Probabilistic Programming and Artificial Intelligence

TU Wien, 194.150, VU 6 ECTS

Jürgen Cito
Markus Böck

TU Wien, Austria

TU Informatics

Probabilistic Programming and AI: Organization

All information on TISS/TUWEL and website: https://probprog-ai-tuwien.github.io/2023/

Registration

Deadline: October 2nd
Drop-date: October 17th (coincides with deadline for A1)

Modality/Grading:

6 Lectures, 4 Assignments, 2 Assignment discussions (mandatory), 1 Group project
Grading: 40% Assignments, 60% projects, no exam

Elective:

066 645 **Data Science**

066 926 **Business Informatics**

066 931 **Logic and Computation**

066 937 **Software Engineering & Internet Computing**

Probabilistic Programming

1. Represent probability distributions as ~~formulas~~ **programs that generate samples *from possible worlds***

2. Build generic algorithms for probabilistic conditioning/inference using probabilistic programs as representations

Bayesian statistics

- Express statistical assumptions via probability distributions

$$\underbrace{\textbf{Pr(parameters, data)}}_{\text{joint}} = \underbrace{\textbf{Pr(parameters)}}_{\text{prior}}\ \underbrace{\textbf{Pr(data | parameters)}}_{\text{likelihood}}$$

- Inference from data (via conditioning)

$$\underbrace{\textbf{Pr(parameters | data = x)}}_{\text{posterior}}$$

Slide adapted from Dan Roy, “A Personal Viewpoint on Probabilistic Programming"

Probabilistic Programming and AI: What is thinking?

How can we describe the intelligent inferences made in everyday human reasoning?

How can we engineer intelligent machines?

Computational theory of mind

mind = computer

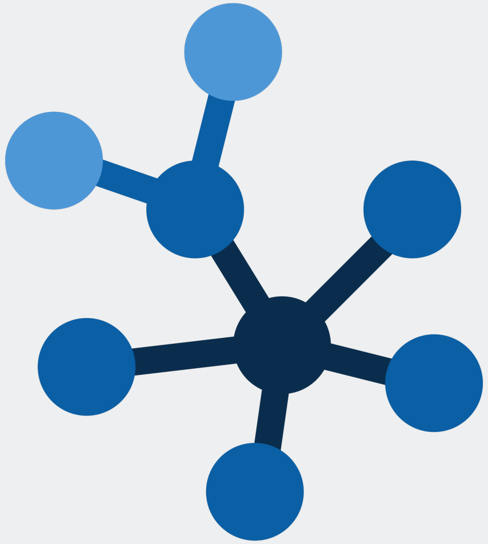
mental representations = computer programs

run(program)

thinking = running a program

Probabilistic Programming and AI: What kind of program can represent thinking?

Structure



Knowledge

Probability

Uncertainty

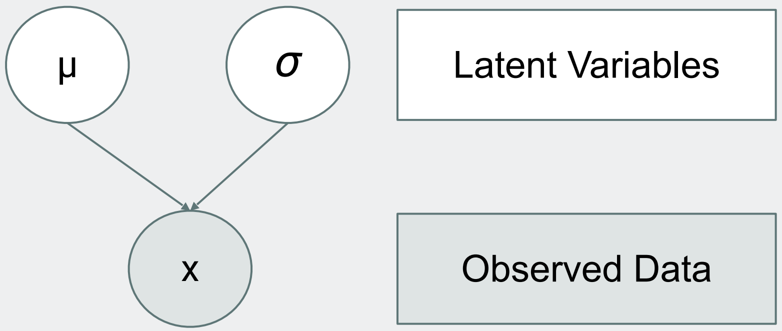
TU WIEN Informatics

Tenenbaum, Kemp, Griffiths, & Goodman (2011) How to grow a mind: Statistics, structure, and abstraction. Science

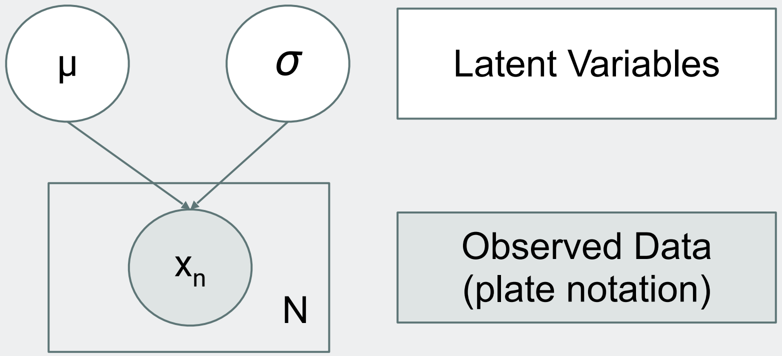
Slide adapted from Tobias Gerstenberg, "Mental Models as Probabilistic Programs"

Probabilistic Models: Brief Teaser

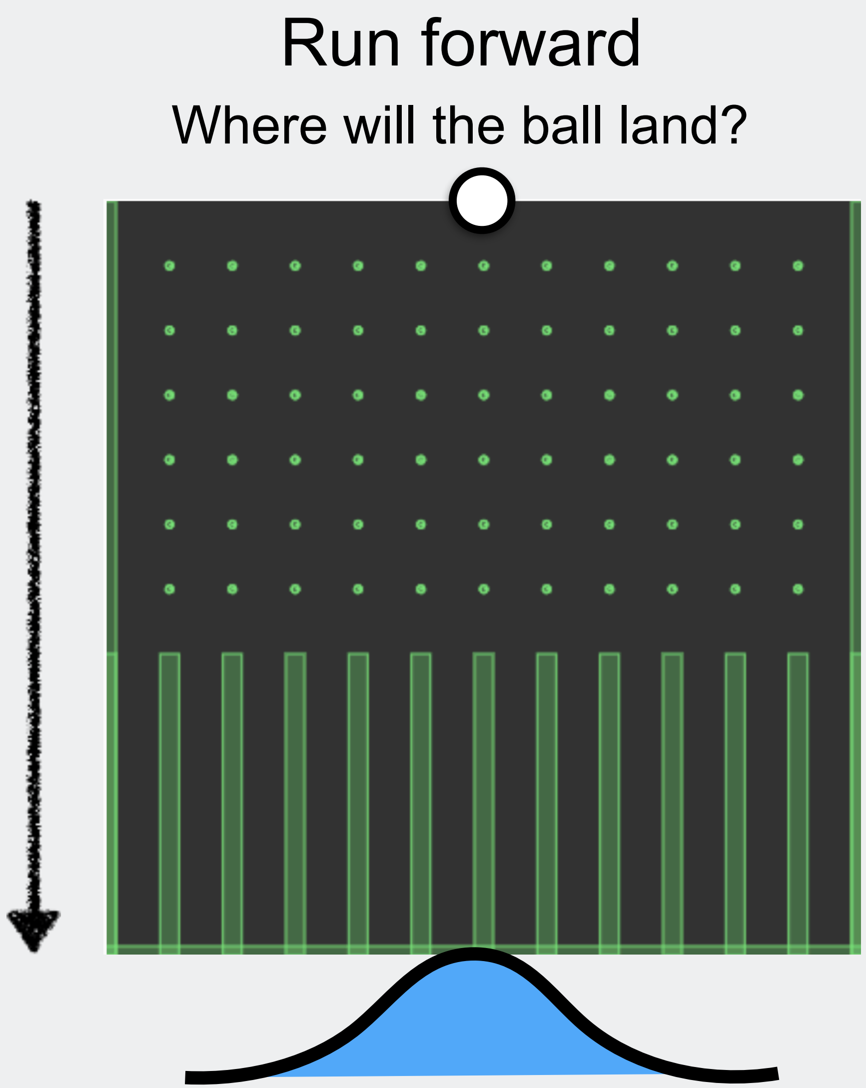
$\mu \sim Normal(0, 10)$
$\sigma \sim LogNormal(0, 5)$
$x \sim Normal(\mu, \sigma)$

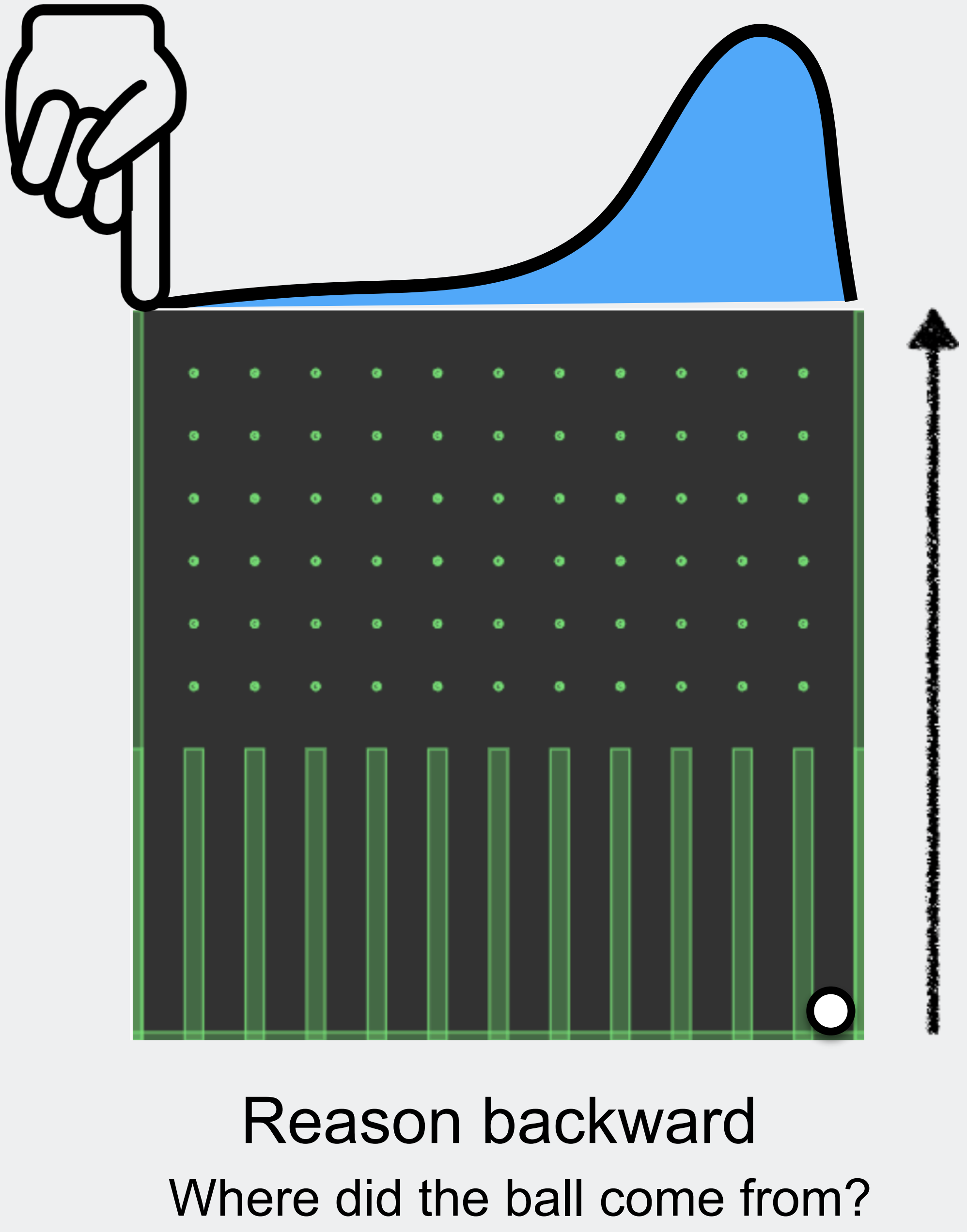


$\mu \sim Normal(0, 10)$
$\sigma \sim LogNormal(0, 5)$
$x_1, \ldots x_n \sim^{iid} Normal(\mu, \sigma)$



Probabilistic Inference

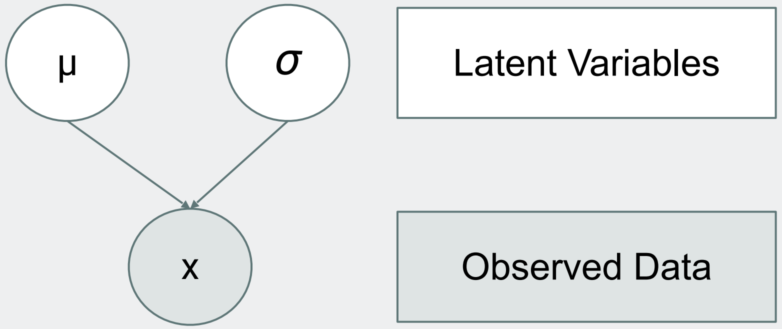




Slide adapted from Tobias Gerstenberg, "Mental Models as Probabilistic Programs"

Probabilistic Models: Brief Teaser

```
μ ~ Normal(0, 10)
σ ~ LogNormal(0, 5)
x ~ Normal(μ, σ)
```



Why are probabilistic models useful?

- Express prior knowledge about the world
- Incorporate noisy data
- Handling uncertainty

Why are Probabilistic Programming Languages useful? **Expressivity!**

TU WIEN Informatics

Probabilistic Programming: Expressivity

Why are PPLs useful? **Expressivity!**

- Probabilistic modeling and inference as first class citizens of a programming language
- Ability to express rich probabilistic models through stochastic control flow (beyond Probabilistic Graphical Models and Bayesian Networks)
- Separating modeling from inference
- **Enable incorporation of programming language and software engineering advances**

Separating Probabilistic Modeling and Inference

Represent probability distributions by programs that generate samples (simulators)

Using general purpose algorithms for inference using probabilistic programs as model representations

Probabilistic Model

```
@gen function line_model(xs::Vector{Float64})
    slope = ({:slope} ~ normal(0, 1))
    intercept = ({:intercept} ~ normal(0, 2))

    for (i, x) in enumerate(xs)
        ({(:y, i)} ~ normal(slope * x + intercept, 0.1))
    end
    return length(xs)
end;
```

Inference

```
function do_inference(model, xs, ys, amount_of_computation)
    observations = Gen.choicemap()
    for (i, y) in enumerate(ys)
        observations[(:y, i)] = y
    end

    (trace, _) = Gen.importance_resampling(model, (xs,),
                     observations, amount_of_computation);
    return trace
end;

xs = [...];
ys = [...];
sample_trace = Gen.simulate(line_model, (xs, ));
...
trace = do_inference(line_model, xs, ys, 100)
```

TU WIEN Informatics

Simple Probabilistic Program: Bayesian Linear Regression in Gen/Julia

```
@gen function line_model(xs::Vector{Float64})
    slope = ({:slope} ~ normal(0, 1))
    intercept = ({:intercept} ~ normal(0, 2))

    for (i, x) in enumerate(xs)
        ({(:y, i)} ~ normal(slope * x + intercept, 0.1))
    end
    return length(xs)
end;

function do_inference(model, xs, ys, amount_of_computation)
    observations = Gen.choicemap()
    for (i, y) in enumerate(ys)
        observations[(:y, i)] = y
    end

    (trace, _) = Gen.importance_resampling(model, (xs,),
                        observations, amount_of_computation);
    return trace
end;

xs = [...];
ys = [...];
sample_trace = Gen.simulate(line_model, (xs, ));
...
trace = do_inference(line_model, xs, ys, 100)
```

Slope and Intercept as Latent Variables (Priors)

Likelihood defined as model over slope and intercept with fixed noise

Observed variable samples explicitly recorded in a trace *{(:y, i)}*

Probabilistic Modeling as Generative Function (Simulator)

Observations recalled in inference step and passed to inference procedure

Approximative inference with importance resampling (can be exchanged with MCMC, HMC, NUTS, etc.)

Inference separated from modeling

TU WIEN Informatics

Software Engineering for Probabilistic Programming

```
@gen function line_model(xs::Vector{Float64})
    slope = ({:slope} ~ normal(0, 1))
    intercept = ({:intercept} ~ normal(0, 2))

    for (i, x) in enumerate(xs)
        ({(:y, i)} ~ normal(slope * x + intercept, 0.1))
    end
    return length(xs)
end;

function do_inference(model, xs, ys, amount_of_computation)
    observations = Gen.choicemap()
    for (i, y) in enumerate(ys)
        observations[(:y, i)] = y
    end

    (trace, _) = Gen.importance_resampling(model, (xs,),
                    observations, amount_of_computation);
    return trace
end;

xs = [...];
ys = [...];
sample_trace = Gen.simulate(line_model, (xs, ));
...
trace = do_inference(line_model, xs, ys, 100)
```

Choice of Priors

Choice of Likelihood

Feedback Cycle

Choice of Inference Method and Parameters

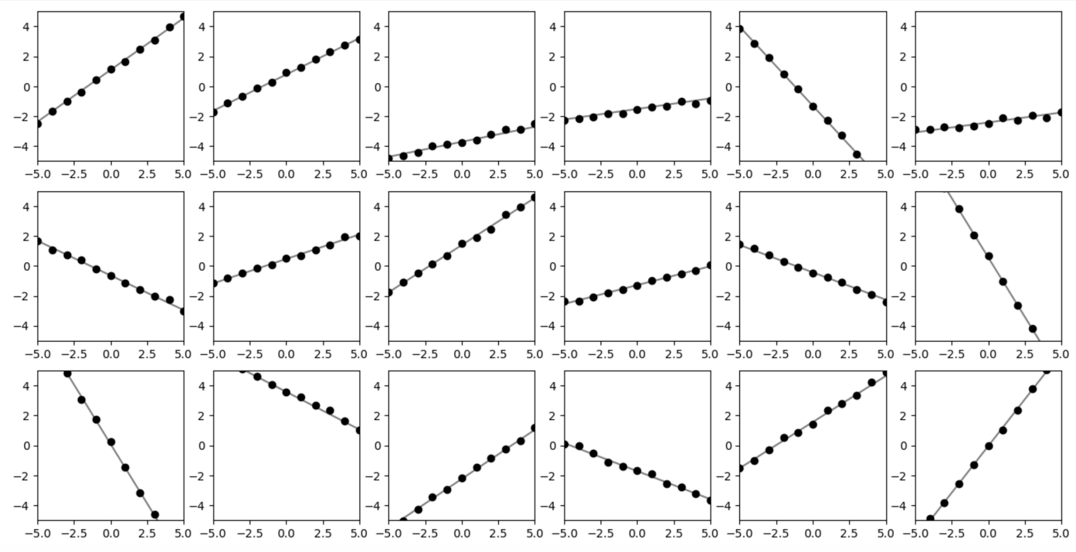
Choice of Domain-specific Visualization

SE for PPL Research in our research group

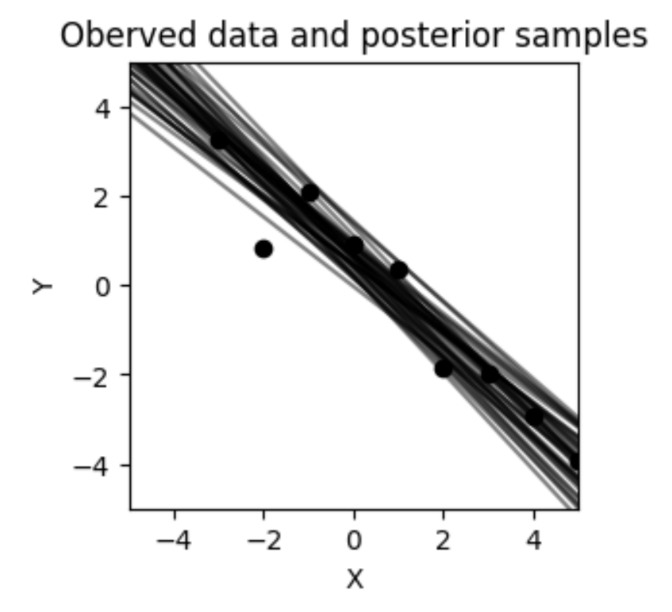
- Program Comprehension (Reasoning about Programs)
- Software Evolution (Reasoning about Change)
- Software Visualization (Reasoning about Large-scale Traces)
- Software Testing (Reasoning about Correctness)

Software Visualization
(Reasoning about Traces)

```
function grid(renderer::Function, traces; ncols=6, nrows=3)
    figure(figsize=(16, 8))
    for (i, trace) in enumerate(traces)
        subplot(nrows, ncols, i)
        renderer(trace)
    end
end;
```



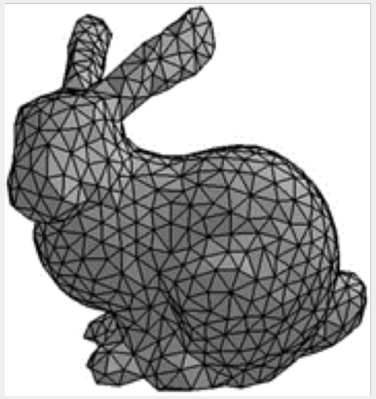
```
function overlay(renderer, traces; same_data=true, args...)
    if !isempty(traces)
        renderer(traces[1], show_data=true, args...)
        for i=2:length(traces)
            renderer(traces[i], show_data=!same_data, args...)
        end
    end
end;
```



Support visualizing of generative worlds and posterior distributions that are embedded in domain

TU WIEN Informatics

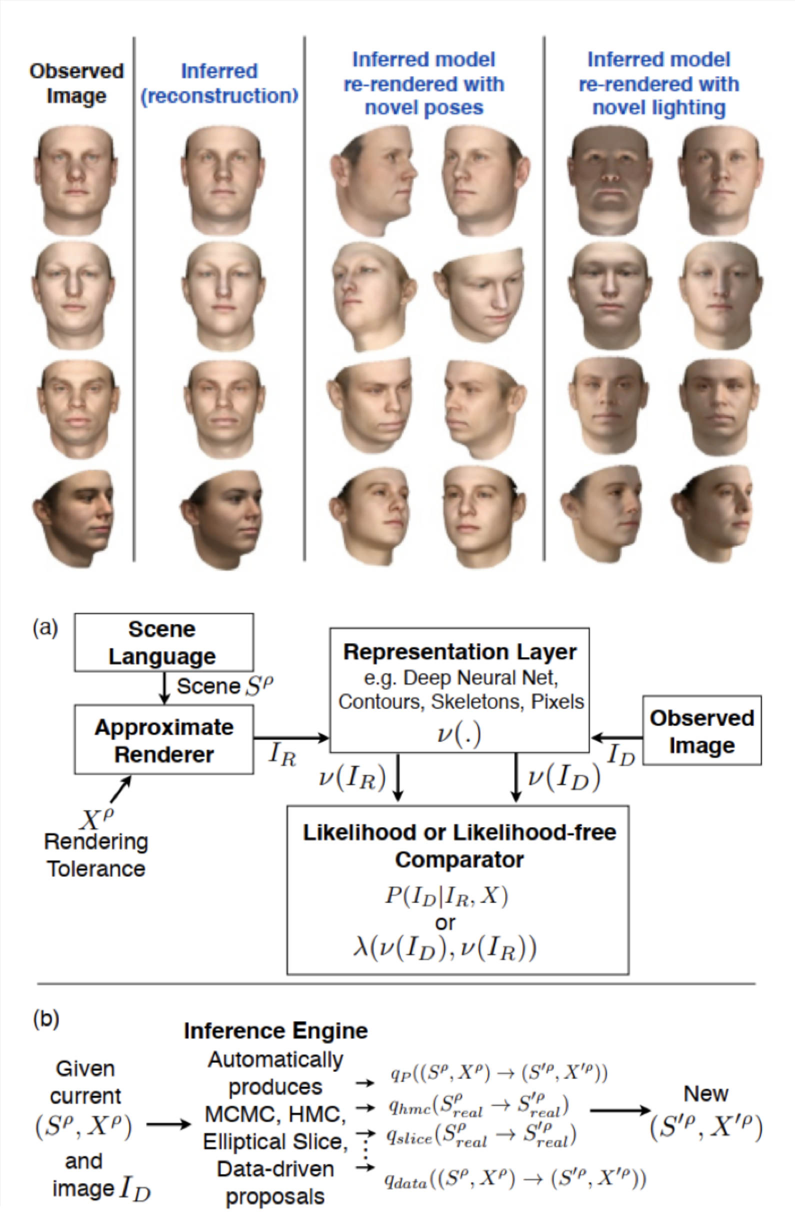
Applications of Probabilistic Programming Languages
Inverse Modeling: Extracting 3D structures from images



inference

Slide adapted from Dan Roy, “A Personal Viewpoint on Probabilistic Programming"

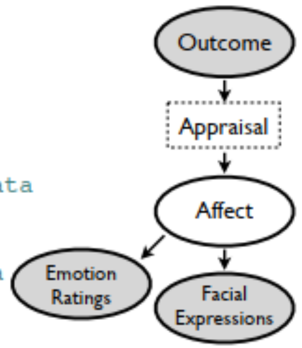
Picture: A probabilistic Programming Language for Scene Perception

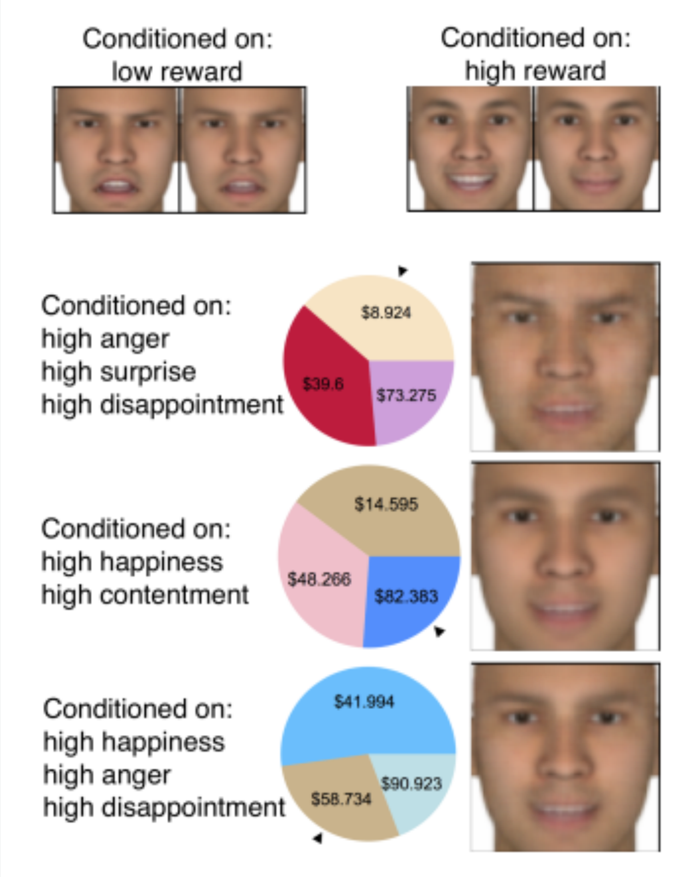



```
function PROGRAM(MU, PC, EV, VERTEX_ORDER)
  # Scene Language: Stochastic Scene Gen
  face=Dict();shape = []; texture = [];
  for S in ["shape", "texture"]
   for p in ["nose", "eyes", "outline", "lips"]
    coeff = MvNormal(0,1,1,99)
    face[S][p] = MU[S][p]+PC[S][p].*(coeff.*EV[S][p])
   end
  end
  shape=face["shape"][:]; tex=face["texture"][:];
  camera = Uniform(-1,1,1,2); light = Uniform(-1,1,1,2)

  # Approximate Renderer
  rendered_img= MeshRenderer(shape,tex,light,camera)

  # Representation Layer
  ren_ftrs = getFeatures("CNN_Conv6", rendered_img)

  # Comparator
  #Using Pixel as Summary Statistics
  observe(MvNormal(0,0.01), rendered_img-obs_img)
  #Using CNN last conv layer as Summary Statistics
  observe(MvNormal(0,10), ren_ftrs-obs_cnn)
end

global obs_img = imread("test.png")
global obs_cnn = getFeatures("CNN_Conv6", img)
#Load args from file
TR = trace(PROGRAM,args=[MU,PC,EV,VERTEX_ORDER])
# Data-Driven Learning
learn_datadriven_proposals(TR,100000,"CNN_Conv6")
load_proposals(TR)
# Inference
infer(TR,CB,20,["DATA-DRIVEN"])
infer(TR,CB,200,["ELLIPTICAL"])
```

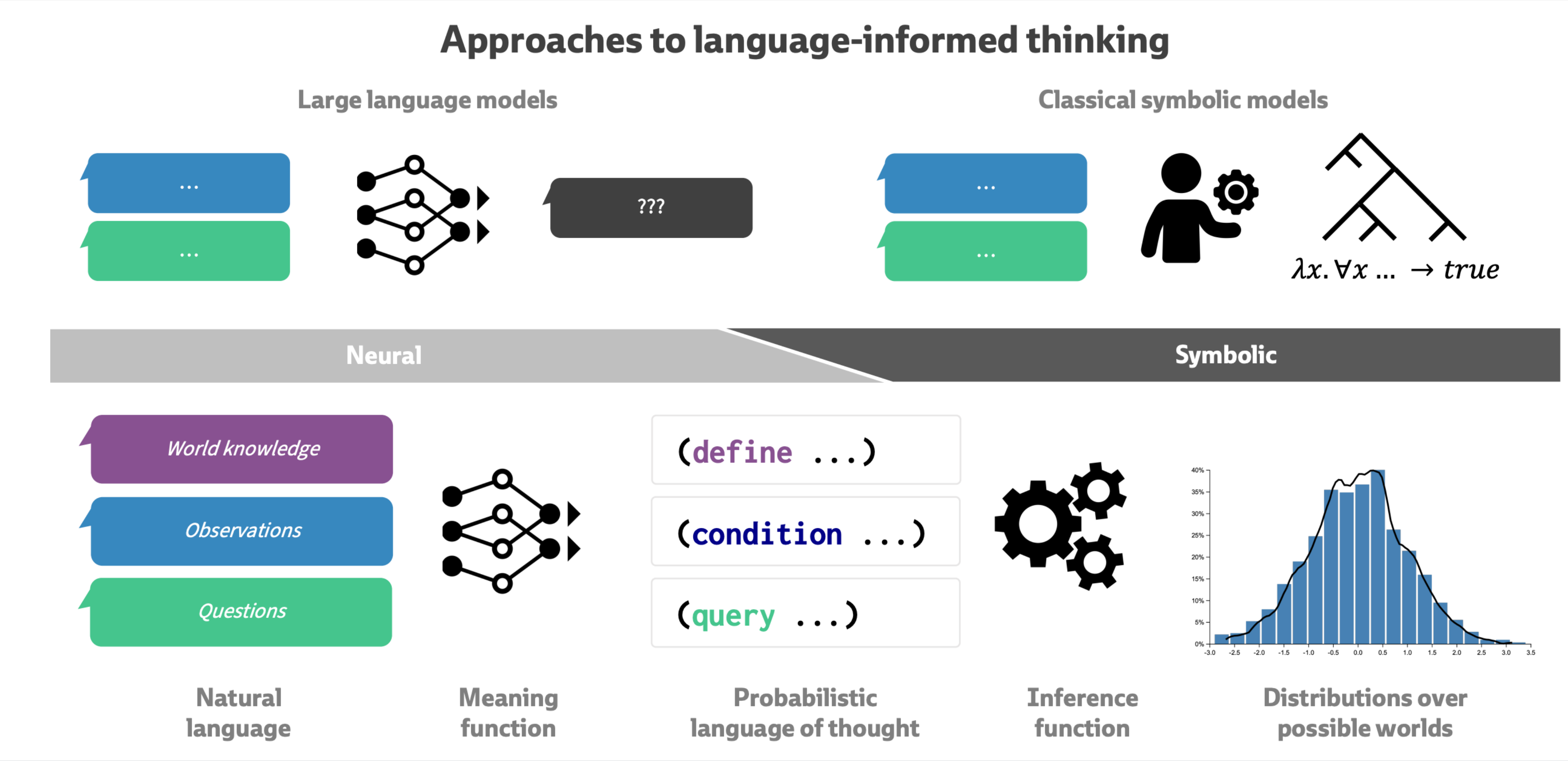
Inferring internal affective states with PPLs

```python
class MultimodalVAE():
  def condition(self, outcome, emotion, image):
    # generate a new emotion from outcome
    prediction_mean = self.outcomes_to_affect(outcome)
    # sample affect given priors
    affect = pyro.sample("affect", Normal(prediction_mean, 1))
    # generate the facial expression, condition on the observed data
    face_mean = self.affectToFace_Decoder(affect)
    face = pyro.sample("face", Bernoulli(face_mean), obs=image)
    # generate the outcome ratings, condition on the observed data
    emo_mean = self.affectToRating_Decoder(affect)
    emo = pyro.sample("emo", Normal(emo_mean, 1), obs=emotion)
```



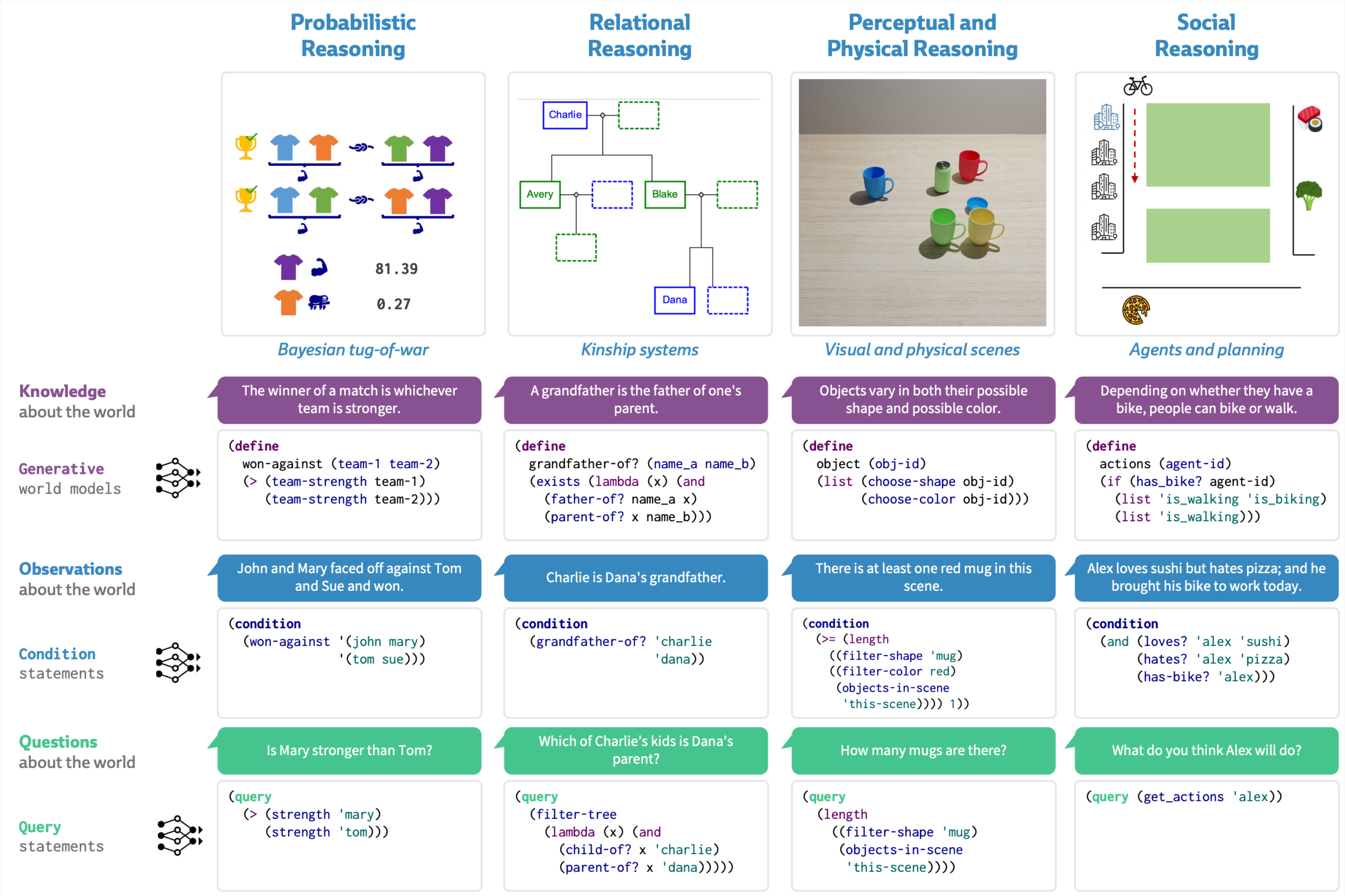


From Word Models to World Models using PPLs



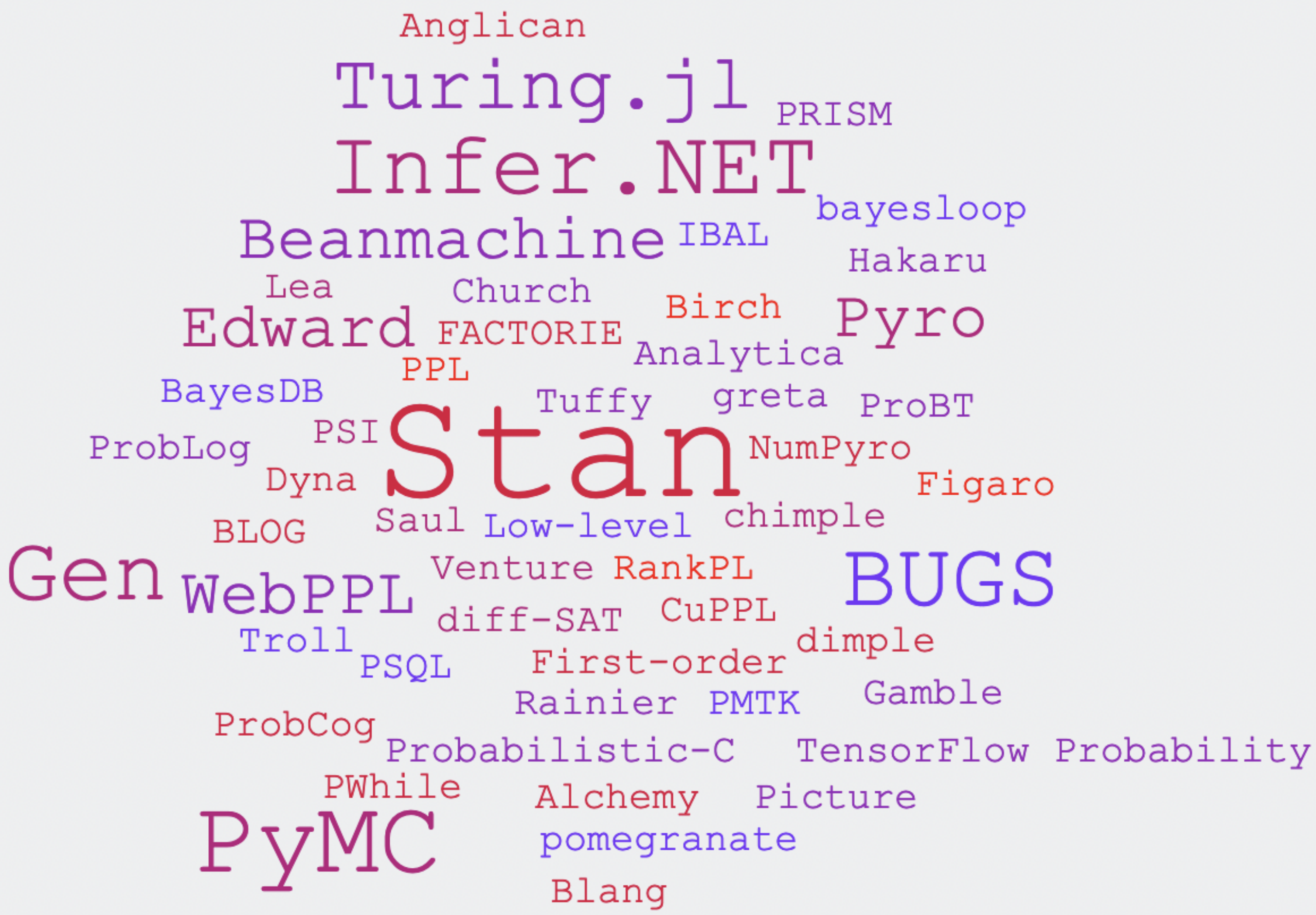
Wong & Grand et al., "From Word Models to World Models: Translating from Natural Language to the Probabilistic Language of Thought"

From Word Models to World Models using PPLs



Wong & Grand et al., "From Word Models to World Models: Translating from Natural Language to the Probabilistic Language of Thought"

Probabilistic Programming Languages (PPLs)



PPL Overview

Classical PPLs

Have their own modeling Language

Offer a limited set of black-box inference methods

Examples:

Bugs, Stan

PPL Overview

Modern PPLs

Are embedded in another language (e.g. Python)

Offer a limited set of black-box inference methods

Examples:

PyMC, Turing.jl

PPL Overview

Deep PPLs

Are embedded in another language (e.g. Python)

Often rely on variational inference and underlying machine learning frameworks (e.g., PyTorch, Tensorflow)

Examples:

Edward, Pyro

PPL Overview

Flexible PPLs

Are embedded in another language (e.g. Python)

Offer the possibility to implement custom inference methods

Examples:

Gen, Beanmachine

PPL Overview

Classical PPLs	Modern PPLs	Deep PPLs	Flexible PPLs
BUGS	PyMC	Edward	Gen
Stan	Turing.jl	Pyro	Beanmachine

PPL Overview

Classical PPLs	Modern PPLs	Deep PPLs	Flexible PPLs
BUGS	PyMC	Edward	**Gen**
Stan	**Turing.jl**	**Pyro**	**Beanmachine**

TU WIEN Informatics

Hello, ~~world~~ coin

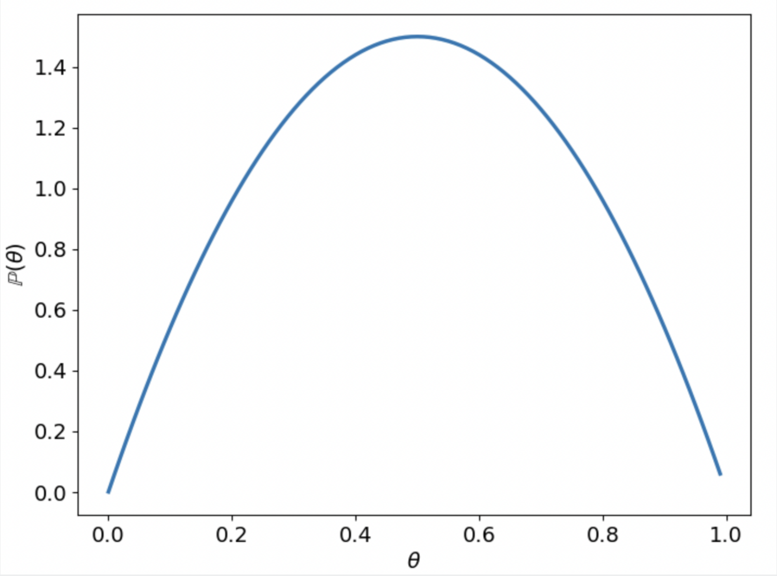
Given 10 coinflips (observations): Can we infer the bias of a Coin?

Model:

  θ ~ Beta(2,2)   // coin bias

  y ~ Bernoulli(θ) // coin flip result

Beta(2,2)



TU WIEN Informatics

Coin Flip Comparison between different PPLs

Examples can be found at:

github.com/ipa-lab/ppl-comparison

TU WIEN Informatics

PPL Comparison

Mathematical Model:

θ ~ Beta(2,2)   // coin bias
y ~ Bernoulli(θ)   // coin flip result

TU WIEN Informatics

PPL Comparison - Modeling

Stan

```
stan_model = """
data {
  int N;
  int y[N];
}
parameters {
  real theta;
}
model {
  theta ~ beta(2, 2);
  for (n in 1:N)
    y[n] ~ bernoulli(theta);
}
"""
```

Gen

```
@gen function my_model(ys::Vector{Bool})
    theta ~ beta(2, 2)
    for (i, y) in enumerate(ys)
        @trace(bernoulli(theta), "y-$i")
    end
end
```

Turing.jl

```
@model function coinflip(y)
    theta ~ Beta(2, 2)
    N = length(y)
    for n in 1:N
        y[n] ~ Bernoulli(theta)
    end
end
```

Pyro

```
def simple_model(flips=None):
    a = pyro.param("a", lambda: torch.tensor(2.0))
    b = pyro.param("b", lambda: torch.tensor(2.0))
    theta = pyro.sample("theta", distP.Beta(a,b))

    with pyro.plate("data"):
        return pyro.sample("obs", dist.Bernoulli(theta), obs=flips)
```

Beanmachine

```
#Heads rate
@bm.random_variable
def theta():
    return dist.Beta(2, 2)

#coin flip
@bm.random_variable
def y(i: int):
    return dist.Bernoulli(theta())
```

PPL Comparison - Inference

Turing (representative for black-box inference)

```
chain = sample(coinflip(data), MH(), 1000)
```

Pyro (representative for variational inference)

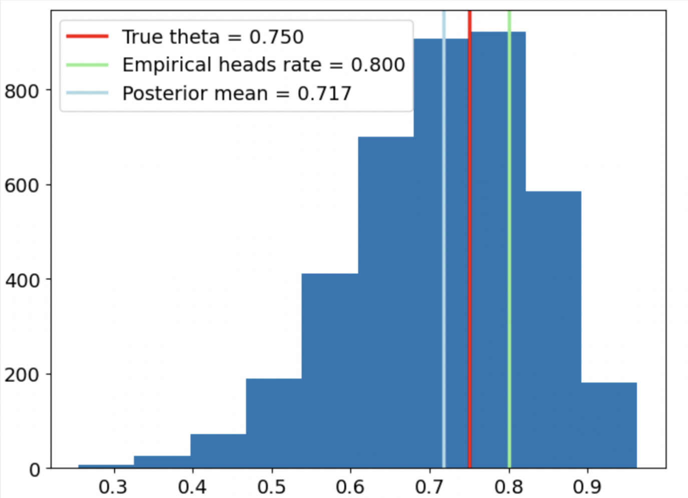
```
guide = pyro.infer.autoguide.AutoNormal(simple_model)

adam = pyro.optim.Adam({"lr": 0.02})  # Consider decre
elbo = pyro.infer.Trace_ELBO()
svi = pyro.infer.SVI(simple_model, guide, adam, elbo)

losses = []
for step in range(1000):
    loss = svi.step(y_obs)
    losses.append(loss)
    if step % 100 == 0:
        print("Elbo loss: {}".format(loss))
```

Gen (representative for programmable inference)

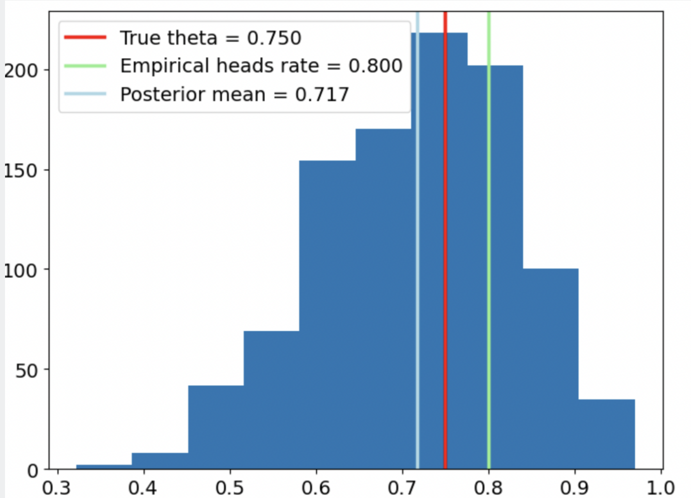
```
function my_inference_program(ys::Vector{Bool}, num_iters::Int)
    # Create a set of constraints fixing the
    # y coordinates to the observed y values
    constraints = choicemap()
    for (i, y) in enumerate(ys)
        constraints["y-$i"] = y
    end

    # Run the model, constrained by `constraints`,
    # to get an initial execution trace
    (trace, _) = generate(my_model, (ys,), constraints)

    xs = Float64[]

    # Iteratively update the slope then the intercept,
    # using Gen's metropolis_hastings operator.
    for iter=1:num_iters
        (trace, _) = metropolis_hastings(trace, select(:theta))
        push!(xs, trace[:theta])
    end

    # From the final trace, read out the slope and
    # the intercept.
    return xs
end
```

PPL Comparison - Posterior Distributions

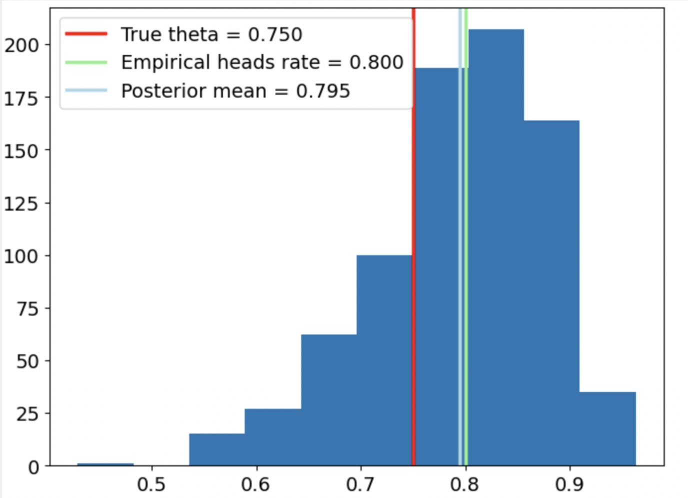
Stan



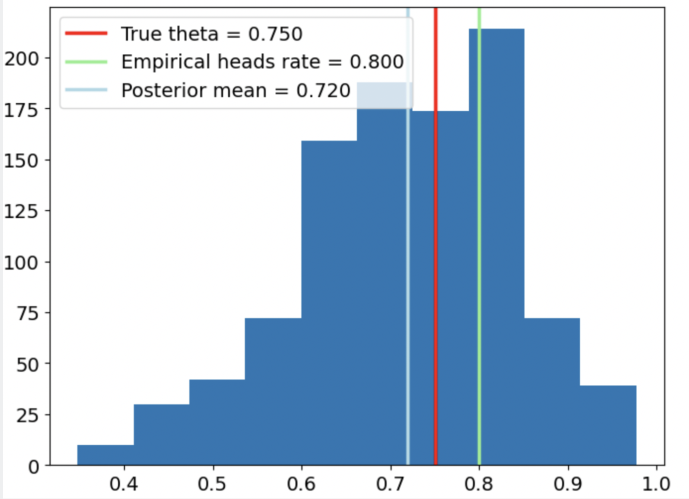
Beanmachine



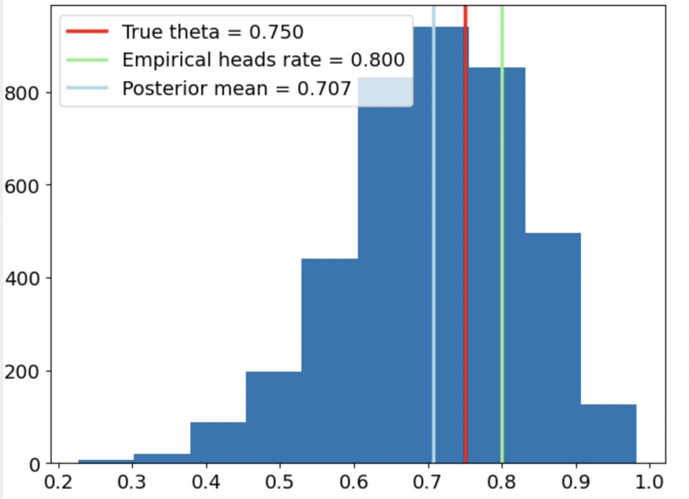
Pyro



Turing.jl



Gen



Probabilistic Programming and AI: Course Overview

Live lectures in Seminarraum FAV 01 A (Seminarraum 183/2) — **Kick-off in FAV Hörsaal 2**
>> not mandatory, but recommended

04.10. 15:00-17:00	Kick-Off & Lecture 1: • Introduction to Probabilistic Programming • Probability and Bayesian Statistics Primer **FAV Hörsaal 2**	• Bayesian Methods for Hackers Probabilistic Programming and Bayesian Inference: Chapter 1 • Seeing Theory: Chapters 1,2,3,5 • 3Blue1Brown Bayes Theorem • AI That Understands the World, Using Probabilistic Programming
04.10.	Release Assignment 1 (A1)	
11.10. 15:00-17:00	Lecture 2: • Bayesian Inference and Generative Modelling • Probabilistic Programming Languages • Implementation Designs • Minimal PPL Implementation • Independent Sampling	• Bayesian Inference Framework • Intuition behind Bayesian inference • Bayesian posterior sampling • Generative Models • A Personal Viewpoint on Probabilistic Programming
11.10.	Release Assignment 2 (A2)	
18.10. 15:00-17:00	Lecture 3: • Dependent Sampling • Markov Chain Monte Carlo • Metropolis Hastings Algorithm • Hamiltonian Monte Carlo	• Why we use dependent sampling to sample from the posterior • An introduction to the Random Walk Metropolis algorithm • Paper: Single-Site MH for PPL • Handbook of MCMC: Chapter 5: MCMC Using Hamiltonian Dynamics • The intuition behind the Hamiltonian Monte Carlo algorithm • MCMC Interactive Gallery • Paper: No-U-Turn Sampler
25.10. 15:00-17:00	Lecture 4: • Variational Inference • Automatic Differentiation VI • Stochastic VI	• KL Divergence - Clearly explained! • Variational Inference + ELBO Intuition • Automatic Differentiation and Gradient Descent • Paper: Automatic Differentiation Variational Inference • Paper: Stochastic Variational Inference

TU WIEN Informatics

Probabilistic Programming and AI: Course Overview

Live lectures in Seminarraum FAV 01 A (Seminarraum 183/2) — **Kick-off in FAV Hörsaal 2**
>> not mandatory, but recommended

04.10. 15:00-17:00	Kick-Off & Lecture 1: • Introduction to Probabilistic Programming • Probability and Bayesian Statistics Primer **FAV Hörsaal 2**	• Bayesian Methods for Hackers Probabilisti Bayesian Ir • Seeing The • 3Blue1Brov • AI That Un Probabilisti
04.10.	Release Assignment 1 (A1)	
11.10. 15:00-17:00	Lecture 2: • Bayesian Inference and Generative Modelling • Probabilistic Programming Languages • Implementation Designs • Minimal PPL Implementation • Independent Sampling	• Bayesian Ir • Intuition be • Bayesian p • Generative • A Personal Programmi
11.10.	Release Assignment 2 (A2)	

18.10. 15:00-17:00	Lecture 3: • Dependent Sampling • Markov Chain Monte Carlo • Metropolis Hastings Algorithm Monte Carlo	• Why we use dependent sampling to sample from the posterior • An introduction to the Random Walk Metropolis algorithm • Paper: Single-Site MH for PPL • Handbook of MCMC: Chapter 5: MCMC Using Hamiltonian Dynamics • The intuition behind the Hamiltonian Monte Carlo algorithm • MCMC Interactive Gallery • Paper: No-U-Turn Sampler
	ference fferentiation VI	• KL Divergence - Clearly explained! • Variational Inference + ELBO Intuition • Automatic Differentiation and Gradient Descent • Paper: Automatic Differentiation Variational Inference • Paper: Stochastic Variational Inference

06.12. 15:00-17:00	Lecture 5: • Advanced Inference • TBD
07.12.	Office Hours 13:15-15:00 (online)
13.12. 15:00-17:00	Lecture 6: • Probabilistic AI • TBD
13.12.	A4 Deadline
13.12.	Project Proposal Deadline
20.12. 15:00-17:00	Assignment Discussion Session A3 & A4
31.01. 14:00-17:00	Final Project Presentations

Probabilistic Programming and AI: Course Overview

40% — 4 Assignments with 2 assignment discussion sessions (online and **mandatory!**)
60% — Group project with final presentation

Assignments:

Assignment 1: Introduction to PPLs

Assignment 2: Minimal PPL implementation

Assignment 3: MH Inference Impl.

Assignment 4: Gradient-based Inference Impl.

Group project: You will submit project proposals (we will provide feedback on feasibility)

Initial ideas:

- Applying probabilistic programming to a non-trivial problem
- Implementing an advanced inference algorithm in our minimal PPL
- Implementing a PPL following a different design principle

Probabilistic Programming and AI:
What kind of program can represent thinking?

Structure

Knowledge

Probability

Uncertainty

```
@gen function line_model(xs::Vector{Float64})
    slope = ({:slope} ~ normal(0, 1))
    intercept = ({:intercept} ~ normal(0, 2))

    for (i, x) in enumerate(xs)
        ({(:y, i)} ~ normal(slope * x + intercept, 0.1))
    end
    return length(xs)
end;

function do_inference(model, xs, ys, amount_of_computation)
    observations = Gen.choicemap()
    for (i, y) in enumerate(ys)
        observations[(:y, i)] = y
    end

    (trace, _) = Gen.importance_resampling(model, (xs,),
                    observations, amount_of_computation);
    return trace
end;

xs = [...];
ys = [...];
sample_trace = Gen.simulate(line_model, (xs, ));
...
trace = do_inference(line_model, xs, ys, 100)
```

Slope and Intercept as Latent Variables (Priors)

Likelihood defined as model over slope and intercept with fixed noise

Observed variable samples explicitly recorded in a trace *{(:y, i)}*

Probabilistic Modeling as Generative Function (Simulator)

Observations recalled in inference step and passed to inference procedure

Approximative inference with importance resampling (can be exchanged with MCMC, HMC, NUTS, etc.)

Inference separated from modeling

You can find assignment 1 (A1) on TUWEL

Remember: Successfully completing A1 until October 17th is mandatory for your final registration