```julia
begin
    import Pkg
    # activate the shared project environment
    Pkg.activate(Base.current_project())
    using Omega, Distributions, UnicodePlots, OmegaExamples
end
```

```
Activating project at `~/Documents/GitHub/Omega.jl/OmegaExamples`
```

Prologue: The performance characteristics of different algorithms

There are many different ways to sample from the same distribution, it is thus useful to separately think about the distributions we are building (including conditional distributions) and how we will sample from them. Indeed, in the last few chapters we have explored the dynamics of inference without worrying about the details of inference algorithms. The efficiency characteristics of different implementations of `randsample` can be very different, however, and this is important both practically and for motivating cognitive hypotheses at the level of algorithms (or psychological processes).

The "guess and check" method of rejection sampling is conceptually useful but is often not efficient: even if we are sure that our model can satisfy the condition, it will often take a very large number of samples to find computations that do so. To see this, let us explore the impact of `baserate` in our simple warm-up example:

```
baserate = 0.1
```

```
baserate = 0.1
```

```
(value = [true, true, true, true, true, true, true, true, true,    more ,false], time = 86.249
```

```
@timed let
        A = @~ Bernoulli(baserate)
        B = @~ Bernoulli(baserate)
        C = @~ Bernoulli(baserate)
        randsample(A |ᶜ (A .+ B .+ C .>= 2), 100)
    end
```

Even for this simple program, lowering the baserate by just one order of magnitude, to 0.01, will make rejection sampling impractical.

There are many other algorithms and techniques for probabilistic inference, reviewed below. They each have their own performance characteristics. For instance, *Markov chain Monte Carlo* inference approximates the posterior distribution via a random walk (described in detail below).

```
 (
    value =   [false, false, false, false, true, true, false, true, true,    more ,false]
    time = 0.0922519
    bytes = 9816176
    gctime = 0.0
    gcstats =   GC_Diff(9816176, 0, 0, 144140, 38, 0, 0, 0, 0)
)
```

```
@timed let
        A = @~ Bernoulli(baserate)
        B = @~ Bernoulli(baserate)
        C = @~ Bernoulli(baserate)
        randsample(A |ᶜ (A .+ B .+ C .>= 2), 100, alg = MH)
    end
```

See what happens in the above inference as you lower the baserate. Unlike rejection sampling, inference will not slow down appreciably (but results will become less stable). Inference should also not slow down exponentially as the size of the state space is increased. This is an example of the kind of trade-offs that are common between different inference algorithms.

The varying performance characteristics of different algorithms for (approximate) inference mean that getting accurate results for complex models can depend on choosing the right algorithm (with the right parameters). In what follows we aim to gain some intuition for how and when algorithms work, without being exhaustive.

Markov chain Monte Carlo (MCMC)

We have already seen that samples from a (conditional) distribution can be an effective way to represent the results of inference – when rejection sampling is feasible it is an excellent approach. Other methods have been developed to take *approximate* samples from a conditional distribution. One popular method uses Markov chains.

Markov chains as samplers

A Markov model (or *Markov chain*, as it is often called in the context of inference algorithms)is a stochastic (i.e., random) process that transitions between states. Here is a Markov chain:

```
transition_probs =
 [[0.48, 0.48, 0.02, 0.02], [0.48, 0.48, 0.02, 0.02], [0.02, 0.02, 0.48, 0.48], [0.02, 0.02, 0.
```

```
transition_probs = [[.48, .48, .02, .02],
    [.48, .48, .02, .02],
    [.02, .02, .48, .48],
    [.02, .02, .48, .48]]
```

```
transition (generic function with 1 method)
```

```
transition(ω, i, state) = (i ~ Categorical(transition_probs[state]))(ω)
```

```
chain (generic function with 1 method)
```

```
chain(state, n, ω) = (n == 0) ? state : chain(transition(ω, n, state), n - 1, ω)
```

State after 10 steps:

```
              ┌                                        ┐
 [1.0, 1.5) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 360
 [1.5, 2.0) ┤ 0
 [2.0, 2.5) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 343
 [2.5, 3.0) ┤ 0
 [3.0, 3.5) ┤■■■■■■■■■■■■■■ 142
 [3.5, 4.0) ┤ 0
 [4.0, 4.5) ┤■■■■■■■■■■■■■■■ 155
              └                                        ┘
                             Frequency
```

```
viz(randsample(ω -> chain(1, 10, ω), 1000))
```

```
              ┌                                        ┐
 [1.0, 1.5) ┤■■■■■■■■■■■■■ 139
 [1.5, 2.0) ┤ 0
 [2.0, 2.5) ┤■■■■■■■■■■■■■ 143
 [2.5, 3.0) ┤ 0
 [3.0, 3.5) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 339
 [3.5, 4.0) ┤ 0
 [4.0, 4.5) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 379
              └                                        ┘
                             Frequency
```

```
viz(randsample(ω -> chain(3, 10, ω), 1000))
```

State after 25 steps:

```
           ┌                                        ┐
[1.0, 1.5) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 292
[1.5, 2.0) ┤ 0
[2.0, 2.5) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 259
[2.5, 3.0) ┤ 0
[3.0, 3.5) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 231
[3.5, 4.0) ┤ 0
[4.0, 4.5) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■ 218
           └                                        ┘
                            Frequency
```

```julia
viz(randsample(ω -> chain(1, 25, ω), 1000))
```

```
           ┌                                        ┐
[1.0, 1.5) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 246
[1.5, 2.0) ┤ 0
[2.0, 2.5) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 238
[2.5, 3.0) ┤ 0
[3.0, 3.5) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 255
[3.5, 4.0) ┤ 0
[4.0, 4.5) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 261
           └                                        ┘
                            Frequency
```

```julia
viz(randsample(ω -> chain(3, 25, ω), 1000))
```

State after 50 steps:

```
           ┌                                        ┐
[1.0, 1.5) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 251
[1.5, 2.0) ┤ 0
[2.0, 2.5) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 230
[2.5, 3.0) ┤ 0
[3.0, 3.5) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 257
[3.5, 4.0) ┤ 0
[4.0, 4.5) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 262
           └                                        ┘
                            Frequency
```

```julia
viz(randsample(ω -> chain(1, 50, ω), 1000))
```

```
           ┌                                        ┐
[1.0, 1.5) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 260
[1.5, 2.0) ┤ 0
[2.0, 2.5) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 253
[2.5, 3.0) ┤ 0
[3.0, 3.5) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 234
[3.5, 4.0) ┤ 0
[4.0, 4.5) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 253
           └                                        ┘
                            Frequency
```

```julia
viz(randsample(ω -> chain(3, 50, ω), 1000))
```

Notice that the distribution of states after only a few steps is highly influenced by the starting state. In the long run the distribution looks the same from any starting state: this long-run distribution is the called the *stable distribution* (also known as *stationary distribution*). To define stationary distribution formally, let $p(x)$ be the target distribution, and let $\pi(x \to x\prime)$ be the transition distribution (i.e. the transition function in the above program). Since the stationary distribution is characterized by not changing when the transition is applied we have a *balance condition*: $p(x\prime) = \sum_x p(x)\pi(x \to x\prime)$. Note that the balance condition holds for the distribution as a whole—a single state can of course be moved by the transition.

For the chain above, the stable distribution is uniform—we have found a (fairly baroque!) way to sample from the uniform distribution on the states! We could have sampled from the uniform distribution using other Markov chains. For instance the following chain is more natural, since it transitions uniformly:

```
transition_ (generic function with 1 method)
```

```julia
transition_(ω, i, state) = (i ~ Categorical([0.25, 0.25, 0.25, 0.25]))(ω)
```

```
chain_ (generic function with 1 method)
```

```julia
chain_(state, n, ω) = (n == 0) ? state : chain_(transition_(ω, n, state), n - 1, ω)
```

State after 10 steps:

```
              ┌                                        ┐
   [1.0, 1.5) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 253
   [1.5, 2.0) ┤ 0
   [2.0, 2.5) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 262
   [2.5, 3.0) ┤ 0
   [3.0, 3.5) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 246
   [3.5, 4.0) ┤ 0
   [4.0, 4.5) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 239
              └                                        ┘
                              Frequency
```

```julia
viz(randsample(ω -> chain_(1, 10, ω), 1000))
```

```
              ┌                                        ┐
   [1.0, 1.5) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 265
   [1.5, 2.0) ┤ 0
   [2.0, 2.5) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 243
   [2.5, 3.0) ┤ 0
   [3.0, 3.5) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 263
   [3.5, 4.0) ┤ 0
   [4.0, 4.5) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 229
              └                                        ┘
                              Frequency
```

```julia
viz(randsample(ω -> chain_(3, 10, ω), 1000))
```

State after 25 steps:

```
              ┌                                        ┐
   [1.0, 1.5) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 251
   [1.5, 2.0) ┤ 0
   [2.0, 2.5) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 265
   [2.5, 3.0) ┤ 0
   [3.0, 3.5) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 249
   [3.5, 4.0) ┤ 0
   [4.0, 4.5) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 235
              └                                        ┘
                              Frequency
```

```julia
viz(randsample(ω -> chain_(1, 25, ω), 1000))
```

```
             ┌                                        ┐
  [1.0, 1.5) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 243
  [1.5, 2.0) ┤ 0
  [2.0, 2.5) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 256
  [2.5, 3.0) ┤ 0
  [3.0, 3.5) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 244
  [3.5, 4.0) ┤ 0
  [4.0, 4.5) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 257
             └                                        ┘
                            Frequency
```

```
viz(randsample(ω -> chain_(3, 25, ω), 1000))
```

State after 50 steps:

```
             ┌                                        ┐
  [1.0, 1.5) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 243
  [1.5, 2.0) ┤ 0
  [2.0, 2.5) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 262
  [2.5, 3.0) ┤ 0
  [3.0, 3.5) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 256
  [3.5, 4.0) ┤ 0
  [4.0, 4.5) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 239
             └                                        ┘
                            Frequency
```

```
viz(randsample(ω -> chain_(1, 50, ω), 1000))
```

```
             ┌                                        ┐
  [1.0, 1.5) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 242
  [1.5, 2.0) ┤ 0
  [2.0, 2.5) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 262
  [2.5, 3.0) ┤ 0
  [3.0, 3.5) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 244
  [3.5, 4.0) ┤ 0
  [4.0, 4.5) ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 252
             └                                        ┘
                            Frequency
```

```
viz(randsample(ω -> chain_(3, 50, ω), 1000))
```

Notice that this chain converges much more quickly to the uniform distribution. (Edit the code to confirm to yourself that the chain converges to the stationary distribution after a single step.) The number of steps it takes for the distribution on states to reach the stable distribution (and hence lose traces of the starting state) is called the *burn-in time*. Thus, while we can use a Markov chain as a way to (approximately) sample from its stable distribution, the efficiency depends on burn-in time. While many Markov chains have the same stable distribution they can have very different burn-in times, and hence different efficiency.

The state space in our examples above involved a small number of states, but Markov chains can also be constructed over infinite state spaces. Here's a chain over the integers:

```
p = 0.7
```

```
p = 0.7
```

```
transition_int (generic function with 1 method)
```

```julia
function transition_int(ω, i, state, p)
    if state == 3
        return (i ~ Categorical([1 - 0.5 * (1 - p), 0.5 * (1 - p)]))(ω) + 2
    end
    return (i ~ Categorical([0.5, 0.5 - 0.5 * (1 - p), 0.5 * (1 - p)]))(ω) + state - 2
end
```

```
chain_int (generic function with 1 method)
```

```julia
chain_int(state, n, p, ω) = (n == 0) ? state :
    chain_int(transition_int(ω, n, state, p), n - 1, p, ω)
```

```
      ┌                                         ┐
    3 ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 0.6998
    4 ┤■■■■■■■■■■ 0.2176
    5 ┤■■■ 0.0574
    6 ┤■ 0.0168
    7 ┤ 0.0058
    8 ┤ 0.0024
    9 ┤ 0.0002
   10 ┤ 0
      └                                         ┘
```

```julia
let
    samples = randsample(ω -> chain_int(3, 250, p, ω), 5000)
    counts = map(i -> count(x -> x == i, samples), 3:10)
    probs = counts ./ sum(counts)
    barplot(3:10, probs)
end
```

As we can see, this Markov chain has as its stationary distribution a geometric distribution conditioned to be greater than 2. The Markov chain above *implements* the inference below, in the sense that it specifies a way to sample from the required conditional distribution.

```
geometric (generic function with 2 methods)
```

```julia
geometric(ω, p, i = 0) = (i ~ Bernoulli(p))(ω) ? 1 : (1 + geometric(ω, p, i + 1))
```

```
mygeom = ᵛ#37
```

```julia
mygeom = Variable(ω -> geometric(ω, p))
```

```
post =  Conditional(ᵛ#37, >ₚ(ᵛ#37, 2))
```

```julia
post = mygeom |ᶜ (mygeom .> 2)
```

```
       ┌                                         ┐
     3 ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 0.69952
     4 ┤■■■■■■■■■ 0.21116
     5 ┤■■■ 0.0638
     6 ┤■ 0.01812
     7 ┤ 0.00516
     8 ┤ 0.00168
     9 ┤ 0.00032
    10 ┤ 0.00024
       └                                         ┘
```

```julia
let
    samples = randsample(post, 25000)
    counts = map(i -> count(x -> x == i, samples), sort(unique(samples)))
    probs = counts ./ sum(counts)
    barplot(sort(unique(samples)), probs)
end
```

Markov chain Monte Carlo (MCMC) is an approximate inference method based on identifying a Markov chain whose stationary distribution matches the conditional distribution you'd like to estimate. If such a transition distribution can be identified, we simply run it forward to generate samples from the target distribution.

Metropolis-Hastings

Fortunately, it turns out that for any given (conditional) distribution there are Markov chains with a matching stationary distribution. There are a number of methods for finding an appropriate Markov chain. One particularly common method is *Metropolis Hastings* recipe.

To create the necessary transition function, we first create a proposal distribution, $q(x \to x\prime)$, which does not need to have the target distribution as its stationary distribution, but should be easy to sample from (otherwise it will be unwieldy to use!). A common option for continuous state spaces is to sample a new state from a multivariate Gaussian centered on the current state. To turn a proposal distribution into a transition function with the right stationary distribution, we either accepting or reject the proposed transition with probability: $min(1, \frac{p(x\prime)q(x\prime \to x)}{p(x)q(x \to x\prime)})$. That is, we flip a coin with that probability: if it comes up heads our next state is x', otherwise our next state is still x.

Such a transition function not only satisfies the balance condition, it actually satisfies a stronger condition, *detailed balance*. Specifically, $p(x)\pi(x \to x\prime) = p(x\prime)\pi(x\prime \to x)$. (To show that detailed balance implies balance, substitute the right-hand side of the detailed balance equation into the balance equation, replacing the summand, and then simplify.) It can be shown that the Metropolis-hastings algorithm gives a transition probability (i.e. $\pi(x \to x\prime)$) that satisfies detailed balance and thus balance.

Note that in order to use this recipe we need to have a function that computes the target probability (not just one that samples from it) and the transition probability, but they need not be normalized (since the normalization terms will cancel).

We can use this recipe to construct a Markov chain for the conditioned geometric distribution, as above, by using a proposal distribution that is equally likely to propose one number higher or lower:

```
target_dist (generic function with 1 method)
```

```
# The target distribution (not normalized):
# prob = 0 if x condition is violated, otherwise proportional to geometric distribution
target_dist(x, p) = (x < 3) ? 0 : (p * ((1-p) ^ (x-1)))
```

```
accept (generic function with 1 method)
```

```
# The MH recipe:
function accept(x1, x2, p, ω, i)
    prob = min(1, target_dist(x2, p)/target_dist(x1, p))
    return ((@uid, i) ~ Bernoulli(prob))(ω)
end
```

```
mcmc (generic function with 1 method)
```

```julia
# the MCMC loop:
function mcmc(state, iterations, p, ω)
    s = [state]
    for i in 2:iterations
        # here we're equally likely to propose x+1 or x-1
        proposed_state = (i ~ Bernoulli())(ω) ? (state - 1) : (state + 1)
        state = accept(state, proposed_state, p, ω, i) ? proposed_state : state
        push!(s, state)
    end
    s
end
```

```
    ┌                                        ┐
  3 ┤■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 0.7068
  4 ┤■■■■■■■■■■ 0.2108
  5 ┤■■■ 0.0599
  6 ┤■ 0.0176
  7 ┤ 0.0039
  8 ┤ 0.0009
  9 ┤ 0.0001
    └                                        ┘
```

```julia
let
    samples = randsample(ω -> mcmc(3, 10000, p, ω)) # mcmc for conditioned geometric
    counts = map(i -> count(x -> x == i, samples), sort(unique(samples)))
    probs = counts ./ sum(counts)
    barplot(sort(unique(samples)), probs)
end
```

Note that the transition function that is automatically derived using the MH recipe is actually the same as the one we wrote by hand earlier.