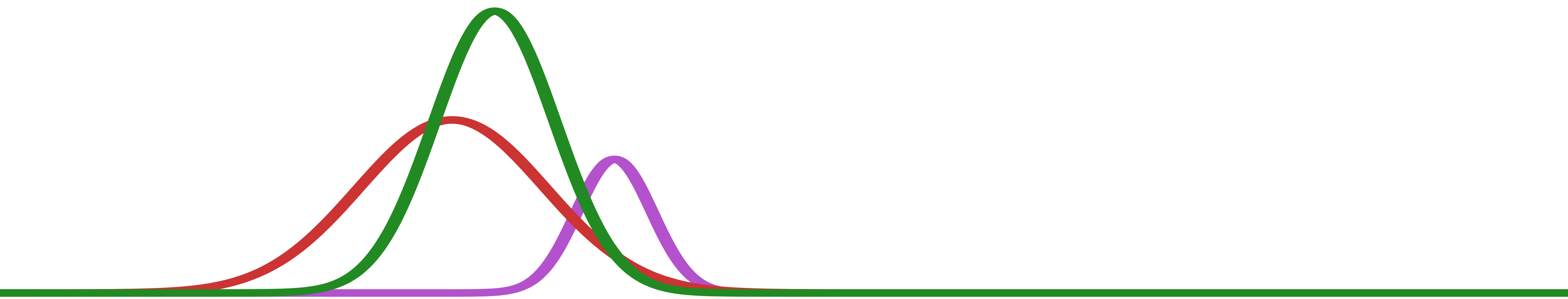
Turing.jl: The Turing language for probabilistic programming

— Martin Trapp —



visit: https://turing.ml

Why do we need probabilistic models?

- Real world problems exhibit multiple levels of uncertainty and noise in the data.
- Thus, we need tools that faithfully represent uncertainty in our model structure and parameters and can account for noise in our data.
- Those modelling tool need to be automated, adaptive and exhibit robustness.
- Further, we need to be able to scale well to large data sets while being effective in small data domains.

Why do we need probabilistic models?

In probabilistic modelling, the inverse probability (Bayes rule) allows us to infer unknown quantities, adapt our model, make predictions and learn from data.

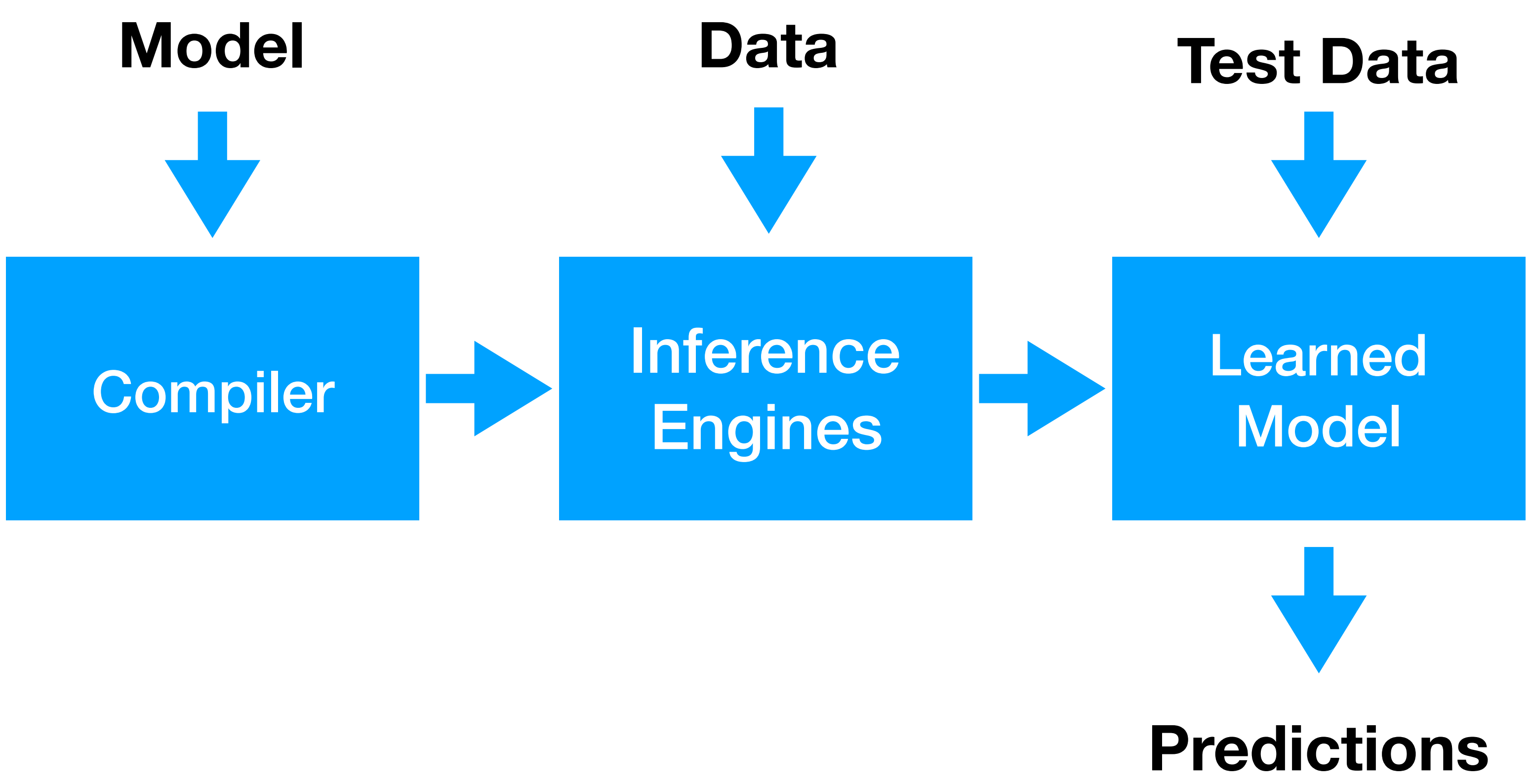
$$p(\boxed{\theta} \mid \boxed{\mathcal{D}}) = \frac{p(\mathcal{D} \mid \theta)p(\theta)}{p(\mathcal{D})}$$

Bayes rule tells us how to do inference about **hypothesis** from **data**. Learning and prediction in the Bayesian framework can be seen as a forms of inference.

Probabilistic Programming

- Probabilistic programs: computer programs represent probabilistic models with probabilistic statements:
  - Declaring *random variables*
  - Conditioning on observed data
- Universal probabilistic programming
  - Stochastic control flows
  - Allows representing arbitrary probabilistic models
- Generic inference engines: HMC, SMC, particle Gibbs, …
- Two approaches to implement a PPL
  - Standalone: Stan, BUGS, Venture, etc
  - Embedded: Anglican, infer.NET, PyMC3, Pyro, Edward, **Turing**, etc

Probabilistic Programming



Workflow & Components of Turing

Turing — Example

```
@model gdemo(x) = begin
  s ~ InverseGamma(2, 3)
  m ~ Normal(0, sqrt(s))
  for i = 1:length(x)
    x[i] ~ Normal(m, sqrt(s))
  end
  return s, m
end
```

Simple Gaussian Model in Turing

Turing — Example

2 `gdemo(x)` defines a Julia function.

1 `@model` translates a normal Julia program into a Turing model.

3 Observations are declared as the parameters in the function definition.

4 When the left hand side of `~` is not declared as an observation, the statement defines a random variable.

5 When the left hand side of `~` is an observation, it performs conditioning.

```
@model gdemo(x) = begin
    s ~ InverseGamma(2, 3)
    m ~ Normal(0, sqrt(s))
    for i = 1:length(x)
        x[i] ~ Normal(m, sqrt(s))
    end
    return s, m
end
```

Everything else follows standard Julia syntax.

Illustration of Turing's Syntax

Inference Algorithms

- Turing provides a range of inference algorithms that can be used, combined and extended.
- Simulation-based MCMC inference
    - Metropolis Hastings, Sequential Monte Carlo, Particle Gibbs, …
- Gradient-based MCMC inference (AdvancedHMC.jl)
    - Hamiltonian Monte Carlo, No-U-Turn Sampler, Stochastic Gradient Langevin Dynamics, …
- Compositional MCMC inference
- Variational Inference (work-in progress)
    - Automatic Differentiation Variational Inference, Structured Variational Inference, …

Inference in Turing

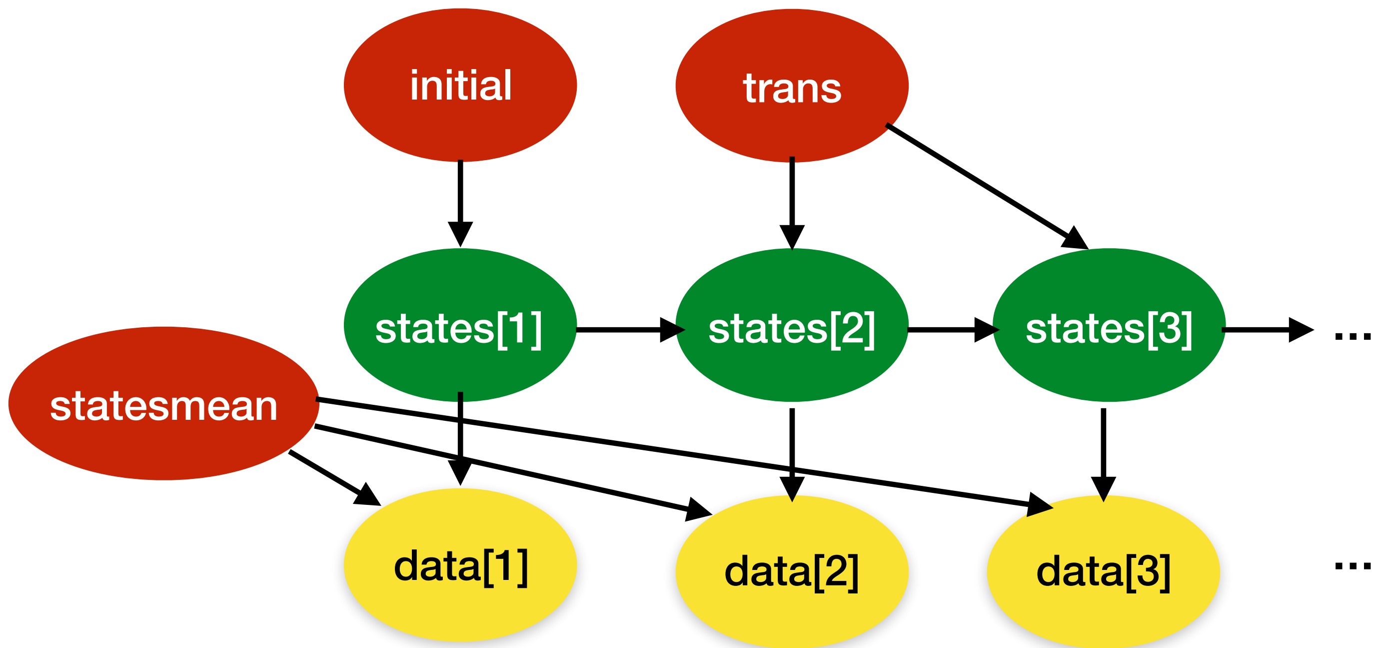
```julia
@model gdemo(x) =  begin
  s ~ InverseGamma(2,3)
  m ~ Normal(0,sqrt(s))
  for i=1:length(x)
    x[i] ~ Normal(m, sqrt(s))
  end
  return(s, m)
end
```

```julia
mf = gdemo([1.5, 2])
alg = HMC(2000, 0.1, 10)
chain = sample(mf, alg)
```

1. By passing data to a compiled model, we get a generated Turing model.
2. An inference algorithm is defined by its name and corresponding parameters.
3. The `sample` function takes a generated model and a sampling algorithm to perform inference.
4. The returned value `chain` stores MCMC samples.

Compositional Inference

- Combine simulation and gradient-based inference
- Generic universal engine



Sampling for Bayesian hidden Markov model:

- Sample states using particle Gibbs
- Sample initial, trans and statesmean using Hamiltonian Monte Carlo

Compositional Inference in Turing

```
# Sampler = HamiltonianMonteCarlo + ParticleGibbs
g1 = Gibbs(500, HMC(1, 0.2, 3, :m), PG(50, 1, :s))
```

1 Gibbs is defined by number of iterations and multiple sampling algorithms as its components.

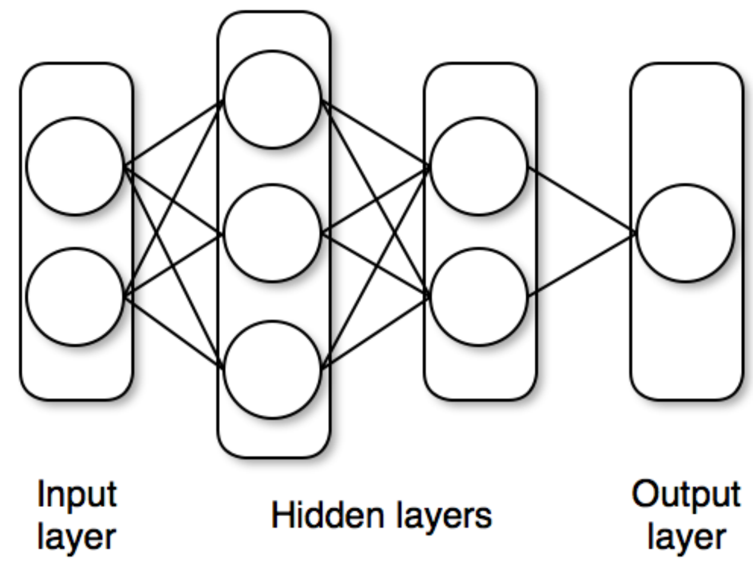
2 HMC is specified to sample variable m.

3 PG is specified to sample variable s.

Integration of Deep Learning

```julia
function nn_forward(x, theta::AbstractVector)
    W₁, b₁, W₂, b₂, Wₒ, bₒ = unpack(theta)
    nn = Chain(Dense(W₁, b₁, tanh),
               Dense(W₂, b₂, tanh),
               Dense(Wₒ, bₒ, σ))
    return nn(x)
end
```

Flux.jl



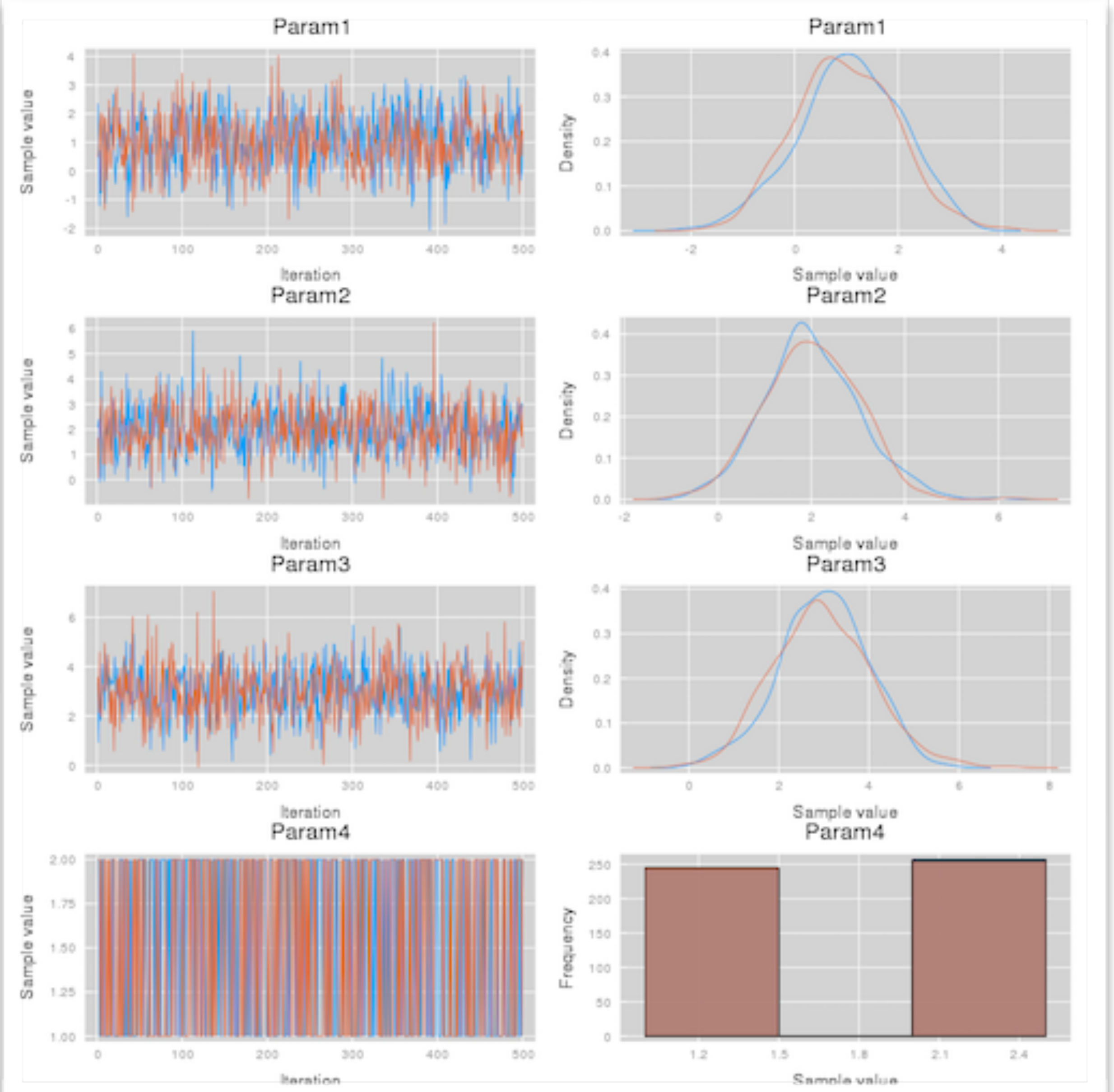

```julia
alpha = 0.09              # regularizatin term
sig = sqrt(1.0 / alpha) # variance of the Gaussian prior

@model bayes_nn(xs, ts) = begin
    theta ~ MvNormal(zeros(20), sig .* ones(20))

    preds = nn_forward(xs, theta)
    for i = 1:length(ts)
        ts[i] ~ Bernoulli(preds[i])
    end
end
```

Turing.jl

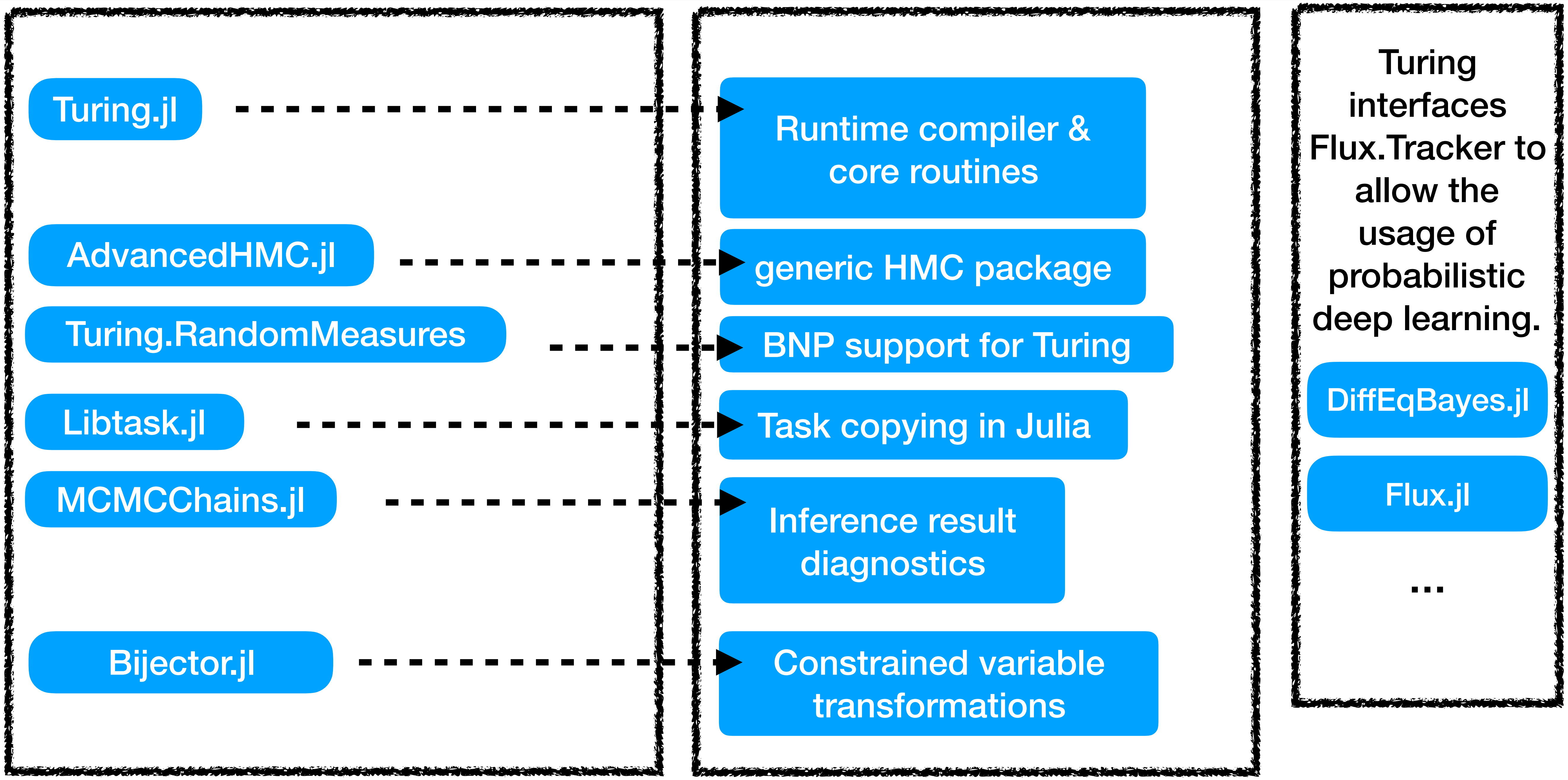
Analysing Results

MCMCChains.jl from TuringLang provides a range of utility functions for diagnostics and visualizations.

- Convergence diagnostics:
  - Gelman, Rubin, and Brooks
  - Geweke
  - Heidelberger and Welch
  - Raftery and Lewis
- Visualisations:
  - Trace plot, running average plot, density and histogram plot, autocorrelation plot and corner plot.
  - MCMCChains uses RecipesBase and StatsPlots for visualisations.



TuringLang



Turing.jl
Runtime compiler & core routines
AdvancedHMC.jl
generic HMC package
Turing.RandomMeasures
BNP support for Turing
Libtask.jl
Task copying in Julia
MCMCChains.jl
Inference result diagnostics
Bijector.jl
Constrained variable transformations
Turing interfaces Flux.Tracker to allow the usage of probabilistic deep learning.
DiffEqBayes.jl
Flux.jl
...

Logistic Regression Example

Contributors

Hong Ge
@yebai

Kai Xu
@xukai92

Will Tebbutt
@willtebbutt

Mohamed Tarek
@mohamed82008

Cameron Pfiffer
@cpfiffer

Martin Trapp
@trappmartin

Wessel Bruinsma
@wesselb

Emile Mathieu
@emilemathieu

Zoubin Ghahramani

many more …

Thanks!