1]]],
     [[[0,  0], [1,  0]], [[0,  1], [1,  1]]]]
for edge_a in H[0]:
  for edge_b in H[1]:
    fr_edge = []
    for vertex_a in edge_a:
      for vertex_b in edge_b:
        fr_edge.append([
          vertex_a[0], vertex_b[0],
          vertex_a[1], vertex_b[1]])
    fr_edges.append(fr_edge)
for mc in range (0,2):
  mc_i = abs(1-mc)
  for edge in H[mc]:
    for j in range (0,2):
      fr_edge = []
      for i in range (0, len (edge)):
        edge_b = H[mc_i][i]
        vertex_a = edge[abs (i-j)]
        vertex_b = edge_b[0]
        vertex_c = edge_b[1]
        vertices_a = [
          vertex_a[0], vertex_b[0],
          vertex_a[1], vertex_b[1]]
        vertices_b = [
          vertex_a[0], vertex_c[0],
          vertex_a[1], vertex_c[1]]
        fr_edge.append([
          vertices_a[mc], vertices_a[mc_i],
          vertices_a[mc+2], vertices_a[mc_i+2]]
          )
        fr_edge.append([
          vertices_b[mc], vertices_b[mc_i],
          vertices_b[mc+2], vertices_b[mc_i+2]])
```

```
        fr_edges.append(fr_edge)
  return fr_edges
def generate_global_distribution(constraints,N):
  hyperedges = foulis_randall_product()
  hyperedges_tallies = zeros(12)
  global_distribution = zeros(16)
  while sum (global_distribution) < N:
    with pm.Model():
      pm.Uniform('C',0.0,1.0)
      pm.Bernoulli('A',0.5)
      pm.Bernoulli('B',0.5)
      pm.Bernoulli('X',0.5)
      pm.Bernoulli('Y',0.5)
      S = pm.sample(N,tune = 0, step = pm.Metropolis())
      c = S.get_values('C')
      a = S.get_values('A')
      b = S.get_values('B')
      x = S.get_values('X')
      y = S.get_values('Y')
    for i in range (0, N):
      if (c[i] < constraints[x[i]][y[i]][a[i],b[i]]):
        for edge in get_hyperedges(hyperedges,
          [a[i], b[i], x[i], y[i]]):
          hyperedges_tallies[edge] += 1
        global_distribution[
          get_vertex(a[i], b[i], x[i], y[i])] += 1
  z = [0, 1]
  for a, b, x, y in product(z,z,z,z):
    summed_tally = (sum (hyperedges_tallies[e]
      for e in get_hyperedges(hyperedges, [a, b, x, y])))
    global_distribution[get_vertex(a, b, x, y)] /=
summed_tally
  global_distribution *= 3
  return global_distribution
```

Turing.jl. Like PyMC3, Turing.jl isolates operations on random variables to a single model with the use of the `@model` macro. To obtain randomly sampled non-negative values for a Bernoulli distribution, the model requires the declaration of a uniform Beta prior, invoked with the `Beta` method. Then Bernoulli distributions are declared with a `Bernoulli` method, once again accompanied by probabilities describing sampling biases for later generated distributions, as well as a `Uniform` distribution.

In the `generate_global_distribution` method of Listing 2, the `sample` function invokes the model, a step-method, as well as the number of desired iterations. In this case, the Sequential Monte Carlo sampling (abbreviated to `SMC`) has been applied.

To obtain the trace of a distribution in the model, the output must be indexed with the precession of a colon. In the example, the results are retrieved, tallied, and normalised by means of the `foulis_randall_product` method, before returning the result.

Listing 2. Turing.jl Implementation Of EPR Simulation

```
using Turing
using Distributions
function foulis_randall_product()
  fr_edges = Array{Array{Array{Float64}}}(0)
  H = [[[[0.0, 0.0],[1.0, 0.0]],[[0.0, 1.0],[1.0, 1.0]]],
       [[[0.0, 0.0],[1.0, 0.0]],[[0.0, 1.0],[1.0, 1.0]]]]
  for i = 1:size(H[1])[1]
    for j = 1:size(H[2])[1]
      fr_edge = Array{Array{Float64}}(0)
      for k = 1:size(H[1][i])[1]
        for l = 1:size(H[1][j])[1]
          append!(fr_edge,
          [[H[1][i][k][1], H[2][j][l][1],
            H[1][i][k][2], H[2][j][l][2]]])
        end
      end
      append!(fr_edges,[fr_edge])
    end
  end
  for mc = 1:2
    mc_i = abs(3-mc)
    for k = 1:size(H[mc])[1]
      for j = 1:2
```

```
        fr_edge = Array{Array{Float64}}(0)
        for i = 1:size(H[mc][k])[1]
          edge_b = H[mc_i][i]
          vertex_a = H[mc][k][abs(i-j)+1]
          vertex_b = edge_b[1]
          vertex_c = edge_b[2]
          vertices_a = [vertex_a[1], vertex_b[1],
                vertex_a[2], vertex_b[2]]
          vertices_b = [vertex_a[1], vertex_c[1],
                vertex_a[2], vertex_c[2]]
          this_edge_b = Array{Float64}(0)
          append!(fr_edge,[[
                vertices_a[mc], vertices_a[mc_i],
                vertices_a[mc+2], vertices_a[mc_i+2]]])
          append!(fr_edge,[[
                vertices_b[mc], vertices_b[mc_i],
                vertices_b[mc+2], vertices_b[mc_i+2]]])
        end
        append!(fr_edges,[fr_edge])
      end
    end
  end
  fr_edges
end
function get_vertex(a,b,x,y)
  ((x×8)+(y×4))+(b+(a×2))+1
end
function get_hyperedges(H, n)
  l = []
  for i = 1:size(H)[1]
    if any(x→x ==n, H[i])
      append!(l,i)
    end
```

```
    end
    l
end
@model mdl() = begin
    z ≈ Beta(1,1)
    a ≈ Bernoulli(0.5)
    b ≈ Bernoulli(0.5)
    x ≈ Bernoulli(0.5)
    y ≈ Bernoulli(0.5)
    c ≈ Uniform(0.0, 1.0)
end
function generate_global_distribution(constraints,N)
    hyperedges = foulis_randall_product( )
    hyperedges_tallies = zeros(12)
    global_distribution = zeros(16)
    while sum(global_distribution) < N
        r = sample(mdl(), SMC(N))
        a = r[:a]
        b = r[:b]
        x = r[:x]
        y = r[:y]
        c = r[:c]
        for i = 1:N
            if (c[i] < constraints[x[i]+1][y[i]+1][a[i]+1][b[i]+
1])
                I = [convert(Float64,a[i]), convert(Float64,b[i]),
                        convert(Float64,x[i]), convert(Float64,y[i])]
                associated_hyperedges = get_hyperedges(hyperedges, I)
                for j = 1:size(associated_hyperedges)[1]
                    hyperedges_tallies[
                        associated_hyperedges[j]] += 1
                end
                global_distribution[
```

```
            get_vertex(a[i], b[i], x[i], y[i])] += 1
        end
      end
    end
    for a = 0:1, b = 0:1, x = 0:1, y = 0:1
      summed_amount = 0
      I = [convert(Float64,a), convert(Float64,b),
              convert(Float64,x), convert(Float64,y)]
      associated_hyperedges = get_hyperedges(hyperedges, I)
      for edge_index = 1:size(associated_hyperedges)[1]
        summed_amount += hyperedges_tallies[edge_index]
      end
      global_distribution[get_vertex(a, b, x, y)] /= summed_amount
    end
    global_distribution .× 3
end
```

Figaro. To achieve a joint-probability distribution on a measurement context of random variables, Figaro's syntactic elements reveal fundamental differences in its approach. A class is the advised object for the purpose of declaring a model. For each random variable in the model, a probability bias is applied to Figaro's `Flip` method, generating a Bernoulli distribution on which the `If` method can then associate the results to desired values, or perpetuation of other methods.

In Listing 3, states are bound to integers. Thereafter, possible joint outcomes of random variables are permuted through articulation of expressions concerning the previously mentioned states. In the `GenerateGlobalDistribution` method, it can be seen that after initialising the FR product (via the `FoulisRandallProduct` method), the sampling process is called by means of `start`, `stop`, and `kill` chains applied on the `algorithm` object. On the preceding line, the `MetropolisHastings` method implies the Metropolis-Hastings step-method will be used for the sampling process, and the `outcomes` of the `model` class will be considered. For more complex experiments, the `ProposalScheme` may be modified, however not in this case. The `sampleFromPosterior` sub-method chained to calls on each variable compile the required distributions on execution. The `take` sub-method chained to the sampling methods are used to declare the number of outcomes retrieved from the sampler. This aspect is consequent of sampler delivering results via `Stream` primitives, a resource-efficient consideration ensuring that only required data is evaluated. Thereafter, the proceeding code tallies the indexes of the `globalDistribution` array, and normalises the results.

Listing 3. Figaro Implementation Of EPR Simulation

```
import com.cra.figaro.algorithm.sampling._
import com.cra.figaro.language._
import com.cra.figaro.library.compound.If
import com.cra.figaro.library.atomic.continuous.Uniform
object Main {
  def FoulisRandallProduct(): Array[Array[Array[Double]]] = {
    var foulisRandallEdges = Array[Array[Array[Double]]] ()
    val hypergraphs = Array(
      Array(Array(Array(0.0, 0.0), Array(1.0, 0.0)),
            Array(Array(0.0, 1.0), Array(1.0, 1.0))),
      Array(Array(Array(0.0, 0.0), Array(1.0, 0.0)),
            Array(Array(0.0, 1.0), Array(1.0, 1.0))))
    for (edgeA <- hypergraphs(0)) {
      for (edgeB <- hypergraphs(1)) {
        var foulisRandallEdge = Array[Array[Double]]()
        for (vertexA <- edgeA) {
          for (vertexB <- edgeB) {
            foulisRandallEdge ++= Array(Array[Double](
              vertexA(0), vertexB(0), vertexA(1), vertexB(1)))
          }
        }
        foulisRandallEdges ++= Array(foulisRandallEdge)
      }
    }
    for (measurementChoice <- 0 to 1){
      val measurementChoiceInverse = 1 - measurementChoice
      for (edge <- hypergraphs(measurementChoice)) {
        for (j <- 0 to 1) {
          var foulisRandallEdge = Array[Array[Double]]()
          for (i <- edge.indices) {
              val edgeB = hypergraphs(measurementChoiceIn-
              verse)(i)
              val vertexA = edge(Math.abs(i-j))
```

```
            val vertexB = edgeB(0)
            val vertexC = edgeB(1)
            val verticesA = Array(vertexA(0), vertexB(0),
                                  vertexA(1), vertexB(1))
            val verticesB = Array(vertexA(0), vertexC(0),
                                  vertexA(1), vertexC(1))
            foulisRandallEdge ++= Array(
              Array[Double](
                verticesA(measurementChoice),
                verticesA(measurementChoiceInverse),
                verticesA(measurementChoice+2),
                verticesA(measurementChoiceInverse+2)))
            foulisRandallEdge ++= Array(
              Array[Double](
                verticesB(measurementChoice),
                verticesB(measurementChoiceInverse),
                verticesB(measurementChoice+2),
                verticesB(measurementChoiceInverse+2)))
          }
          foulisRandallEdges ++= Array(foulisRandallEdge)
        }
      }
    }
    foulisRandallEdges
}
class Model() {
  var outcomes = Array[Element[Double]]()
  for (i <- 0 to 3){
    outcomes: += If(Flip(0.5), 0.0, 1.0)
  }
  outcomes: += Uniform(0.0, 1.0)
}
def GetVertex(a: Int, b: Int, x: Int, y: Int): Int = {
```

```
    ((x*8)+(y*4))+(b+(a*2))
  }
  def GetHyperedges(H: Array[Array[Array[Double]]],
  n: Array[Double]): Array[Int] = {
    var l = Array[Int]()
    for (i <- H.indices) {
      if (H(i).deep.contains(n.deep)) {
        l: += i
      }
    }
    l
  }
  def GenerateGlobalDistribution(constraints:
    Array[Array[Array[Array[Double]]]], N:Int): Unit = {
    val hyperedges = FoulisRandallProduct()
    var hyperedgesTallies = Array[Double].fill(12){0.0}
    var globalDistribution = Array[Double].fill(16){0.0}
    while (globalDistribution.sum < N) {
      var model = new Model()
      val algorithm = MetropolisHastings(N,
      ProposalScheme.default, model.outcomes: _*)
      algorithm.start()
      algorithm.stop()
      algorithm.kill()
      val a = algorithm.sampleFromPosterior(
        model.outcomes(0)).take(N).toArray
      val b = algorithm.sampleFromPosterior(
        model.outcomes(1)).take(N).toArray
      val x = algorithm.sampleFromPosterior(
        model.outcomes(2)).take(N).toArray
      val y = algorithm.sampleFromPosterior(
        model.outcomes(3)).take(N).toArray
      val c = algorithm.sampleFromPosterior(
```

```
        model.outcomes(4)).take(N).toArray
      for (i <- 0 until N) {
      val x_x = x(i).toInt
      val y_y = y(i).toInt
      val a_a = a(i).toInt
      val b_b = b(i).toInt
      if (c(i) < constraints(x_x)(y_y)(a_a)(b_b)) {
        for (edge <- GetHyperedges(
          hyperedges, Array(a_a, b_b, x_x, y_y))) {
            hyperedgesTallies(edge) += 1.0
        }
        globalDistribution(GetVertex(a_a, b_b, x_x, y_y)) +=
        1.0
      }
    }
  }
  for (a <- 0 to 1;b <- 0 to 1; x <- 0 to 1; y <- 0 to 1) {
    var summedAmount = 0.0
    val associatedHyperedges = GetHyperedges(hyperedges,
      Array(a.toDouble,b.toDouble,x.toDouble,y.toDouble))
    for (edgeIndex <- associatedHyperedges.indices) {
      summedAmount += hyperedgesTallies(edgeIndex)
    }
    globalDistribution(GetVertex(a,b,x,y)) =
      globalDistribution(GetVertex(a,b,x,y)) / summedAmount
    }
    globalDistribution
  }
}
```

Pyro. Pyro's context management is integrated into Python's `def` containers; or can be flexibly given implicitly, encouraging the use of stochastic functions to specify probabilistic models. Inside a container, probabilities of random variables are specified first. As Pyro is built upon PyTorch, explicit values match PyTorch types, in this case resulting in `Tensor` type values.

As can be seen in Listing 4 in the `generate_global_distribution` method, Pyro's atomic sampling capabilities allow for the requirement of fewer syntactic constructs to communicate the sampling process. Each `sample` call accepts a distribution, in this case either `Bernoulli` or `Uniform`. Upon sampling, the outcomes are tallied and normalised, before presenting the result.

Listing 4. Pyro Implementation Of EPR Simulation

```
from pyro import sample
from torch import Tensor
from torch.autograd import Variable
from numpy import zeros, array, fliplr, sum
from itertools import product
from pyro.distributions import Bernoulli, Uniform
def get_vertex(a, b, x, y):
  return ((x*8)+(y*4))+(b+(a*2))
def get_hyperedges(H, n):
  l = []
  for idx, e in enumerate(H):
    if n in e:
      l.append(idx)
  return l
def foulis_randall_product():
  fr_edges = []
  H = [[[[0,  0], [1,  0]], [[0,  1], [1,  1]]],
       [[[0,  0], [1,  0]], [[0,  1], [1,  1]]]]
  for edge_a in H[0]:
    for edge_b in H[1]:
      fr_edge = []
      for vertex_a in edge_a:
        for vertex_b in edge_b:
          fr_edge.append([
            vertex_a[0], vertex_b[0],
            vertex_a[1], vertex_b[1]])
      fr_edges.append(fr_edge)
  for mc in range(0,2):
```

```
    mc_i = abs(1-mc)
    for edge in H[mc]:
      for j in range(0,2):
        fr_edge = []
        for i in range(0, len(edge)):
          edge_b = H[mc_i][i]
          vertex_a = edge[abs(i-j)]
          vertex_b = edge_b[0]
          vertex_c = edge_b[1]
          vertices_a = [
            vertex_a[0], vertex_b[0],
            vertex_a[1], vertex_b[1]]
          vertices_b = [
            vertex_a[0], vertex_c[0],
            vertex_a[1], vertex_c[1]]
          fr_edge.append([
            vertices_a[mc], vertices_a[mc_i],
            vertices_a[mc+2], vertices_a[mc_i+2]]
            )
          fr_edge.append([
            vertices_b[mc], vertices_b[mc_i],
            vertices_b[mc+2], vertices_b[mc_i+2]])
        fr_edges.append(fr_edge)
  return fr_edges
def generate_global_distribution(constraints,N):
  hyperedges = foulis_randall_product()
  hyperedges_tallies = zeros(12)
  global_distribution = zeros(16)
  while sum(global_distribution) < N:
    a = int(sample('A', Bernoulli(Variable(Tensor([0.5])))))
    b = int(sample('B', Bernoulli(Variable(Tensor([0.5])))))
    x = int(sample('X', Bernoulli(Variable(Tensor([0.5])))))
    y = int(sample('Y', Bernoulli(Variable(Tensor([0.5])))))
```

```
    value = float(sample('C', Uniform(Variable(Tensor([0.0])),
      Variable(Tensor([1.0])))))
    if (value < constraints[x][y][a,b]):
      for edge in get_hyperedges(hyperedges, [a, b, x, y]):
        hyperedges_tallies[edge] += 1
      global_distribution[get_vertex(a, b, x, y)] += 1
  z = [0, 1]
  for a, b, x, y in product(z,z,z,z):
    summed_tally = (sum(hyperedges_tallies[e]
      for e in get_hyperedges(hyperedges, [a, b, x, y])))
    global_distribution[get_vertex(a, b, x, y)] /=
summed_tally
  global_distribution *= 3
  return global_distribution
```

Input correlations for sampling

In order to provide flexibility in investigating simulations of quantum and super-quantum correlations, correlations between A and B in the four measurement contexts are specified. For example, the following code fragments 5, 6 and 7 specify that super-quantum correlations will be simulated by specifying A and B to be maximally correlated in three measurement contexts and maximally anti-correlated in the fourth. With these input correlations, maximum violation of the CHSH inequalities would be expected, essentially simulating a PR box [21].

Listing 5. Implementation of input correlations in PYMC3 And Pyro

```
constraints = [[array([[0.5, 0], [0., 0.5]]),
                array([[0.5, 0], [0., 0.5]])],
               [array([[0.5, 0], [0., 0.5]]),
                array([[0, 0.5], [0.5, 0.]])]]
p = generate_global_distribution((constraints, 5000)
```

Listing 6. Implementation of input correlations in Turing.jl

```
constraints = [[[[0.5, 0.0], [0.0, 0.5]],
                [[0.5, 0.0], [0.0, 0.5]]],
               [[[0.5, 0.0], [0.0, 0.5]],
                [[0.0, 0.5], [0.5, 0.0]]]]
p = generate_global_distribution(constraints, 5000)
```

Listing 7. Implementation of input correlations in Figaro

```
val constraints = Array(
  Array(Array(Array(0.5, 0.0), Array(0.0, 0.5)),
        Array(Array(0.5, 0.0), Array(0.0, 0.5))),
  Array(Array(Array(0.5, 0.0), Array(0.0, 0.5)),
        Array(Array(0.0, 0.5), Array(0.5, 0.0))))
val P = GenerateGlobalDistribution(constraints,5000)
```

Specifying the CHSH inequalities

For each of the four PPLs, code is specified 8, 9, and 10 that implements the system of four CHSH inequalities. The outcome is a vector of four Boolean values expressing whether the respective inequality was violated.

Listing 8. Specification of the CHSH inequalities in PYMC3 And Pyro

```
def equality(v1, v2, v3, v4):
  def f1(v1, v2):
    return abs((2 * (p[v1] + p[v2])) - 1)
  def f2(v1, v2, v3, v4):
    return (p[v1] + p[v2]) - (p[v3] + p[v4])
  delta = 0.5 * (
    (f1(0,1) - f1(4,5)) + (f1(8,9) - f1(12,13)) +
    (f1(0,2) - f1(4,6)) + (f1(8,10) - f1(12,14)))
  return 2 * (1 + delta) >= abs(
```

```
    (v1*f2(0,3,1,2)) + (v2*f2(4,7,5,6)) +
    (v3*f2(8,11,9,10)) + (v4*f2(12,15,13,14)))
tests = [
  equality(1,1,1,-1),
  equality(1,1,-1,1),
  equality(1,-1,1,1),
  equality(-1,1,1,1)]
```

Listing 9. Specification of the CHSH inequalities in Turing.jl

```
function equality (v1,v2,v3,v4)
  function f1(v1,v2)
    abs((2 * (p[v1] + p[v2])) - 1)
  end
  function f2(v1,v2,v3,v4)
    (p[v1]+p[v2]) - (p[v3]+p[v4])
  end
  delta = 0.5 * (
    (f1 (1,2) - f1(5,6)) + (f1(9,10) - f1(13,14)) +
    (f1(1,3) - f1(5,7)) + (f1(9,11) - f1(13,15)))
  (2 * (1 + delta)) >= abs(
    (v1 * f2(1,4,2,3)) + (v2 * f2(5,8,6,7)) +
    (v3 * f2(9,12,10,11)) + (v4 * f2(13,16,14,15)))
end
tests = [
  equality(1,1,1,-1),
  equality(1,1,-1,1),
  equality(1,-1,1,1),
  equality(-1,1,1,1)]
```

Listing 10. Specification of the CHSH inequalities in Figaro

```
def Equality(v1: Int, v2: Int, v3: Int, v4: Int): Boolean = {
  def f1(v1: Int, v2: Int): Double = {
    Math.abs((2 * (p(v1) + p(v2))) - 1)
  }
  def f2(v1: Int, v2: Int, v3: Int, v4: Int): Double = {
    (p(v1) + p(v2)) - (p(v3) + p(v4))
  }
  val delta = 0.5 * (
    (f1(0,1) - f1(4,5)) + (f1(8,9) - f1(12,13)) +
    (f1(0,2) - f1(4,6)) + (f1(8,10) - f1(12,14)))
  (2 * (1 + delta)) >= Math.abs(
    (v1*f2(0,3,1,2)) + (v2*f2(4,7,5,6)) +
    (v3*f2(8,11,9,10)) + (v4*f2(12,15,13,14)))
}
val tests = Array[Boolean](
  Equality(1,1,1,-1),
  Equality(1,1,-1,1),
  Equality(1,-1,1,1),
  Equality(-1,1,1,1)
)
```

Upon conducting simulations using the input correlations given above, the predicted maximum violation of the CHSH inequalities were observed for all four PPLs specified above.

Numerical Evaluation

We conducted several experiments to compare the numerical accuracy and execution time of the different implementations.

Accuracy Of tests

For all PPLs, statistical outputs confirming the success of the FR product (and consequently the no-signalling condition) are given with an acceptable margin of error. While this is consequent of more than a single factor, it is perceived that the largest contributor to accuracy is the computation of random values for each PPL. What can be observed is that, with larger sample sizes (bearing more perfectly random distributions), that the margin of error decreases, as can be seen below. This is typical, as the experiment design's normalisation process is dependent

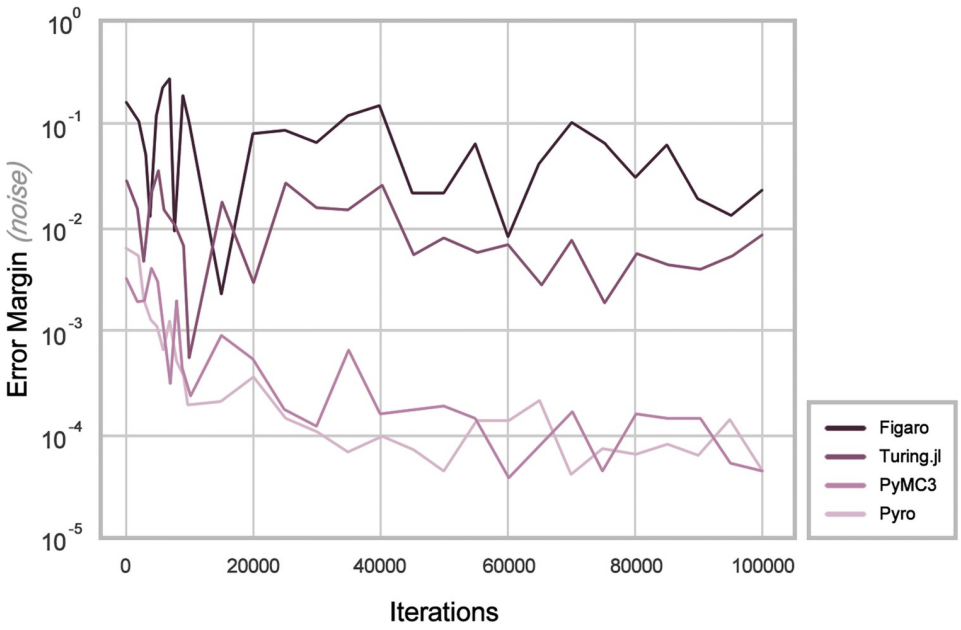

Fig 7. Accuracy Of EPR experimentation.

https://doi.org/10.1371/journal.pone.0208555.g007

on the even spread of tallies across the global distribution, and more specifically, the degree to which the sampler is random. It should also be considered that the various PPLs apply data types that round values for a loss of statistical precision where it may serve meaning. For example, while a single value may lose a minute portion of its whole beyond the decimal point (due to automatic rounding), when calculating a handful of these values per iteration of some few thousand iterations, the difference becomes observable.

From observing the results in Fig 7, it can be seen immediately that before the first 20,000 iterations, all PPLs exhibit substantial noise that rules out the possibility of accounting for said window of results. In Monte Carlo inference, noise of this kind is common, where accounting for 'burn-in' iterations at the beginning of the sampling process may possibly minimize the unpredictability of the results. It is also seen that Figaro and Turing.jl have consistently greater margins of error than those of PyMC3 and Pyro. Overcoming this error would be achieved through improved float precision where applicable in either PPL's programming language. It cannot be said which of either PyMC3 or Pyro display the most accurate results. Of interest, it is perceived that where other PPLs may have required multiple instances of the entire sampling process to tally the global distribution, Pyro could accurately equate the global distribution atomically, thus improving its accuracy.

Elapsed time of execution

Another statistic that has been observed is the compilation time of each PPL, which typically increases with number of iterations. For relativity of results, it should be noted that all non-accelerated tests of this kind were executed within a Bash execution terminal, on a Macintosh operating system, bearing a 45nm "Penryn" 2.4 GHz Intel "Core 2 Duo" processor, and 4 GB of SDRAM, whereas all accelerated tests were executed within a bash execution terminal of Amazon Web Services Linux (2nd distribution). The specification of the 'Elastic-Compute Cloud' on which the Linux distribution executed was a 2018 "p2.xlarge" 2.7 GHz Intel

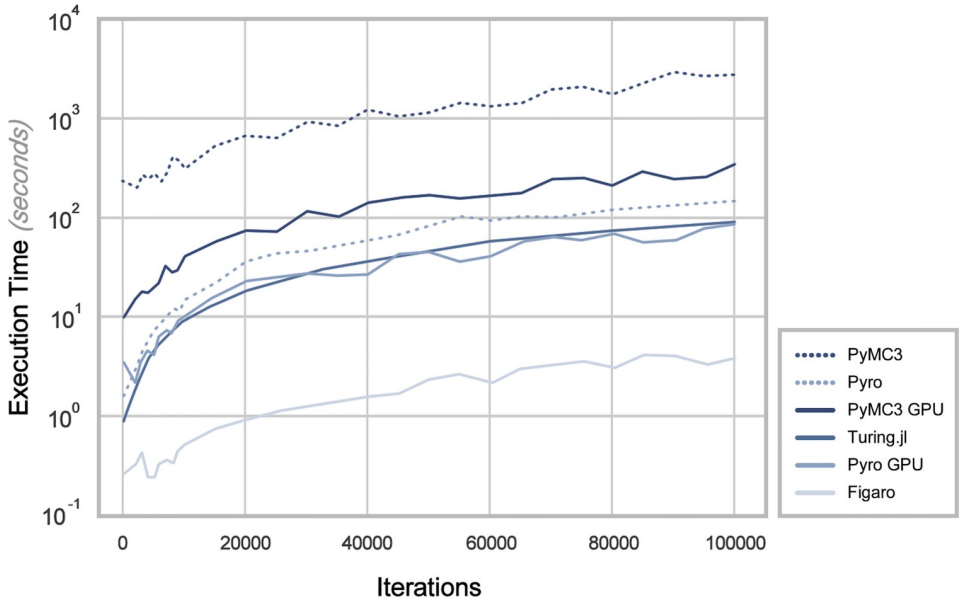

Fig 8. Execution time of EPR experimentation.

https://doi.org/10.1371/journal.pone.0208555.g008

Broadwell processor, with 61 GB of SDRAM. The instance also provides an NVIDIA GK210 GPU multi-vCPU (count of 4) processor, and 12 GB of GPU RAM.

In Fig 8, it can be seen that among the accelerated results, the fastest PPL is Figaro by scale of almost an entire logarithmic unit. Thereafter, Pyro and Turing.jl tie in second position, however Pyro demonstrates inference of a stabler nature. Despite the Theano architecture's utilisation of the supplied GPU, PyMC3 is then the most affected by size of experimental setup. When the Theano architecture is non-accelerated, it can be seen that PyMC3's performance drastically decreases. For comparison, Pyro has also been tested on a non-accelerated architecture, where the difference in performance is reasonably smaller than that of PyMC3. This forms the suggestion that PyMC3 should not be applied in non-accelerated environments. In all cases, it can be seen that all PPLs exhibit a linear order of growth in the given scenario.

Discussion

Recall that the challenge posed was how to develop probabilistic program which can simulate quantum correlations in an EPR experiment. The solution adopted was to program a hypergraph formalism to underpin the simulations. This formalism is modular where the FR product of the modules is used to impose the no-signalling constraint. In execution, all four PPLs successfully simulated an EPR experiment producing quantum correlations. Therefore, we conclude that the hypergraph formalism has been shown to be a promising basis for such simulations. In addition, the hypergraph formalism is also rendered into program syntax in a fairly straightforward way. However, the formalism does pose a challenge with respect to the accessibility criterion of the PPLs. The challenge is due to an inherent ambiguity present in the composite hypergraph produced by the FR product, which has 12 edges in the EPR experiment. Four of those edges represent "actual" measurement contexts (depicted in Fig 1), whilst the remaining eight edges impose the no-signalling condition. The hypergraph formalism is

agnostic to this distinction, which is important to distinguish when designing and programming simulations.

On the other hand, the strength of the hypergraph formalism is its flexibility and modularity. In particular, modularity offers the potential to cover a wide variety of experimental designs whilst at the same time offering a conceptually simple route to program specification. We have shown that the EPR experiment is based on two modules which represent measurements on the individual systems *A* and *B*. More generally, joint measurements on multiple systems, and the constraints they must satisfy can then be expressed in terms of a composition operator that combines the modules into a suitable global data structure in the program, which underpins both the sampling and simulation.

With regard to sampling, an immediate suggestion is stronger control-flow integration in the sampling process. Rather than repeatedly generating distributions that are indexed for random results, all PPLs should offer atomic sampling similar to the likes of Pyro or Figaro, where single values could be observed, or returned from a `Stream` primitive in a single procedure. As a sampling process contains variables that are akin to those of iterative loops, it may also serve PPLs to re-imagine the sampling process as a paradigm of the contained programming language, rather than as a single procedure that operates in isolation of the entire program. Providing control over each iteration of the distribution could also improve the legibility of the program, while minimising convolution in the procedures that typically come afterwards.

In terms of the qualitative comparison between the four PPLs, Figaro demonstrated the most benefits for general usage. This could be seen in its capacity to deliver specialised features where other PPLs could not. For what features the alternatives provide, they may appropriately match Figaro i.e., consider that PyMC3 offers more configurations for distributions described in probabilistic models, or that Pyro achieves control-flow independence with fewer syntactic constructs. Such arguments have been overlooked when taking into account the efforts that the main developer, Charles River Analytics, has made to ensure that its PPL is competitively implemented in wider applications. For the likes of accessibility, Figaro's origins in Scala do not present the same benefits as Pyro in agile development. However, the simulation of quantum correlations is a rigorous process. Furthermore, acceleration of PPLs was observed to be minimal in difference (exempt of PyMC3) for the case of the experimentation. Thus it wouldn't be perceived that this is a determinant factor.

In experimentation, we found that Pyro provided the syntactic constructs needed to neatly describe its processes in fewer procedures than those of the others. While PyMC3's origins in Python also made it an expressive alternative, the excessive nomenclature surrounding the declarations of methods and data-types for both Turing.jl and Figaro convoluted their descriptions. While in comparison to the other PPLs, Figaro's accuracy is inferior, it could be argued that the sample iterations describe an experimental setting that does not consider improving float precision. Coupled with the trend of Figaro's improvement in its number of iterations, and the measure of accuracy between PPLs may converge. The same cannot be said for the time complexity of the EPR experimentation, where it was observed that PyMC3's compilation grew substantially with the number of sample iterations being executed. Still, in instances where accuracy is a key factor and limitations are perceived in Pyro's functionality, PyMC3 would be the suitable alternative.

Conclusion

Probabilistic programming offers new possibilities for quantum physicists to specify and simulate experiments, such as the EPR experiment illustrated in this article. This is particularly relevant for experiments requiring advanced statistical inference on unknown parameters,

especially in the case of techniques that involve large amounts of data and computational time. Furthermore, probabilistic machine learning models that are conveniently expressed in probabilistic programming languages can advance our understanding of the underlying physics of the experiments.

It is important to note that the benefits of probabilistic programming are not restricted to experiments involving the analysis of quantum correlations. Since any probabilistic programming language is based on random variables, we can ask the question what exactly is a random variable in quantum physics. Focusing on a single measurement context, due to the normalization constraint, we can think of the measurement context as a (conditional) probability distribution over random variables, which describes the measurement outcomes. The probability distribution is a normalized measure over the sigma algebra defined by the outcomes. This measure is defined via Born's rule, that is, the quantum state is embedded in the measure. An EPR experiment is essentially a state preparation protocol where deterministic operations are embedded in the quantum state (the unitary operations leading to its preparation), followed by the measurement, which results in stochastic outcomes. More generally, we can think of a larger system where we only measure a subsystem. This leads to quantum channels, which are described by completely positive trace preserving maps. A quantum channel, however, must be deterministic in the sequence of events, and, for instance, a measurement choice at a later time step cannot depend on the outcome of a previous measurement. We must factor in such classical and quantum memory effects, as well as the potentially indeterminate causal order of events. The quantum comb [22, 23] or process matrix [24–26] formalism addresses these more generic requirements. Either formalism introduces a generalized Born's rule, where deterministic and stochastic parts of the system clearly separate, and thus give a clear way of defining random variables. Probabilistic programming offers potential in expressing models designed in these frameworks.

Acknowledgments

This research was supported by the Asian Office of Aerospace Research and Development (AOARD) grant: FA2386-17-1-4016. This research was supported by Perimeter Institute for Theoretical Physics. Research at Perimeter Institute is supported by the Government of Canada through Industry Canada and by the Province of Ontario through the Ministry of Economic Development and Innovation.

Author Contributions

Conceptualization: Abdul Karim Obeid, Peter D. Bruza, Peter Wittek.

Data curation: Peter D. Bruza, Peter Wittek.

Formal analysis: Abdul Karim Obeid, Peter D. Bruza, Peter Wittek.

Funding acquisition: Peter D. Bruza.

Investigation: Peter D. Bruza, Peter Wittek.

Methodology: Abdul Karim Obeid, Peter D. Bruza, Peter Wittek.

Project administration: Peter D. Bruza, Peter Wittek.

Resources: Abdul Karim Obeid, Peter Wittek.

Software: Abdul Karim Obeid, Peter Wittek.

Supervision: Peter D. Bruza, Peter Wittek.

Validation: Abdul Karim Obeid, Peter D. Bruza, Peter Wittek.

Visualization: Abdul Karim Obeid, Peter D. Bruza, Peter Wittek.

Writing – original draft: Abdul Karim Obeid, Peter D. Bruza, Peter Wittek.

Writing – review & editing: Abdul Karim Obeid, Peter D. Bruza, Peter Wittek.

References

1. Gordon A, Henzinger TA, Nori A, Rajamani S. Probabilistic programming. In: Proceedings of FOSE-14, Future of Software Engineering. ACM Press; 2014. p. 167–181.
2. Goodman ND, Stuhlmüller A. The Design and Implementation of Probabilistic Programming Languages; 2014. http://dippl.org.
3. Bell JS. On the Einstein-Podolsky-Rosen Paradox. Physics. 1964; 1(3):195–200. https://doi.org/10.1103/PhysicsPhysiqueFizika.1.195
4. Acín A, Fritz T, Leverrier A, Sainz AB. A Combinatorial Approach to Nonlocality and Contextuality. Communications in Mathematical Physics. 2015; 334(2):533–628. https://doi.org/10.1007/s00220-014-2260-1
5. Einstein A, Podolsky B, Rosen N. Can quantum-mechanical description of physical reality be considered complete? Physical Review. 1935; 47(10):777. https://doi.org/10.1103/PhysRev.47.777
6. Shimony A. Bell's Theorem. In: Zalta EN, editor. The Stanford Encyclopedia of Philosophy. summer 2009 ed. Stanford University; 2009. Available from: http://plato.stanford.edu/archives/sum2009/entries/bell-theorem/.
7. Bruza PD. Modelling contextuality by probabilistic programs with hypergraph semantics. Theoretical Computer Science. 2017;
8. Carpenter B, Gelman A, Hoffman MD, Lee D, Goodrich B, Betancourt M, et al. Stan: A probabilistic programming language. Journal of Statistical Software. 2017; 76(1):1. https://doi.org/10.18637/jss.v076.i01
9. Goodman N, Mansinghka V, Roy DM, Bonawitz K, Tenenbaum JB. Church: a language for generative models. In: Proceedings of UAI-08, 24th Conference on Uncertainty in Artificial Intelligence; 2008. p. 220–229.
10. Tran D, Kucukelbir A, Dieng AB, Rudolph M, Liang D, Blei DM. Edward: A library for probabilistic modeling, inference, and criticism. 2016;.
11. Salvatier J, Wiecki TV, Fonnesbeck C. Probabilistic programming in Python using PyMC3. PeerJ Computer Science. 2016; 2:e55. https://doi.org/10.7717/peerj-cs.55
12. Charles River Analytics. Figaro; 2017. Available from: https://github.com/p2t2/figaro.
13. Ge H, Xu K, Scibior A, Ghahramani Z, et al. The Turing language for probabilistic programming. 2016;.
14. Bingham E, Chen JP, Jankowiak M, Obermeyer F, Pradhan N, Karaletsos T, et al. Pyro: Deep Universal Probabilistic Programming. 2018;.
15. Noack A, Arsian A, Garborg S, Stephen G, Chen J, Pearson J, et al. Distributions.jl; 2017. Available from: https://juliastats.github.io/Distributions.jl/stable/.
16. Milch B, Russell S. Extending Bayesian networks to the open-universe case. Heuristics, Probability and Causality: A Tribute to Judea Pearl College Publications. 2010;.
17. Al-Rfou R, Alain G, Almahairi A, Angermueller C, Bahdanau D, Ballas N, et al. Theano: A Python framework for fast computation of mathematical expressions. 2016;.
18. Peng T. RIP Theano; 2017. Available from: https://syncedreview.com/2017/09/29/rip-theano/.
19. Goodman N. Uber AI Labs Open Sources Pyro, a Deep Probabilistic Programming Language; 2017. Available from: https://eng.uber.com/pyro/.
20. Paszke A, Gross S, Chintala S, Chanan G. PyTorch: Tensors and dynamic neural networks in Python with strong GPU acceleration. 2017;.
21. Popescu S, Rohrlich D. Quantum nonlocality as an axiom. Foundations of Physics. 1994; 24(3):379–385. https://doi.org/10.1007/BF02058098
22. Chiribella G, D'Ariano GM, Perinotti P. Quantum circuit architecture. Physical Review Letters. 2008; 101(6):060401. https://doi.org/10.1103/PhysRevLett.101.060401 PMID: 18764438
23. Chiribella G, D'Ariano GM, Perinotti P. Theoretical framework for quantum networks. Physical Review A. 2009; 80(2):022339. https://doi.org/10.1103/PhysRevA.80.022339

24. Oreshkov O, Costa F, Brukner C. Quantum correlations with no causal order. Nature Communications. 2012; 3:1092. https://doi.org/10.1038/ncomms2076 PMID: 23033068

25. Pollock FA, Rodríguez-Rosario C, Frauenheim T, Paternostro M, Modi K, Pollock FA, et al. Non-Markovian quantum processes: Complete framework and efficient characterization. Physical Review A. 2018; 97(1):012127. https://doi.org/10.1103/PhysRevA.97.012127

26. Pollock FA, Rodríguez-Rosario C, Frauenheim T, Paternostro M, Modi K. Operational Markov Condition for Quantum Processes. Physical Review Letters. 2018; 120(4):040405. https://doi.org/10.1103/PhysRevLett.120.040405 PMID: 29437441