Chapter 8: Interlude – Algorithms for Inference

8.1 Prologue: Performance Characteristics of Different Algorithms

When we introduced conditioning we pointed out that the rejection sampling and enumeration (or mathematical) definitions are equivalent—we could take either one as the definition of how `Infer` should behave with `condition` statements. There are many different ways to compute the same distribution, it is thus useful to separately think about the distributions we are building (including conditional distributions) and how we will compute 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 `Infer` 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 (implemented in `method:"rejection"`) 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:

```
const baserate = 0.1

var infModel = function() {
  Infer(
    {method: 'rejection', samples: 100},
    function() {
      var A = flip(baserate)
      var B = flip(baserate)
      var C = flip(baserate)
      condition(A + B + C >= 2)
      return A
    }
  )
}
```

```
// A timing utility: run 'foo' 'trials' times,
// report average time.
var time = function(foo, trials) {
  var start = _.now()
  var ret = repeat(trials, foo)
  var end = _.now()
  return (end - start) / trials
}

time(infModel, 10)
```

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

Another option that we've seen before is to enumerate all of the possible executions of the model, using the rules of probability to calculate the conditional distribution:

```
///fold:
// A timing utility: run 'foo' 'trials' times, report average time.
var time = function(foo, trials) {
  var start = _.now()
  var ret = repeat(trials, foo)
  var end = _.now()
  return (end - start) / trials
}
///

var baserate = 0.1

var infModel = function(){
  Infer(
    {method: 'enumerate'},
    function(){
      var A = flip(baserate)
      var B = flip(baserate)
      var C = flip(baserate)
      condition(A + B + C >= 2)
      return A
    }
  )
}

time(infModel, 10)
```

Notice that the time it takes for this program to run doesn't depend on the baserate. Unfortunately it does depend critically on the number of random choices in an execution history: the number of possible histories that must be considered grows exponentially in the number of random choices. To see this we modify the model to allow a flexible number of `flip` choices:

```
///fold:
// A timing utility: run 'foo' 'trials' times,
// report average time.
var time = function(foo, trials) {
  var start = _.now()
  var ret = repeat(trials, foo)
  var end = _.now()
  return (end - start) / trials
}
///

const baserate = 0.1
const numFlips = 3

var infModel = function() {
  Infer(
    {method: 'enumerate'},
    function() {
      var choices = repeat(
        numFlips,
        function() {
          flip(baserate)
        }
      )
      condition(sum(choices) >= 2)
      return choices[0]
    }
  )
}

time(infModel, 10)
```

The dependence on size of the execution space renders enumeration impractical for many models. In addition, enumeration isn't feasible at all when the model contains a continuous distribution (because there are uncountably many value that would need to be enumerated). Try inserting `var x = gaussian(0,1)` in the above model.

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).

```
///fold:
// A timing utility: run 'foo' 'trials' times,
// report average time.
var time = function(foo, trials) {
  var start = _.now()
  var ret = repeat(trials, foo)
  var end = _.now()
  return (end - start) / trials
}
///

const baserate = 0.1
const numFlips = 3

var infModel = function() {
  Infer(
    {method: 'MCMC', lag: 100},
    function() {
      var choices = repeat(
        numFlips,
        function() {
          flip(baserate)
        }
      )
      condition(sum(choices) >= 2)
      return choices[0]
    }
  )
}

time(infModel, 10)
```

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). Unlike enumeration, 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.

8.2 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.

8.2.1 Markov Chains as Samplers

A Markov model (or Markov *chain*, as it is often called in the context of inference algorithms) is a discrete dynamical system that unfolds over iterations of the `transition` function. Here is a Markov chain:

```
var states = ['a', 'b', 'c', 'd'];
var transitionProbs = {
  a: [.48, .48, .02, .02],
  b: [.48, .48, .02, .02],
  c: [.02, .02, .48, .48],
  d: [.02, .02, .48, .48]
}

var transition = function(state) {
  return categorical({
    vs: states,
    ps: transitionProbs[state]
  })
}

var chain = function(state, n) {
  return (
    n == 0
      ? state
      : chain(transition(state), n-1)
  )
}
```

```
print("State after 10 steps:")
viz.hist(
  repeat(1000,function() {chain('a',10)})
)
viz.hist(
  repeat(1000,function() {chain('c',10)})
)

print("State after 25 steps:")
viz.hist(
  repeat(1000,function() {chain('a',25)})
)
viz.hist(
  repeat(1000,function() {chain('c',25)})
)

print("State after 50 steps:")
viz.hist(
  repeat(1000,function() {chain('a',50)})
)
viz.hist(
  repeat(1000,function() {chain('c',50)})
)
```

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 $\pi(x \to x')$ 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*: $p(x') = \sum_x p(x)\pi(x \to x')$. 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 `['a', 'b', 'c', 'd']`! We could have sampled from the uniform distribution using other Markov chains. For instance the following chain is more natural, since it transitions uniformly:

```
var states = ['a', 'b', 'c', 'd'];
var transition = function(state) {
    return categorical({
    vs: states,
```

```
    ps: [.25, .25, .25, .25]
  })
}

var chain = function(state, n) {
    return (
    n == 0
      ? state
      : chain(transition(state), n-1)
  )
}

print("State after 10 steps:")
viz.hist(
  repeat(1000,function() {chain('a',10)})
)
viz.hist(
  repeat(1000,function() {chain('c',10)})
)

print("State after 25 steps:")
viz.hist(
  repeat(1000,function() {chain('a',25)})
)
viz.hist(
  repeat(1000,function() {chain('c',25)})
)

print("State after 50 steps:")
viz.hist(
  repeat(1000,function() {chain('a',50)})
)
viz.hist(
  repeat(1000,function() {chain('c',50)})
)
```

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:

```
const p = 0.7

var transition = function(state) {
    return (
    state == 3
    ? sample(
        Categorical({
          vs: [
            3,
            4
          ],
          ps: [
            (1 - 0.5 * (1 - p)),
            (0.5 * (1 - p))
          ]
        })
    )
    : sample(
        Categorical({
          vs: [
            (state - 1),
            state,
            (state + 1)
          ],
          ps: [
            0.5,
            (0.5 - 0.5 * (1 - p)),
            (0.5 * (1 - p))
          ]
        })
    )
  )
}

var chain = function(state, n) {
    return (
    n == 0
    ? state
```

```
    : chain(transition(state), n-1)
  )
}

var samples = repeat(
  5000,
  function() {chain(3, 250)}
)
viz.table(samples)
```

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.

```
const p = .7

var geometric = function(p) {
    return (
    flip(p) == true
    ? 1
    : (1 + geometric(p))
  )
}

var post = Infer({
  method: 'MCMC',
  samples: 25000,
  lag: 10,
  model: function() {
    var mygeom = geometric(p)
    condition(mygeom > 2)
    return(mygeom)
    }
})
viz.table(post)
```

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.

8.2.2 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*, $q(x \to x')$, 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:

$$\min\left(1, \frac{p(x')q(x' \to x)}{p(x)q(x \to x')}\right)$$

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,

$$p(x)\pi(x \to x') = p(x')\pi(x' \to x)$$

(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. $\pi(x \to x')$) that satisfies detailed balance and thus balance.[1]

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:

[1] Recommended exercise: prove this fact! *Hint:* the probability of transitioning depends on first proposing a given new state, then accepting it; if you don't accept the proposal you "transition" to the original state.

```
const p = 0.7

// The target distribution (not normalized):
// prob = 0 if x condition is violated,
// otherwise proportional to geometric distribution
var target_dist = function(x) {
  return (
    x < 3
    ? 0
    : p * Math.pow((1-p),(x-1))
  )
}

// The proposal function and distribution:
// here we're equally likely to propose
// x+1 or x-1.
var proposal_fn = function(x) {
  return (
    flip()
    ? x - 1
    : x + 1
  )
}
var proposal_dist = function (x1, x2) {
  return 0.5
}

// The MH recipe:
var accept = function (x1, x2) {
  var p = Math.min(
    1, (
      (target_dist(x2) * proposal_dist(x2, x1))
      /
      (target_dist(x1) * proposal_dist(x1,x2))
    )
  )
  return flip(p)
}
var transition = function(x) {
  var proposed_x = proposal_fn(x)
  return (
    accept(x, proposed_x)
```

```
    ? proposed_x
    : x
  )
}

// The MCMC loop:
var mcmc = function(state, iterations) {
  return (
    iterations == 1
    ? [state]
    : mcmc(
      transition(state), iterations - 1
    ).concat(state)
  )
}

// mcmc for conditioned geometric
var chain = mcmc(3, 10000)
viz.table(chain)
```

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:

```
var transition = function(state) {
    return (
    state == 3
    ? sample(
      Categorical({
        vs: [
          3,
          4
        ],
        ps: [
          1 - 0.5 * (1 - p),
          0.5 * (1 - p)
        ]
      })
    )
    : sample(
      Categorical({
        vs: [
          state - 1,
```

```
          state,
          state + 1
        ],
        ps: [
          0.5,
          0.5 - 0.5 * (1 - p),
          0.5 * (1 - p)
        ]
      })
    )
  )
}
```

8.2.3 Hamiltonian Monte Carlo

WebPPL's `method: 'MCMC'` uses *Metropolis-Hastings* by default. However, it is not the only option, nor is it always the best.

When the input to a `factor` statement is a function of multiple variables, those variables become correlated in the posterior distribution. If the induced correlation is particularly strong, MCMC can sometimes become 'stuck.'

In controlling the random walk, Metropolis-Hastings choses a new point in probability space to go to and then decides whether or not to go based on the probability of the new point. If it has difficulty finding new points with reasonable probability, it will get stuck and simply stay where it is. Given an infinite amount of time, Metropolis-Hastings will recover. However, the first N samples will be heavily dependent on where the chain started (the first sample) and will be a poor approximation of the true posterior.

Take this example below, where we use a Gaussian likelihood to encourage ten uniform random numbers to sum to the value 5:

```
var constrainedSumModel = function() {
  var xs = repeat(
    10,
    function() { uniform(0, 1) }
  )
  var targetSum = 5.0
  observe(
    Gaussian({mu: targetSum, sigma: 0.005}),
    sum(xs)
  )
  return xs
```

```
};

var opts = {
  method: 'MCMC',
  samples: 5000,
  callbacks: [MCMC_Callbacks.finalAccept]
}
var post = Infer(opts, constrainedSumModel)

print('')
print(sample(post))
print(sample(post))
print(sample(post))
print(sample(post))
print(sample(post))
print(sample(post))
```

The output box displays 10 random samples from the posterior. You'll notice that they are all very similiar, despite there being many distinct ways for ten real numbers to sum to 5. The reason is technical but straight-forward. The default version of MCMC used by WebPPL is Metropolis Hastings (MH). As described above, MH proposes new states and then uses the MH acceptance rule to decide whether to move there. The program above uses the `callbacks` option to `MCMC` to display the final acceptance ratio (i.e. the percentage of proposed samples that were accepted)–we see it is around 1-2%. This means there are few "new" states accepted by the Markov chain.

To deal with situations like this one, WebPPL provides an implementation of Hamiltonian Monte Carlo (HMC). HMC automatically computes the gradient of the target distribution with respect to the random choices made by the program. It uses this gradient information to make coordinated proposals to all the random choices. This can yield much better proposals for MH. Below, we apply HMC to `constrainedSumModel`:

```
var constrainedSumModel = function() {
  var xs = repeat(
    10,
    function() { uniform(0, 1) }
  )
  var targetSum = 5.0
  observe(
    Gaussian({mu: targetSum, sigma: 0.005}),
    sum(xs)
  )
```

```
  return xs
}

var opts = {
  method: 'MCMC',
  samples: 100,
  callbacks: [MCMC_Callbacks.finalAccept],
  kernel: {
    HMC: { steps: 50, stepSize: 0.0025 }
  }
}
var post = Infer(opts, constrainedSumModel)

print('')
print(sample(post))
print(sample(post))
print(sample(post))
print(sample(post))
print(sample(post))
print(sample(post))
```

The approximate posterior samples produced by this program are more varied, and the final acceptance rate is much higher!

There are a couple of caveats to keep in mind when using HMC:

- Its parameters can be extremely sensitive. Try increasing the `stepSize` option to `0.004` and see how the output samples degenerate.
- It is only applicable to continuous random choices, due to its gradient-based nature.[2]

8.2.4 Some Technicalities and Practicalities

- The successive samples generated by a single MCMC chain can be very similar, technically speaking they have high autocorrelation. A way of limiting the effects of this autocorrelation is to simple thin out the samples. This corresponds to the `lag` parameter to the `MCMC` inference method.
- In MH autocorrelation often comes from too many rejected proposals. To monitor this we can keep an eye on the acceptance rate. When it is too low (rule of thumb: below

0.3

[2]You can use WebPPl's HMC with models that include discrete choices: under the hood, this will alternate between HMC for the continuous choices and MH for the discrete choices.

for MH, below

0.9

for HMC) we will need to adjust the parameters of the algorithm or switch to a different method.

- For MH, and related techniques, the very first state of a Markov chain needs to be possible (have non-zero probability according to the target model), otherwise transition probabilities are not well defined. This can be surprisingly hard, especially when the hard `condition` operator is used. Switching to `observe`, where possible, solves this problem (and also improves efficiency during sampling).
- In order for MCMC to converge to a stationary distribution that accurately reflects the target model, it must be possible to reach any (possible) state from any other eventually. This condition is called *ergodicity*. It is mathematically important, but not something practitioners usually need to worry about. (It can generally be guaranteed by randomly restarting the chain occasionally, though this impacts efficiency.)

8.3 Particle Filters

A particle filter – also known as Sequential Monte Carlo – maintains a collection of samples (aka particles) *simultaneously* in parallel while executing the model. (This is different than MCMC, where samples are complete executions, each constructed sequentially from the last.) The particles are "re-sampled" upon encountering new evidence, in order to adjust the numbers so that the population will be approximately distributed according to the model. SMC is particularly useful for models where beliefs can be incrementally updated as new observations come in.

Let's consider another simple model, where five real numbers are constrained to be close to their neighbors:

```
var pairwiseSameModel = function() {
  var a = uniform(0, 1)
  var b = uniform(0, 1)
  var c = uniform(0, 1)
  var d = uniform(0, 1)
  var e = uniform(0, 1)
  observe(
    Gaussian({mu: 0, sigma: 0.005}),
    a - b
  )
  observe(
    Gaussian({mu: 0, sigma: 0.005}),
    b - c
  )
```

```
  observe(
    Gaussian({mu: 0, sigma: 0.005}),
    c - d
  )
  observe(
    Gaussian({mu: 0, sigma: 0.005}),
    d - e
  )
  return [a,b,c,d,e]
};

var opts = {
  method: 'MCMC',
  samples: 100,
  callbacks: [MCMC_Callbacks.finalAccept]
}
var post = Infer(opts, pairwiseSameModel)

print('')
print(sample(post))
print(sample(post))
print(sample(post))
print(sample(post))
print(sample(post))
print(sample(post))

viz(
  marginalize(
    post,
    function(x) { x[0] }
  )
)
```

We can easily tell that the marginal distribution on the first number *should* be approximately uniform, yet the sampled values are nowhere close to that. This is another case where the MH procedure finds it difficult to accept new states. Intuitively, rather than choosing the five numbers at once, we could choose one number then use the pairwise similarity constraint to choose the next number close to the first, and so on. Particle filtering will be a (mathematically correct) version of this idea, but only if we rewrite the model to interleave the random choices with the observations:

```
var pairwiseSameModel = function() {
  var a = uniform(0, 1)
  var b = uniform(0, 1)
  observe(Gaussian(
    {mu: 0, sigma: 0.005}),
    a - b
  )
  var c = uniform(0, 1)
  observe(
    Gaussian({mu: 0, sigma: 0.005}),
    b - c
  )
  var d = uniform(0, 1)
  observe(
    Gaussian({mu: 0, sigma: 0.005}),
    c - d
  )
  var e = uniform(0, 1)
  observe(
    Gaussian({mu: 0, sigma: 0.005}),
    d - e
  )
  return [a,b,c,d,e]
};

var opts = {
  method: 'SMC',
  particles: 1000
}
var post = Infer(opts, pairwiseSameModel)

print('')
print(sample(post))
print(sample(post))
print(sample(post))
print(sample(post))
print(sample(post))
print(sample(post))

viz(
  marginalize(
    post,
```

```
    function(x) { x[0] }
  )
)
```

Try switching back to the version of the model that does not interleave observations with sampling. Do the samples look worse?

When a particle filter encounters new evidence, it updates its collection of particles. Those particles that predict the new data well are likely to be retained or even multiplied. Those particles that do not predict the new data well are likely to be eliminated. Thus, particle filters integrate new data with prior beliefs. However, if few particles capture the new observation well, the particle filter may be forced to use *only* these few – this dynamic, called "filter collapse", results in poor samples. Try reducing the number of particles in the above model.

Consider a more complex example, motivated by radar detection, where the aim is to infer the trajectory of a moving object–the program receives a sequence of noisy observations and must infer the underlying sequence of true object locations.

The code below generates observations from a randomly-sampled underlying trajectory. Our program assumes that the object's motion is governed by a momentum term which is a function of its previous two locations; this tends to produce smoother trajectories.

```
///fold: helper functions for drawing
var drawLines = function(canvas, start, positions) {
  if (positions.length == 0) {
    return []
  }
  var next = positions[0];
  canvas.line(
    start[0], start[1], next[0], next[1], 4, 0.2
  );
  drawLines(
    canvas, next, positions.slice(1)
  )
}

var drawPoints = function(canvas, positions, mycolor) {
  if (positions.length == 0) {
    return []
  }
  var next = positions[0]
  canvas.circle(
    next[0], next[1], 2, mycolor, "white"
```

```
  )
  drawPoints(
    canvas, positions.slice(1), mycolor
  )
}
///

var genObservation = function(pos) {
  return map(
    function(x){ return gaussian(x, 15) },
      pos
  )
}

var init = function(){
    var state1 = [
    gaussian(300, 1),
    gaussian(300, 1)
  ]
    var state2 = [
    gaussian(300, 1),
    gaussian(300, 1)
  ]
    var states = [state1, state2]
  var observations = map(genObservation, states)
    return {
        states: states,
        observations: observations
    }
}

var transition = function(lastPos, secondLastPos) {
  return map2(
    function(lastX, secondLastX) {
      var momentum = (lastX - secondLastX) * .7
      return gaussian(lastX + momentum, 3)
    },
    lastPos,
    secondLastPos
  )
}
```

```
var trajectory = function(n) {
  var prevData = (n == 2)
    ? init()
    : trajectory(n - 1)
  var prevStates = prevData.states
  var prevObservations = prevData.observations
  var newState = transition(
    last(prevStates),
    secondLast(prevStates)
  )
  var newObservation = genObservation(newState)
  return {
    states: prevStates.concat([newState]),
    observations: prevObservations.concat([newObservation])
  }
}

const numSteps = 80
var atrajectory = trajectory(numSteps)
var synthObservations = atrajectory.observations
var trueLocs = atrajectory.states
var canvas = Draw(400, 400, true)
drawPoints(
  canvas, synthObservations, "grey"
) // observations
drawPoints(
  canvas, trueLocs, "blue"
) // actual trajectory
```

The actual trajectory is displayed in blue. The observations are in grey.

We can then use `'SMC'` inference to estimate the underlying trajectory which generated a synthetic observation sequence:

```
///fold:
var drawLines = function(canvas, start, positions, mycolor) {
  if (positions.length == 0) {
    return []
  }
  var next = positions[0]
  canvas.line(
    start[0], start[1], next[0], next[1], 4, 0.2, mycolor
```

```
  )
  drawLines(
    canvas, next, positions.slice(1), mycolor
  )
}

var drawPoints = function(canvas, positions, mycolor) {
  if (positions.length == 0) {
    return []
  }
  var next = positions[0]
  canvas.circle(
    next[0], next[1], 2, mycolor, "white"
  )
  drawPoints(
    canvas, positions.slice(1), mycolor
  )
}

var genObservation = function(pos) {
  return map(
    function(x){ return gaussian(x, 15) },
      pos
  )
}

var init = function() {
    var state1 = [
    gaussian(250, 1),
    gaussian(250, 1)
  ]
    var state2 = [
    gaussian(250, 1),
    gaussian(250, 1)
  ]
    var states = [state1, state2]
  var observations = map(genObservation, states)
    return {
        states: states,
        observations: observations
    }
}
```

```
var transition = function(lastPos, secondLastPos) {
  return map2(
    function(lastX, secondLastX) {
      var momentum = (lastX - secondLastX) * .7
      return gaussian(lastX + momentum, 3)
    },
      lastPos,
    secondLastPos
  )
}

var trajectory = function(n) {
  var prevData = (n == 2)
    ? init()
    : trajectory(n - 1)
  var prevStates = prevData.states;
  var prevObservations = prevData.observations;
  var newState = transition(
    last(prevStates),
    secondLast(prevStates)
  )
  var newObservation = genObservation(newState);
  return {
    states: prevStates.concat([newState]),
    observations: prevObservations.concat([newObservation])
  }
}
///

var observeSeq = function(pos, trueObs) {
  return map2(
    function(x, trueObs) {
        return observe(
        Gaussian({mu: x, sigma: 5}),
        trueObs
      )
    },
      pos,
    trueObs
  )
}
```

```
var initWithObs = function(trueObs) {
    var state1 = [
    gaussian(250, 1),
    gaussian(250, 1)
  ]
    var state2 = [
    gaussian(250, 1),
    gaussian(250, 1)
  ]
  var obs1 = observeSeq(state1, trueObs[0]);
  var obs2 = observeSeq(state2, trueObs[1]);
    return {
        states: [state1, state2],
        observations: [obs1, obs2]
    }
}

var trajectoryWithObs = function(n, trueObservations) {
  var prevData = (n == 2)
    ? initWithObs(
      trueObservations.slice(0, 2)
    )
    : trajectoryWithObs(
      n - 1,
      trueObservations.slice(0, n-1)
    )
  var prevStates = prevData.states;
  var prevObservations = prevData.observations;
  var newState = transition(
    last(prevStates),
    secondLast(prevStates)
  )
  var newObservation = observeSeq(
    newState,
    trueObservations[n-1]
  )
  return {
    states: prevStates.concat([newState]),
    observations: prevObservations.concat([newObservation])
  }
}
```

```
const numSteps = 80
const numParticles = 10

// Gen synthetic observations
var atrajectory = trajectory(numSteps)
var synthObservations = atrajectory.observations;
var trueLocs = atrajectory.states;

// Infer underlying trajectory using particle filter
var posterior = Infer(
  {method: 'SMC', particles: numParticles},
  function() {
    return trajectoryWithObs(
      numSteps,
      synthObservations
    )
  }
)
var inferredTrajectory = sample(posterior).states

// Draw model output
var canvas = Draw(400, 400, true)
drawPoints(
  canvas, synthObservations, "grey"
) // observations
drawLines(
  canvas,
  inferredTrajectory[0],
  inferredTrajectory.slice(1),
  "blue"
) // inferred
drawLines(
  canvas,
  trueLocs[0],
  trueLocs.slice(1),
  "green"
) // true
```

Again, the actual trajectory is in green, the observations are in grey, and the inferred trajectory is in blue. Try increasing or decreasing the number of particles to see how this affects inference. Also try using MH or HMC for inference.

8.4 Variational Inference

The previous parts of this chapter focused on Monte Carlo methods for approximate inference: algorithms that generate a (large) collection of samples to represent a conditional distribution. Another way to represent a distribution is by finding the closest approximation among a set (or "family") of simpler distributions. This is the approach taken by *variational inference*. At a high level, we declare a set of models that have the same choices as our target model, but don't have any conditions (i.e. no `condition`, `observe`, or `factor`); we then try to find the member of this set closest to our target model and use it as the result of `Infer`.

To search for a good approximating model, we will eventually use gradient-based techniques. For this reason, we don't want a set of isolated models, but a continuous family. In WebPPL we declare parameters of a family with `param()`. For instance, here is a family of Gaussian distributions with fixed variance but different means:

```
Gaussian({mu: param(), sigma: 0.1})
```

Because we want to make sure the family of distributions has the same choices as the target model, we define the two together. This is done with `guide` annotations to `sample` statements:

```
var gaussianModel = function() {
  var mu = sample(
    Gaussian({mu:0, sigma:20}), {
      guide: function() {
        Gaussian({mu:param(), sigma:param()})
      }
    }
  )
  var sigma = Math.exp(
    sample(
      Gaussian({mu:0, sigma:1}), {
        guide: function() {
          Gaussian({mu:param(), sigma:param()})
        }
      }
    )
  )
  map(
    function(d) {
      observe(
        Gaussian({mu: mu, sigma: sigma}),
        d
```

```
      )
    },
    data
  )
  return {mu: mu, sigma: sigma}
}
```

This represents both the conditional model, observed data drawn from a Gaussian of unknown mean and variance, and the (unconditional) family: Gaussian of adjustable mean and variance. If we were to separate them out they would look like this:

```
// Target model:
var gaussianModel = function() {
  var mu = sample(
    Gaussian({mu:0, sigma:20})
  )
  var sigma = Math.exp(
    sample(
      Gaussian({mu:0, sigma:1})
    )
  )
  map(
    function(d) {
      observe(
        Gaussian({mu: mu, sigma: sigma}),
        d
      )
    },
    data
  )
  return {mu: mu, sigma: sigma}
}

// Variational family:
var guideModelFamily = function() {
  var mu = sample(
    Gaussian({mu:param(), sigma:param()})
  )
  var sigma = Math.exp(
    sample(
      Gaussian({mu:param(), sigma:param()})
    )
```

```
  )
  //Note no observes!
  return {mu: mu, sigma: sigma}
}
```

Once we have specified the target model and the family we'll use to approximate it, our goal is to find the best member of the family – that is the one closest to the target model. Formally we want the member of the family with smallest Kullback-Liebler distance to the target model. WebPPL has built-in algorithms for minimizing this distance via gradient descent. This is called *variational inference*.

Here we use WebPPL inference method `optimize` to do variational inference in the above model:

```
const trueMu = 3.5
const trueSigma = 0.8

var data = repeat(
  100, function() {
    return gaussian(trueMu, trueSigma)
  }
)

var gaussianModel = function() {
  var mu = sample(
    Gaussian({mu:0, sigma:20}), {
      guide: function() {
        Gaussian({
          mu: param(),
          sigma: Math.exp(param())
        })
      }
    }
  )
  var sigma = Math.exp(
    sample(
      Gaussian({mu:0, sigma:1}), {
        guide: function() {
          Gaussian({
            mu: param(),
            sigma: Math.exp(param())
          })
```

```
        }
      }
    )
  )
  map(
    function(d) {
      observe(
        Gaussian({mu: mu, sigma: sigma}),
        d
      )
    },
    data
  )
  return {mu: mu, sigma: sigma}
}

var post = Infer({
  method: 'optimize',
  optMethod: {adam: {stepSize: .25}},
  steps: 250,
  samples: 1000
  },
  gaussianModel
)

viz.marginals(post)
```

Run this code, then try using MCMC to achieve the same result. You'll notice that MCMC takes significantly more steps/samples to give good results.

It is worth knowing that if no `guide` family is provided, WebPPL will fill one in by default:

```
const trueMu = 3.5
const trueSigma = 0.8

var data = repeat(
  100,
  function() {
    return gaussian(trueMu, trueSigma)
  }
)
```

```
var gaussianModel = function() {
  var mu = gaussian(0, 20)
  var sigma = Math.exp(
    gaussian(0, 1)
  )
  map(
    function(d) {
      observe(
        Gaussian({mu: mu, sigma: sigma}),
        d
      )
    },
    data
  )
  return {mu: mu, sigma: sigma}
}

var post = Infer({
  method: 'optimize',
  optMethod: {adam: {stepSize: .25}},
  steps: 250,
  samples: 1000
}, gaussianModel
)

viz.marginals(post)
```

The default guide family is constructed by replacing the arguments of random choices in the program with free parameters, which it then optimizes. This approach is known as *mean-field variational inference*: approximating the posterior with a product of independent distributions (one for each random choice in the program). Though it can be very useful, the mean-field approximation necessarily fails to capture correlation between variables. To see this, return to the model we used to explain the checkershaddow illusion:

```
const observedLuminance = 3

var model = function() {
  var reflectance = gaussian({mu: 1, sigma: 1})
  var illumination = gaussian({mu: 3, sigma: 1})
  var luminance = reflectance * illumination
  observe(
    Gaussian({
```

```
      mu: luminance,
      sigma: 1
    }),
    observedLuminance
  )
  return {
    reflectance: reflectance,
    illumination: illumination
  }
}

var post = Infer({
  // First use MCMC (with a lot of samples) to see what the posterior should look like
  method: 'MCMC',
  samples: 15000,
  lag: 100
  //then try optimization (VI):
  //     method: 'optimize',
  //     optMethod: {adam: {stepSize: .25}},
  //     steps: 250,
  //     samples: 5000
}, model)

viz.heatMap(post)
```

Try the above model with both `'optimize'` and `'MCMC'`, do you see how 'optimize' fails to capture the correlation? Think about why this is!

There are other methods for variational inference in addition to *mean-field*. We can instead approximate the posterior with a more complex family of distributions; for instance one that directly captures the (potential) correlation in the above example. To do so in WebPPL we need to explicitly describe the approximating (guide) family. First let's look at the above mean field approach, written with explicit guides:

```
const observedLuminance = 3

var model = function() {
  var reflectance = sample(
    Gaussian({mu: 1, sigma: 1}), {
      guide: function(){
        Gaussian({
          mu: param(),
```

```
          sigma: Math.exp(param())
        })
      }
    }
  )
  var illumination = sample(
    Gaussian({mu: 3, sigma: 1}), {
      guide: function() {
        Gaussian({
          mu: param(),
          sigma: Math.exp(param())
        })
      }
    }
  )
  var luminance = reflectance * illumination
  observe(
    Gaussian({mu: luminance, sigma: 1}),
    observedLuminance
  )
  return {
    reflectance: reflectance,
    illumination: illumination
  }
}

var post = Infer({
  method: 'optimize',
  optMethod: {adam: {stepSize: .01}},
  steps: 10000,
  samples: 1000
}, model)

viz.heatMap(post)
```

Now, we can alter the code in the guide functions to make the `illumination` posterior depend on the `reflectance`:

```
const observedLuminance = 3

var model = function() {
  var reflectance = sample(Gaussian({mu: 1, sigma: 1}),
```

```
                              {guide: function(){Gaussian({mu: param(), sigma:Math.exp(param())}
  var illumination = sample(Gaussian({mu: 3, sigma: 1}),
                               {guide: function(){Gaussian({mu: param()+reflectance*param(), sig
  var luminance = reflectance * illumination
  observe(Gaussian({mu: luminance, sigma: 1}), observedLuminance)
  return {reflectance: reflectance, illumination: illumination}
}

var post = Infer({
  method: 'optimize',
  optMethod: {adam: {stepSize: .01}},
  steps: 10000,
  samples: 1000
}, model)

viz.heatMap(post)
```

Here we have explicitly described a linear dependence of the mean of `illumination` on `reflectance`. Can you think of ways to adjust the guide functions to even better capture the true posterior?

By definition a parametric function can be described by some finite number of parameters. For instance, a Gaussian is fully described by two numbers: its mean and standard deviation. By approximating a complex posterior distribution within a parametric family, we can often achieve reasonable results much more quickly than Monte Carlo methods. Unlike Monte Carlo methods, however, if the true posterior is badly fit by the family we will never get good results!

8.4.1 Some Technicalities and Practicalities

- You might wonder why the results of variational inference look "bumpy" when the approximating family is, for example, nice smooth Gaussians? This is because optimization finds the best approximating guide model, but the marginal distribution on return values of this best guide must still be estimated. In WebPPL this final step is done by forward sampling from the guide model, hence the final result is still samples.
- Performance of variational inference can depend a lot on the parameters of the optimizer, especially the step size.
- Even though guide models must not observe data (or factor/condition), they *can* actually depend on the data that is observed in the target model. This is sometimes called *amortized* inference.

8.5 Heuristics for Choosing an Algorithm

Given this zoo of different algorithms for doing inference, which should you choose? Here is a simple checklist that can be useful:

- Can you use `enumerate`? If so this is the best choice. It won't work if you have continuous choices or especially huge discrete state spaces.
- Can you do rejection sampling? Try taking one sample and see how long it takes. If it's reasonable, this is the next best option.
- Do you want inference to be fast and can tolerate bias? If so, try variational inference. Start with mean field and then make fancier guide families as you understand your model better.
- If you notice that your model has observations interleaved with sampling (or can be written that way), then give SMC a shot. Keep an eye out for filter collapse.
- MCMC tends to be a good fall back (if you run it long enough, it'll do well…). HMC tends to be better when your model has continuous variables. Tune step size carefully by looking at acceptance rate.

In fact, this decision heuristic is pretty much what WebPPL `Infer` does when no method is specified!

This is unfortunately not an exhaustive procedure, and some of the steps can require intuition. (e.g. how do you know if your variational algorithm is working?) There is a great deal written about diagnostics and rules of thumb for each inference algorithm!