Evaluating probabilistic programming languages for simulating quantum correlations

Abdul Obeid[1], Peter WIttek[1], Peter D. Bruza[1]
[1]Co-Author

dx.doi.org/10.17504/protocols.io.u79ezr6

Abdul Obeid
QUT

PROTOCOL STATUS

In development
We are still developing and optimizing this protocol

Set up of environment required for PyMC3, Pyro & Turing.jl (MAC OS X)

1 Navigate to the folder in which you would like to conduct the experimentation.

Then open your terminal application:

SOFTWARE
Terminal
Mac OS X
by Apple

COMMAND
```
virtualenv env
```
Create a virtual environment to house the required packages.

COMMAND
```
source env/bin/activate
```
Activate the environment

Then create a new file entitled "pymc3-main.py", and populate it with the following text:

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


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,
```

```
            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

# execution

def accuracy_time(N):
    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.]]) ]]
    start = time.time()
    Q = generate_global_distribution(constraints,N)
    end = time.time()
    p = Q
    A11 = (2 * (p[0] + p[1])) - 1
    A12 = (2 * (p[4] + p[5])) - 1
    A21 = (2 * (p[8] + p[9])) - 1
    A22 = (2 * (p[12] + p[13])) - 1
    B11 = (2 * (p[0] + p[2])) - 1
    B12 = (2 * (p[8] + p[10])) - 1
    B21 = (2 * (p[4] + p[6])) - 1
    B22 = (2 * (p[12] + p[14])) - 1
    delta = (abs(A11 - A12) + abs(A21 - A22) + abs(B11 - B21) + abs(B12 - B22))/2
    A11B11 = (p[0] + p[3]) - (p[1] + p[2])
    A12B12 = (p[4] + p[7]) - (p[5] + p[6])
    A21B21 = (p[8] + p[11]) - (p[9] + p[10])
    A22B22 = (p[12] + p[15]) - (p[13] + p[14])
    print("Time:")
    print(end - start)

    print("Normalization in contexts: ", [p[0]+p[1]+p[6]+p[7]])
    print("Normalization in contexts: ", [p[2]+p[3]+p[4]+p[5]])
    print("Normalization in contexts: ", [p[8]+p[9]+p[14]+p[15]])
    print("Normalization in contexts: ", [p[10]+p[11]+p[12]+p[13]])

    print("delta: ", delta)
    print("Potential violations: ")
    print(abs(A11B11 + A12B12 + A21B21 - A22B22), 2 * (1 + delta))
    print(abs(A11B11 + A12B12 - A21B21 + A22B22), 2 * (1 + delta))
    print(abs(A11B11 - A12B12 + A21B21 + A22B22), 2 * (1 + delta))
    print(abs(-A11B11 + A12B12 + A21B21 + A22B22), 2 * (1 + delta))


accuracy_time(1000)
accuracy_time(2000)
accuracy_time(3000)
accuracy_time(4000)
accuracy_time(5000)
accuracy_time(6000)
accuracy_time(7000)
accuracy_time(8000)
accuracy_time(9000)
accuracy_time(10000)
accuracy_time(15000)
accuracy_time(20000)
accuracy_time(25000)
accuracy_time(30000)
accuracy_time(35000)
accuracy_time(40000)
```

```
accuracy_time(45000)
accuracy_time(50000)
accuracy_time(55000)
accuracy_time(60000)
accuracy_time(65000)
accuracy_time(70000)
accuracy_time(75000)
accuracy_time(80000)
accuracy_time(85000)
accuracy_time(90000)
accuracy_time(95000)
accuracy_time(100000)
```

Then create a new file entitled "pyro-main.py", and populate it with the following text:

COMMAND

```
from pyro import sample
import torch
from numpy import zeros, array, fliplr, sum
from functools import reduce
from itertools import product
from pyro.distributions import Bernoulli, Uniform
import pprint
import sys
import time

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)
```

```
            fr_edges.append(fr_edge)
    return fr_edges

def variable(v):
    return torch.autograd.Variable(torch.Tensor([v]))

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 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(0.5))))
        b = int(sample('B', Bernoulli(variable(0.5))))
        x = int(sample('X', Bernoulli(variable(0.5))))
        y = int(sample('Y', Bernoulli(variable(0.5))))
        value = float(sample('C', Uniform(variable(0.0), variable(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
    for a, b, x, y in product(range(2), range(2), range(2), range(2)):
        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






def accuracy_time(N):
    print("Iterations %s" % (N))
    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.]]) ]]
    start = time.time()
    Q = generate_global_distribution(constraints,N)
    end = time.time()
    p = Q
    A11 = (2 * (p[0] + p[1])) - 1
    A12 = (2 * (p[4] + p[5])) - 1
    A21 = (2 * (p[8] + p[9])) - 1
    A22 = (2 * (p[12] + p[13])) - 1
    B11 = (2 * (p[0] + p[2])) - 1
    B12 = (2 * (p[8] + p[10])) - 1
    B21 = (2 * (p[4] + p[6])) - 1
    B22 = (2 * (p[12] + p[14])) - 1
    delta = (abs(A11 - A12) + abs(A21 - A22) + abs(B11 - B21) + abs(B12 - B22))/2
    A11B11 = (p[0] + p[3]) - (p[1] + p[2])
    A12B12 = (p[4] + p[7]) - (p[5] + p[6])
    A21B21 = (p[8] + p[11]) - (p[9] + p[10])
    A22B22 = (p[12] + p[15]) - (p[13] + p[14])
    print("Time:")
    print(end - start)
```

```
    print("Normalization in contexts: ", [p[0]+p[1]+p[6]+p[7]])
    print("Normalization in contexts: ", [p[2]+p[3]+p[4]+p[5]])
    print("Normalization in contexts: ", [p[8]+p[9]+p[14]+p[15]])
    print("Normalization in contexts: ", [p[10]+p[11]+p[12]+p[13]])

    print("delta: ", delta)
    print("Potential violations: ")
    print(abs(A11B11 + A12B12 + A21B21 - A22B22), 2 * (1 + delta))
    print(abs(A11B11 + A12B12 - A21B21 + A22B22), 2 * (1 + delta))
    print(abs(A11B11 - A12B12 + A21B21 + A22B22), 2 * (1 + delta))
    print(abs(-A11B11 + A12B12 + A21B21 + A22B22), 2 * (1 + delta))


accuracy_time(1000)
accuracy_time(2000)
accuracy_time(3000)
accuracy_time(4000)
accuracy_time(5000)
accuracy_time(6000)
accuracy_time(7000)
accuracy_time(8000)
accuracy_time(9000)
accuracy_time(10000)
accuracy_time(15000)
accuracy_time(20000)
accuracy_time(25000)
accuracy_time(30000)
accuracy_time(35000)
accuracy_time(40000)
accuracy_time(45000)
accuracy_time(50000)
accuracy_time(55000)
accuracy_time(60000)
accuracy_time(65000)
accuracy_time(70000)
accuracy_time(75000)
accuracy_time(80000)
accuracy_time(85000)
accuracy_time(90000)
accuracy_time(95000)
accuracy_time(100000)
```

Then create a new file entitled "turing-main.jl", and populate it with the following text:

COMMAND

```
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 float(n)
 convert(Float64,n)
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])
    l = [convert(Float64,a[i]), convert(Float64,b[i]),convert(Float64,x[i]), convert(Float64,y[i])]
    associated_hyperedges = get_hyperedges(hyperedges, l)
    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
  l = [ convert(Float64,a), convert(Float64,b), convert(Float64,x), convert(Float64,y)]
  associated_hyperedges = get_hyperedges(hyperedges, l)
  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

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]] ]]




function accuracy_time(N)
    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]] ]]
 tic()
    Q = generate_global_distribution(constraints,N)
    toc()
    p = Q
    A11 = (2 * (p[1] + p[2])) - 1
    A12 = (2 * (p[5] + p[6])) - 1
    A21 = (2 * (p[9] + p[10])) - 1
    A22 = (2 * (p[13] + p[14])) - 1
    B11 = (2 * (p[1] + p[3])) - 1
    B12 = (2 * (p[9] + p[11])) - 1
    B21 = (2 * (p[5] + p[7])) - 1
    B22 = (2 * (p[13] + p[15])) - 1
    delta = (abs(A11 - A12) + abs(A21 - A22) + abs(B11 - B21) + abs(B12 - B22))/2
    A11B11 = (p[1] + p[4]) - (p[2] + p[3])
    A12B12 = (p[5] + p[8]) - (p[6] + p[7])
    A21B21 = (p[9] + p[12]) - (p[10] + p[11])
    A22B22 = (p[13] + p[16]) - (p[14] + p[15])

    println(p[1]+p[2]+p[7]+p[8])
    println(p[3]+p[4]+p[5]+p[6])
    println(p[9]+p[10]+p[15]+p[16])
    println(p[11]+p[12]+p[13]+p[14])

    a = abs(1-(p[1]+p[2]+p[7]+p[8]))
    b = abs(1-(p[3]+p[4]+p[5]+p[6]))
    c = abs(1-(p[9]+p[10]+p[15]+p[16]))
```

```
    d = abs(1-(p[11]+p[12]+p[13]+p[14]))
    println(delta)
    println(2 * (1 + delta))
    print(abs(A11B11 + A12B12 + A21B21 - A22B22))
    print(abs(A11B11 + A12B12 - A21B21 + A22B22))
    print(abs(A11B11 - A12B12 + A21B21 + A22B22))
    print(abs(-A11B11 + A12B12 + A21B21 + A22B22))
end



accuracy_time(1000)
accuracy_time(2000)
accuracy_time(3000)
accuracy_time(4000)
accuracy_time(5000)
accuracy_time(6000)
accuracy_time(7000)
accuracy_time(8000)
accuracy_time(9000)
accuracy_time(10000)
accuracy_time(15000)
accuracy_time(20000)
accuracy_time(25000)
accuracy_time(30000)
accuracy_time(35000)
accuracy_time(40000)
accuracy_time(45000)
accuracy_time(50000)
accuracy_time(55000)
accuracy_time(60000)
accuracy_time(65000)
accuracy_time(70000)
accuracy_time(75000)
accuracy_time(80000)
accuracy_time(85000)
accuracy_time(90000)
accuracy_time(95000)
accuracy_time(100000)
```

Running PyMC3 & Pyro

2

To run the file "pymc3-main.py", you will firstly need to install PyMC3 via the following page: https://docs.pymc.io/

Then run the following command from the terminal:

COMMAND

```
python "pymc3-main.py"
```

To run the file "pyro-main.py", you will firstly need to install Pyro via the following page: http://pyro.ai/

Then run the following command from the terminal:

COMMAND

```
python "pyro-main.py"
```

Running Turing.jl

3

To run the Turing implementation, you will need to install Julia:
https://julialang.org/

Then install both the "Distributions" package, and "Turing" package from within the Julia package manager.

Finally, run the implementation using the following command:

COMMAND

```
julia "turing-main.jl"
```

Running Figaro

4

The Figaro implementation will require the following software:

SOFTWARE

IntelliJ IDEA Community Editio

by Jetbrains

Create a new project, and add the following jar to your project "figaro_2.11-5.0.0.0-sources.jar" from the Figaro web page:
https://www.cra.com/work/case-studies/figaro

Then run the implementation from within the IntelliJ IDEA Community Edition IDE.