簡単にturing.jlを紹介する

Turign.jl is 何?

公式曰く...

> Turing.jl is a Julia library for general-purpose probabilistic programming. Turing allows the user to write models using standard Julia syntax, and provides a wide range of sampling-based inference methods for solving problems across probabilistic machine learning, Bayesian statistics, and data science.

要は...

- `stan` などのような確率的プログラミング言語(というよりライブラリ)
- 機械学習からベイズ統計まで幅広く使える
- `Julia` の構文を**そのまま使える**(<= 今回のメインテーマ)

どんな感じかみてみよう(公式ドキュメントより)

```
@model function infiniteGMM(x)
    α = 1.0
    μ0 = 0.0
    σ0 = 1.0

    rpm = DirichletProcess(α)
    H = Normal(μ0, σ0)
    z = tzeros(Int, length(x))
    μ = tzeros(Float64, 0)

    for i in 1:length(x)
        K = maximum(z)
        nk = Vector{Int}(map(k -> sum(z .== k), 1:K))
        z[i] ~ ChineseRestaurantProcess(rpm, nk)
        if z[i] > K
            push!(μ, 0.0)
            μ[z[i]] ~ H
        end
        x[i] ~ Normal(μ[z[i]], 1.0)
    end
end
```

無限混合ガウスモデルも簡単に書けちゃうね

なんとinfinite GMMがたったの19行!! ぴょえぇ〜〜〜

Turing.jlの強み

- Infinite GMMが簡単に実装できるように離散パラメータを容易に扱える
- `Julia`の構文が使えるから`stan`のように言語の勉強が不要
- 上の理由でシミュレーションと`turing.jl`のモデルでコードを共有できる

Turing.jlの弱み

- コミュニティが `stan` などと比べて小さい
- サンプルが機械学習寄りが多く統計モデルが少なめ
- エラーメッセージが親切ではない(コミュニティが小さいからググっても出てこないことが多い)

とはいえ書きやすいので上の問題はあまり大きくはない気もする

→ とりあえず `turing.jl` やってみろ

実際にQ-learningを実装してみる

1. 最低限の `julia` の知識のおさらい
2. 念の為Q-learningの簡単な説明
3. シミュレーション用プログラムの解説
4. `Turing.jl` による実装

Juliaのおさらい

今回使うものだけを簡単におさらい

関数宣言

- `function ~ end`で関数宣言ができる
- 引数の型は`::Type`で指定できる
- 型宣言をしない場合はジェネリクスとなる

```
function f(x, y)
    x + y
end

function f(x::Real, y::Real)
    x + y
end
```

多重ディスパッチ(Multiple dispatch)

- `Julia`では同名の関数でも引数の型が異なれば異なる実装が可能
- 複数の型によって呼び出されるメソッドが決定される

```
function add(x::Number, y::Number)
    x + y
end

function add(x::String, y::String)
    "$x$y"
end
```

上の例にはないが`add(x::String, y::Number)`も可能なのが多重ディスパッチの特徴

関数呼び出し

当たり前だが`f(args)`で関数呼び出しが可能

```
add(1, 2)
```

ブロードキャスト

`.`を使用することで引数がスカラーであっても配列に適用可能になる

```
add([1, 2, 3], 2) # Error
add.([1, 2, 3], 2)
```

`.`は演算子にも使える

```
1 - [1, 2, 3] # Error
1 .- [1, 2, 3]
```

構造体

- `struct ~ end`で構造体を定義できる
- フィールドへのアクセスは`instance.fieldname`で可能

```
struct Hoge
    foo
    bar
end

hoge = Hoge(1, 2)
hoge.foo
hoge.bar
```

- `::Type`でフィールドの型を指定することができる
- 関数と同様に型宣言をしない場合はジェネリクス

```
struct HogeWithType
    foo::Real
    bar::Real
end
```

- 構造体はデフォルトでイミュータブルなのでミュータブルしたい場合は`mutable`を使用する

```
mutable struct MutableHoge
    foo::Real
    bar::Real
end

hoge = Hoge(1, 2)
hoge.foo = 3 # Error
mutable_hoge = MutableHoge(1, 2)
mutable_hoge.foo = 3
```

コンストラクタ

`Python`でいうところの`__init__`にあたる処理

```
struct HogeWithConstructor
    foo::Real
    bar::Real
    HogeWithConstructor(foo) = new(foo, foo * 2)
end

HogeWithConstructor(2)
```

`function`でも可能

```
struct HogeWithConstructor
    foo::Real
    bar::Real
    function HogeWithConstructor(foo)
        new(foo, foo * 2)
    end
end
```

for

見れば分かるよね

```
for i in 1:100
    println(i)
end
```

ちなみに配列に対する関数適用に`.`を使うのは便利だが`for`で書いた方が早い

```
function add_array(x::Array{Real, 1}, y::Real)
    ret = Array{Real, 1}(undef, length(x))
    for x_ in x
        x_ + y
    end
end

x = Array{Real, 1}(0.:0.1:100.)
@time add.(x, 2)
@time add_array(x, 2)
```

Q-learningを実装する: おさらい編

今回は下の式を実装していく

行動価値関数は以下の式に従って更新する

$$Q(s_t, a) \leftarrow Q(s_t, a) + \alpha \cdot [r_{t+1} + \gamma \max_p Q(s_{t+1}, p) - Q(s_t, a)]$$

ちなみにバンディットのような状態遷移を伴わない環境であれば

$$Q(s_t, a) \leftarrow Q(s_t, a) + \alpha \cdot [r_{t+1} - Q(s_t, a)]$$

方策をソフトマックス関数とすると

$$\pi(s_t, a) = \frac{\exp(Q(s_t, a) \cdot \beta)}{\sum_{p \in A} \exp(Q(s_t, p) \cdot \beta)}$$

Q-learningを実装する: エージェントの実装

まずは`struct`でエージェントの構造体を定義する
Q値を更新するため`mutable`で宣言する必要がある

```
mutable struct QLearningAgent
    q::Array{Real, 1}
    α::Real
    β::Real
    function QLearningAgent(α::Real, β::Real, K::Int64)
        q = Array{Real, 1}(zeros(K))
        new(q, α, β)
    end
end
```

Q-learningを実装する: エージェントのメソッドの実装

```
function update(agent::QLearningAgent, action::Real, reward::Real)
    agent.q[action] += agent.α * (reward - agent.q[action])
end

function policy(agent::QLearningAgent)
    qmax = maximum(agent.q)
    qexp = exp.(agent.β .* (agent.q .- qmax))
    return qexp ./ sum(qexp)
end
```

`exp`は入力が大きいとオーバーフローを起こすので最大値からの差分を取っている

今回のモデルではQ値は0. - 1.の範囲しか取らないため必ずしも必要ではない

Q値は配列であるため`.`を使用している

Q-learningを実装する: バンディットの実装

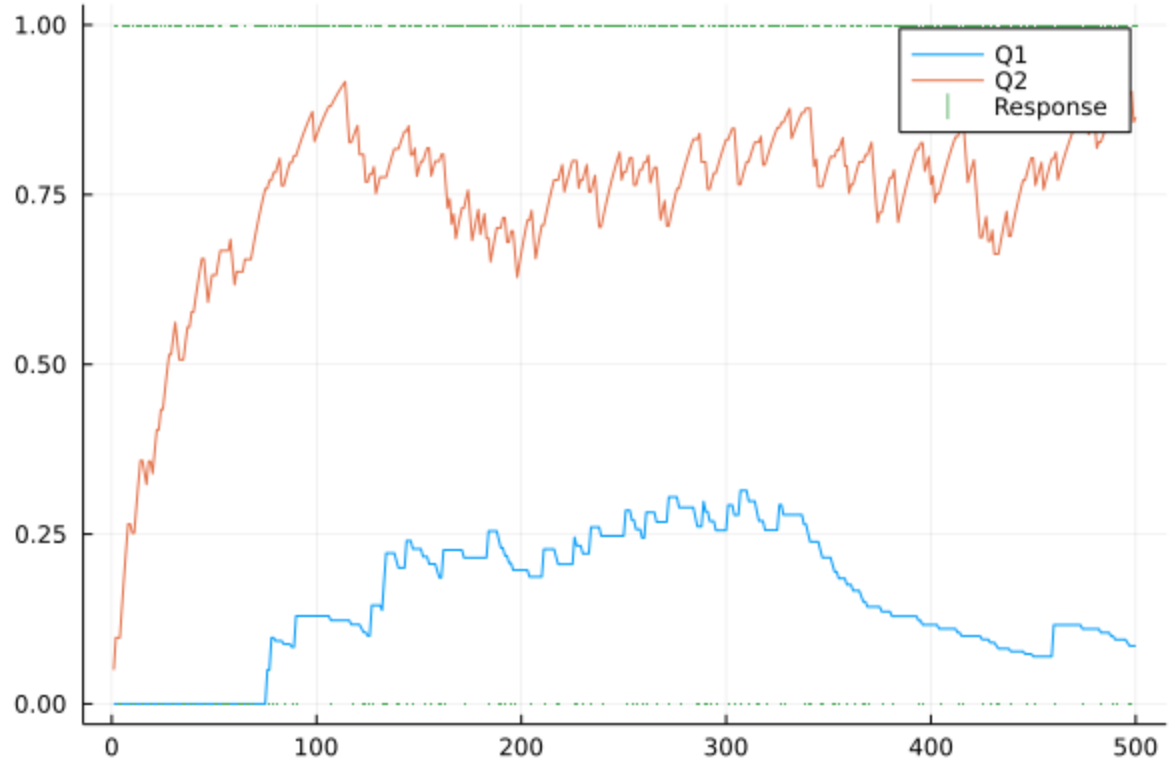
```
struct Bandit
    p::Array{Real, 1}
end

function step(env::Bandit, action::Int64)
    k = length(env.p)
    rewards = env.p .>= rand(k)
    return rewards[action]
end
```

Q-learningを実装する: シミュレーションの実装

```
using Distributions, DataFrames


function step(agent::QLearningAgent, env::Bandit)
    p = policy(agent)
    action = rand(Categorical(p))
    reward = step(env, action)
    update(agent, action, reward)
    d = DataFrame(action = action, reward = reward)
    return hcat(d, DataFrame(agent.q', :auto))
end

function run(agent::QLearningAgent, env::Bandit, trial::Int64)
    return reduce(vcat, map(_ -> step(agent, env), 1:trial))
end
```

Q-learningを実装する: 実際にシミュレーション

```
using Plots


bandit = Bandit([0.2, 0.8])
k = length(bandit.p)
agent = QLearningAgent(0.05, 2., k)
result = run(agent, bandit, 500)

plot(result.x1)
plot!(result.x2)
scatter!(result.action .- 1,
         markersize=1,
         markershape=:vline)
```



まあ大体こんなもんだよね

Q-learningを実装する: Turing.jlで書いてみる

`@model Model() = begin ~ end` でモデルを定義する

サンプリングしたいパラメータについて `param ~ Dist()` で宣言する

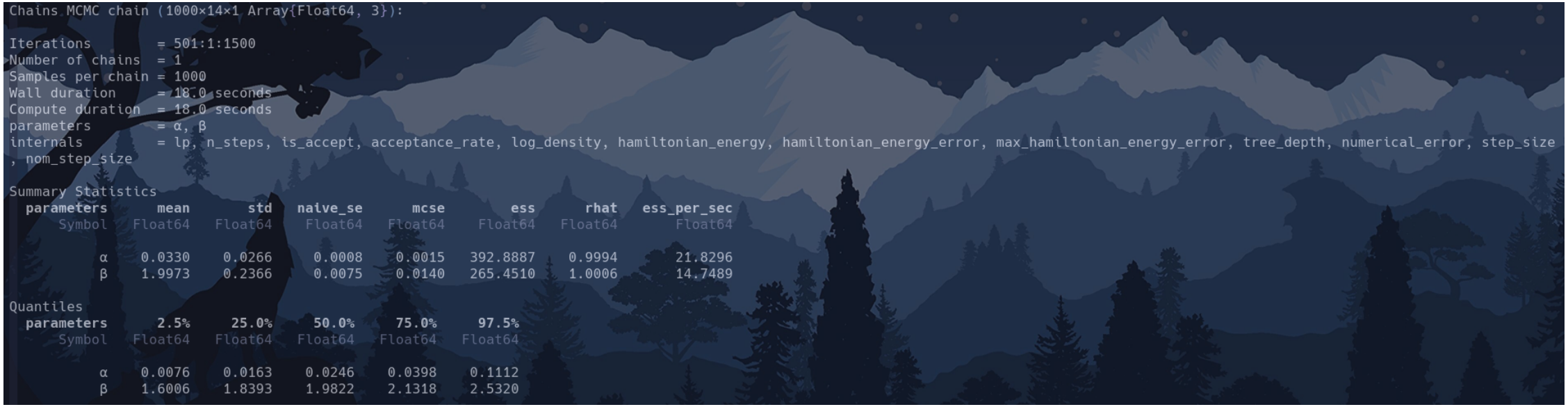
```
using Turing


@model QLearningModel(actions::Array{Real, 1}, rewards::Array{Real, 1}, k::Int64) = begin
    T = length(actions)
    α ~ Beta(1, 1)
    β ~ Gamma(1, 100)
    agent = QLearningAgent(α, β, k)
    for t in 1:T
        action = actions[t]
        reward = rewards[t]
        p = policy(agent)
        actions[t] ~ Categorical(p)
        update(agent, action, reward)
    end
end
```

Let's MCMC

- `sample(Model(args), Sampler(), numiter)` でサンプリングできる
- `sample` の結果を `plot` に渡すことでトレースプロットが表示できる
  - `Turing.jl` の結果を `plot` するために `StatsPlots` が必要

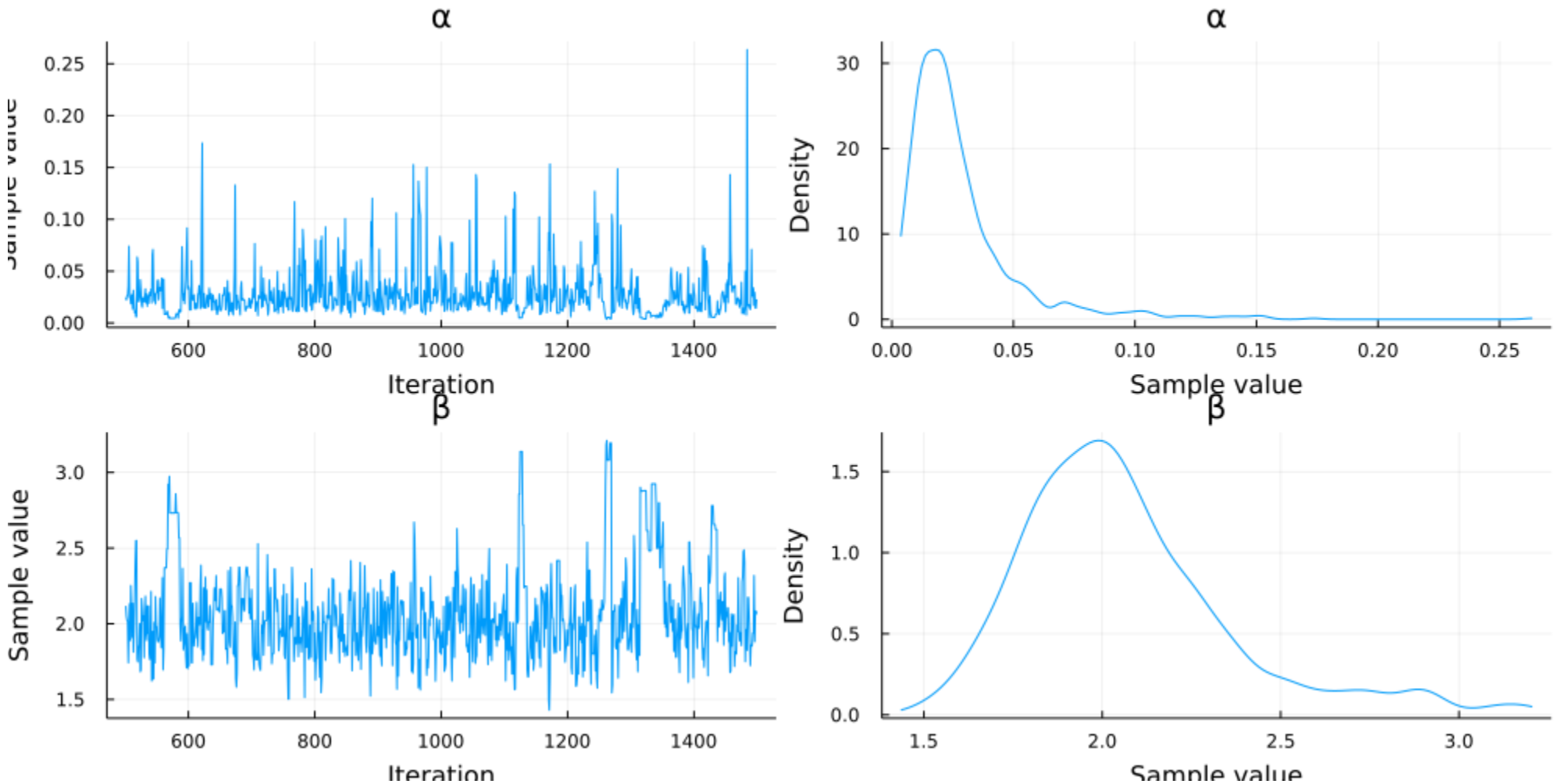
```
using StatsPlots


actions = Array{Real, 1}(result.action)
rewards = Array{Real, 1}(result.reward)
chains = sample(QLearningModel(actions, rewards, 2), NUTS(), 1000)
plot(chains)
```

MCMCが終わると結果が表示される



```
Chains MCMC chain (1000×14×1 Array{Float64, 3}):

Iterations        = 501:1:1500
Number of chains  = 1
Samples per chain = 1000
Wall duration     = 18.0 seconds
Compute duration  = 18.0 seconds
parameters        = α, β
internals         = lp, n_steps, is_accept, acceptance_rate, log_density, hamiltonian_energy, hamiltonian_energy_error, max_hamiltonian_energy_error, tree_depth, numerical_error, step_size
, nom_step_size

Summary Statistics
  parameters      mean       std   naive_se      mcse         ess      rhat   ess_per_sec
      Symbol   Float64   Float64    Float64   Float64     Float64   Float64       Float64

           α    0.0330    0.0266     0.0008    0.0015   392.8887    0.9994       21.8296
           β    1.9973    0.2366     0.0075    0.0140   265.4510    1.0006       14.7489

Quantiles
  parameters      2.5%     25.0%     50.0%     75.0%     97.5%
      Symbol   Float64   Float64   Float64   Float64   Float64

           α    0.0076    0.0163    0.0246    0.0398    0.1112
           β    1.6006    1.8393    1.9822    2.1318    2.5320
```

`plot(MCMCChains)` でトレースプロットが表示される



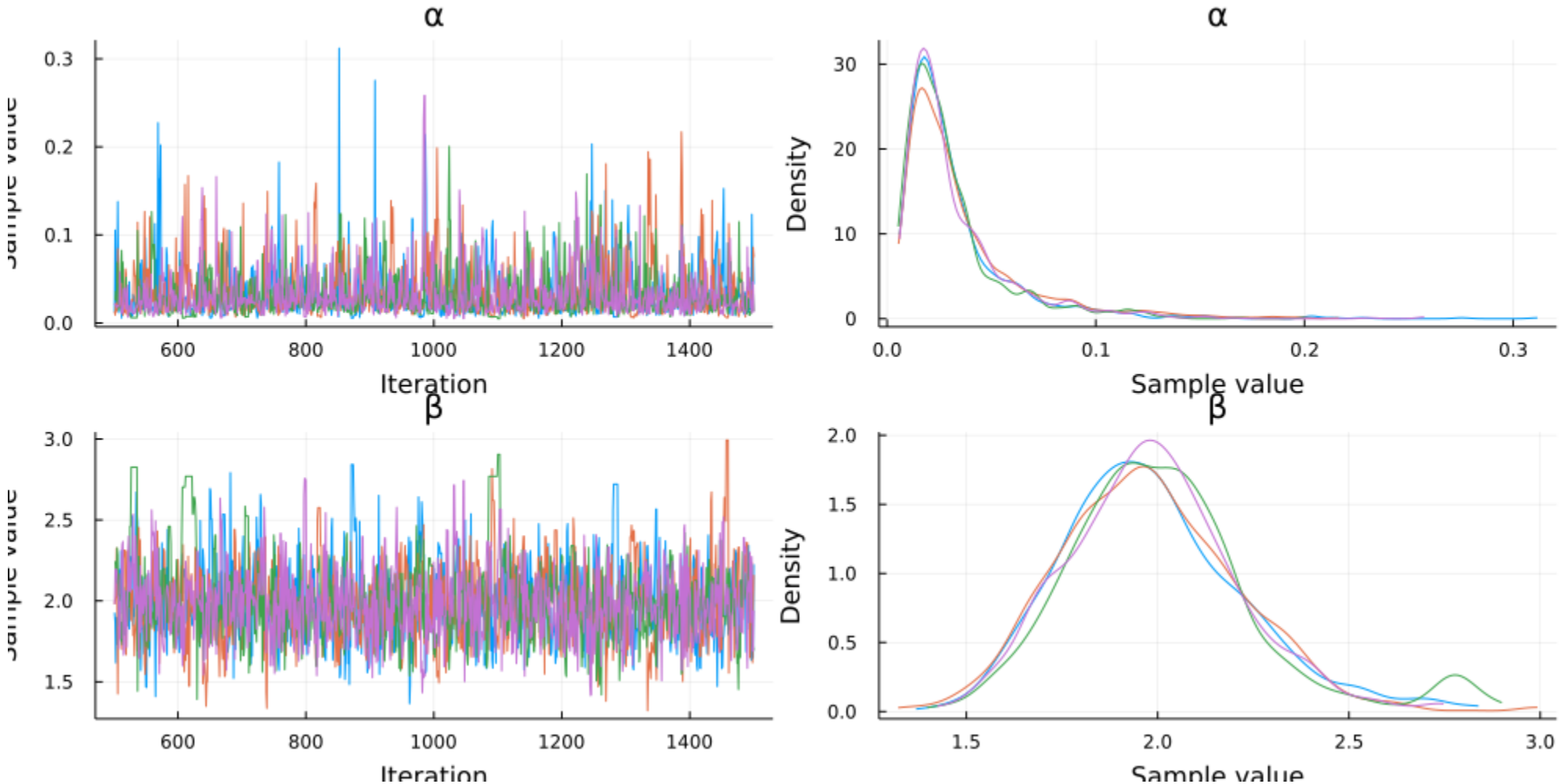
複数のチェインを回す

- `sample`の引数に`MCMCThreads()`とチェイン数を追加するだけ

```
using MCMCChains


chains = sample(QLearningModel(actions, rewards, 2), NUTS(), MCMCThreads(), 1000, 4)
plot(chains)
```

環境変数の`JULIA_NUM_THREADS`以上のチェイン数は並立化されない

あらかじめ`JULIA_NUM_THREADS`の値を4以上にしておくと良い


α
Sample value
Iteration
α
Density
Sample value
β
Sample value
Iteration
β
Density
Sample value

Turing.jlを触った所感

- `Julia`の構文を使えるため圧倒的に楽にモデルが書ける
- シミュレーションコードの使い回しができるのが強い
- モデルのコーディング量が圧倒的に少なくて済む
- `Stan`で書けないモデル(少なくとも自分には)も簡単に書ける

結論: みんな`turing.jl`使えばいいのに