Algorithms for inference

Prologue: The performance characteristics of different algorithms

When we introduced conditioning (/chapters/conditioning.html) 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:

```
x
```

```
var infModel = function(baserate){
```

```
  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
```

```
}
```

```

```

```
var baserate = 0.1
```

```
time(function(){infModel(baserate)}, 10)
```

run ▼

Even for this simple program, lowering the baserate by just one order of magnitude, to 0.010.01, 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:

```
var infModel = function(baserate){
  Infer({method: 'enumerate'}, 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
}

var baserate = 0.1
time(function(){infModel(baserate)}, 10)
```

run ▼

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:

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

```
}

//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 numFlips = 3
time(function(){infModel(baserate,numFlips)}, 10)
```

run ▼

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).

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.

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

//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 numFlips = 3
```

```
time(function(){infModel(baserate,numFlips)}, 10)
```

run ▼

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 tradeoffs that are common between different inference algorithms.

The landscape of inference algorithms

Portions of the following were adapted from "Notes of the PPAML Summer School 2016 (http://probmods.github.io/ppaml2016/)".

Analytic Solutions

Conceptually, the simplest way to determine the probability of some variable under Bayesian inference is simply to apply Bayes' Rule and then carry out all the necessary multiplication, etc. However, this is not always possible.

For instance, suppose your model involves a continuous function such as a `gaussian` and `gamma`. Such choices can take on an infinite number of possible values, so it is not possible to consider every one of them. In WebPPL, if we use `method: 'enumerate'` to try to calculate the analytic solution for such a model using, we get a runtime error:

```
var gaussianModel = function() {
```

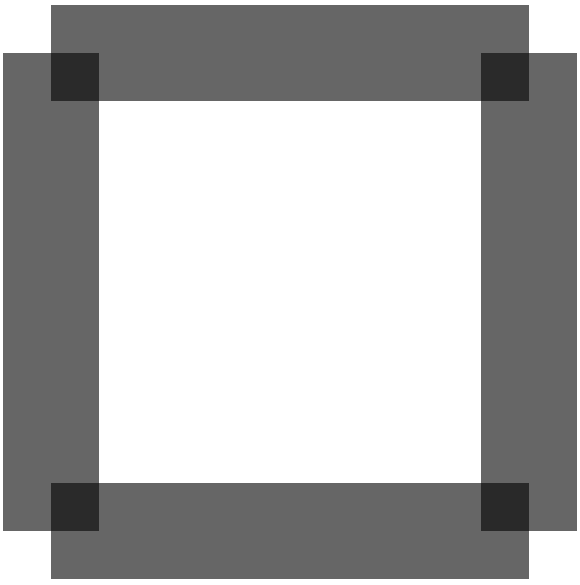
```
    return sample(Gaussian({mu: 0, sigma: 1}))
```

```
};
```

```
Infer({method: 'enumerate'}, gaussianModel);
```

run ▼

Even when all the variables are categorical, problems arise quickly. As a program makes more random choices, and as these choices gain more possible values, the number of possible execution paths through the program grows exponentially. Explicitly enumerating all of these paths can be prohibitively expensive. For instance, consider this program which computes the posterior distribution on rendered 2D lines, conditioned on those lines approximately matching this target image:



```
///fold:
```

```
...
```

```

```

```
var makeLines = function(n, lines, prevScore){
```

```
  // Add a random line to the set of lines
```

```
  var x1 = randomInteger(50);
```

```
  var y1 = randomInteger(50);
```

```
  var x2 = randomInteger(50);
```

```
  var y2 = randomInteger(50);
```

```
  var newLines = lines.concat([[x1, y1, x2, y2]]);
```

```
  // Compute image from set of lines
```

```
  var generatedImage = Draw(50, 50, false);
```

```
  drawLines(generatedImage, newLines);
```

```
  // Factor prefers images that are close to target image
```

```
  var newScore = -targetImage.distance(generatedImage)/1000;
```

```
  factor(newScore - prevScore);
```

```
  generatedImage.destroy();
```

```
  // Generate remaining lines (unless done)
```

```
  return (n==1) ? newLines : makeLines(n-1, newLines, newScore);
```

```
};
```

```

```

```
var lineDist = Infer(
```

```
  { method: 'enumerate', strategy: 'depthFirst', maxExecutions: 10 },
```

```
  function(){
```

```
    var lines = makeLines(4, [], 0);
```

```
    var finalGeneratedImage = Draw(50, 50, true);
```

```
    drawLines(finalGeneratedImage, lines);
```

```
    return lines;
```

```
  });
```

```

```

```
viz.table(lineDist);
```

run ▼

Running this program, we can see that enumeration starts by growing a line from the bottom-right corner of the image, and then proceeds to methodically plot out every possible line length that could be generated. These are all fairly terrible at matching the target image, and there are billions more states like them that enumeration would have to wade through in order to find those few that have high probability.

Approximate Inference

Luckily, it is often possible to estimate the posterior probability fairly accurately, even though we cannot calculate it exactly. There are a number of different algorithms, each of which has different properties.

Rejection Sampling

Rejection sampling (implemented in `method:"rejection"`), which we introduced in conditioning (/chapters/conditioning.html), is conceptually the simplest. However, it is not very efficient. Recall that it works by randomly sampling values for the variables and then checking to see if the condition is met, rejecting the sample if it is not. If the condition is *a priori* unlikely, the vast majority of samples will be rejected, and so it will take a very large number of samples to find computations that do so. To see this, try running the following model with progressively smaller values for `baserate`:

```
var baserate = 0.1

var model = function(){
  var A = flip(baserate)
  var B = flip(baserate)
  var C = flip(baserate)
  condition(A+B+C >= 2)
  return A
}

viz(Infer({method: 'rejection', samples: 100}, model))
```

run ▼

Even for this simple program – and even though we are only asking for 100 successful (non-rejected) samples – lowering the baserate by just one order of magnitude, to 0.01, slows down inference considerably. Lowering the baserate to 0.001 makes inference impractical.

It can be useful to compare this directly to what happens with enumeration. Changing the baserate has no effect on runtime, but adding additional variables (var D = flip(baserate), var E = flip(baserate), etc.) can slow down inference dramatically. (Why?)

```
var baserate = 0.1

var model = function(){
  var A = flip(baserate)
  var B = flip(baserate)
  var C = flip(baserate)
  condition(A+B+C >= 2)
```

```
  return A
}

viz(Infer({method: 'enumerate'}, model))
```

run ▼

Markov chain Monte Carlo (MCMC)

With rejection sampling, each sample is an independent draw from the model's prior. Markov chain Monte Carlo, in contrast involves a random walk through the posterior. Each sample depends on the prior sample – but ony the prior sample (it is a *Markov* chain). We describe this in more detail below.

Importantly, while you can approximate an arbitrary conditional distribution with arbitrary precision using rejection sampling or MCMC if you run the algorithms long enough, MCMC tends to approach the conditional distribution much more rapidly. Consider again this simple model:

```
var baserate = 0.1

var model = function(){
  var A = flip(baserate)
  var B = flip(baserate)
  var C = flip(baserate)
  condition(A+B+C >= 2)
  return A
}

viz(Infer({method: 'MCMC', lag: 100}, model))
```

run ▼

Again, see what happens in the above inference as you lower the baserate. Unlike rejection sampling, inference will not slow down appreciably, though 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 tradeoffs that are common between different inference algorithms.

Next, we provide more intuition on how MCMC works.

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)}))
```

run ▼

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 $p(x)p(x)$ be the target distribution, and let $\pi(x \rightarrow x')π(x→x′)$ 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*: $p(x') = \sum_x p(x)\pi(x \rightarrow x')p(x′)=∑xp(x)π(x→x′)$. 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 another (fairly baroque!) way to sample from the uniform distribution on `['a', 'b', 'c', 'd']`! Of course 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)}))
```

run ▼

Notice that this chain converges much more quickly to the uniform distribution. Edit the code to confirm to yourself that the chain converges on 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.

While state space in our examples above involved a finite number of states (4!), Markov chains can also be constructed over infinite state spaces. Here's a chain over the integers:

```
var 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
}
```

```
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)
```

run ▼

As we can see, this Markov chain has as its stationary distribution a geometric distribution (https://en.wikipedia.org/wiki/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. We can get the same computation using `Infer`:

```
var 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)
```

run ▼

Thus, MCMC involves identifying a Markov chain whose stationary distribution matches the condition distribution you'd like to estimate. That is, you want a Markov chain such that in the limit a histogram (or density plot) of states in the Markov chain approaches the conditional distribution in question.

As we have already seen, each successive sample from a Markov chain is highly correlated with the prior state. (Why?). To see another example, let's return to our attempt to match the 2D image. This time, we will take 50 MCMC samples:

```
///fold:
...

var makeLines = function(n, lines, prevScore){
```

```
  // Add a random line to the set of lines

  var x1 = randomInteger(50);

  var y1 = randomInteger(50);

  var x2 = randomInteger(50);

  var y2 = randomInteger(50);

  var newLines = lines.concat([[x1, y1, x2, y2]]);

  // Compute image from set of lines

  var generatedImage = Draw(50, 50, false);

  drawLines(generatedImage, newLines);

  // Factor prefers images that are close to target image

  var newScore = -targetImage.distance(generatedImage)/1000;

  factor(newScore - prevScore);

  generatedImage.destroy();

  // Generate remaining lines (unless done)

  return (n==1) ? newLines : makeLines(n-1, newLines, newScore);

};



var lineDist = Infer(

  { method: 'MCMC', samples: 50},

  function(){

    var lines = makeLines(4, [], 0);

    var finalGeneratedImage = Draw(50, 50, true);

    drawLines(finalGeneratedImage, lines);

    return lines;

  });



viz.table(lineDist);
```

run ▼

As you can see, each successive sample is highly similar to the previous one. Since the first sample is chosen randomly, the sequence you see will be very different if you re-run the model. If you run the chain long enough, these local correlations wash out. However, that can result in a very large collection of samples. For convenience, modelers sometimes record only every Nth states in the chain. WebPPL provides an option for MCMC called `'lag'`, which we actually saw in the first example from this section.

Metropolis-Hastings

Fortunately, it turns out htat for any given (condition) distribution we might want to sample from, there is at least one Markov chain 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\rightarrow x')q(x→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'\rightarrow x)}{p(x)q(x\rightarrow x')}\right).min(1,p(x)q(x→x′)p(x′)q(x′→x)

). 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 x.

Such a transition function not only satisfies the *balance condition*, it actually satisfies a stronger condition, *detailed balance*. Specifically, p(x)\pi(x \rightarrow x') = p(x')\pi(x' \rightarrow x)p(x)π(x→x′)=p(x′)π(x′→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\rightarrow x')π(x→x′)) that satisfies detailed balance and thus balance. (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.)

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:

```
var 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){
  let 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){
  let 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))
}

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

run ▼

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
}
```

run ▼

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 controling 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 simplly 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 factor to encourage ten uniform random numbers to sum to the value 5:

```
var bin = function(x) {
```

```
  return Math.floor(x * 1000) / 1000;
};

var constrainedSumModel = function() {
  var xs = repeat(10, function() {
    return uniform(0, 1);
  });
  var targetSum = xs.length / 2;
  observe(Gaussian({mu: targetSum, sigma: 0.005}), sum(xs));
  return map(bin, xs);
};

var post = Infer({
    method: 'MCMC',
    samples: 5000,
    callbacks: [MCMC_Callbacks.finalAccept]
}, constrainedSumModel);
var samps = repeat(10, function() { return sample(post); });
reduce(function(x, acc) {
    return acc + 'sum: ' + sum(x).toFixed(3) + ' | nums: ' + x.toString() + '\n';
}, '', samps);
```

run ▼

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 program above uses the `callbacks` option to `MCMC` to display the final acceptance ratio (i.e. the percentage of proposed samples that were accepted)–it should be around 1-2%, which is very inefficient.

To deal with situations like this one, WebPPL provides an implementation of Hamiltonian Monte Carlo (http://docs.webppl.org/en/master/inference.html#kernels), or HMC. HMC automatically computes the gradient of the posterior with respect to the random choices made by the program. It can then use the gradient information to make coordinated proposals to all the random choices, maintaining posterior correlations. Below, we apply HMC to `constrainedSumModel`:

```
///fold:
...

var post = Infer({
```

```
    method: 'MCMC',
    samples: 100,
    callbacks: [MCMC_Callbacks.finalAccept],
    kernel: {
        HMC : { steps: 50, stepSize: 0.0025 }
    }
}, constrainedSumModel);

var samps = repeat(10, function() { return sample(post); });

reduce(function(x, acc) {
    return acc + 'sum: ' + sum(x).toFixed(3) + ' | nums: ' + x.toString() + '\n';
}, '', samps);
```

run ▼

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 seeing how the output samples degenerate.
- It is only applicable to continuous random choices, due to its gradient-based nature. You can still use HMC with models that include discrete choices, though: under the hood, this will alternate between HMC for the continuous choices and MH for the discrete choices.

Particle Filters

Particle filters – also known as Sequential Monte Carlo (http://docs.webppl.org/en/master/inference.html#smc) – maintain a collection of samples (particles) that are resampled upon encountering new evidence. They are particularly useful for models that incrementally update beliefs as new observations come in. Before considering such models, though, let's get a sense of how particle filters work. Below, we apply a particle filter to our 2D image rendering model, using `method: 'SMC'`.

```
///fold: 2D image drawing
...

var numParticles = 100;

var post = Infer(
  {method: 'SMC', particles: numParticles},
  function(){
    return makeLines(4, [], 0);
   });
```

```
repeat(20, function() {
  var finalGeneratedImage = Draw(50, 50, true);
  var lines = sample(post);
  drawLines(finalGeneratedImage, lines);
});
```

run ▼

Try running this program multiple times. Note that while each run produces different outputs, within a run, all of the output particles look extremely similar. We will return to this issue later on in the next section.

Notice the variable `numParticles`. This sets the number of estimates (particles) drawn at each inference step. More particles tends to mean more precise estimates. Try adjusting `numParticles` in order to see the difference in accuracy.

For another example, consider inferring the 2D location of a static object given several noisy observations of its position, i.e. from a radar detector:

```
///fold: helper drawing function
...

// condition on x and y coords
var observePoint = function(pos, obs) {
  observe(Gaussian({mu: pos[0], sigma: 5}), obs[0]);
  observe(Gaussian({mu: pos[1], sigma: 5}), obs[1]);
};

var radarStaticObject = function(observations) {
  var pos = [gaussian(200, 100), gaussian(200, 100)];
  map(function(obs) { observePoint(pos, obs); }, observations);
  return pos;
};

var trueLoc = [250, 250]
var numParticles = 1000
var numObservations = 20

var observations = repeat(numObservations, function() {
  return [ gaussian(trueLoc[0], 100), gaussian(trueLoc[1], 100) ];
```

```
});

var posterior = Infer({method: 'SMC', particles: 1000}, function() {
  return radarStaticObject(observations);
});

var posEstimate = sample(posterior);

var canvas = Draw(400, 400, true);

drawPoints(canvas, observations, 'grey'); // observations

drawPoints(canvas, [posEstimate], 'blue'); // estimate

drawPoints(canvas, [trueLoc], 'green'); // actual location

posEstimate;
```

run ▼

We display the true location (`trueLoc`) in green, the observations in grey, and the inferred location (`posEstimate`) in blue. Again, try adjusting the number of particles (`numParticles`) and number of observations (`numObservations`) to see how these affect accuracy.

Interlude on `factor`/`observe` vs. `condition`

In earlier chapters we introduced the `factor` and `observe` keywords as soft alternatives to the hard `condition` statement in WebPPL. While the notion of `condition`ing on an observation being true is conceptually straight-forward, it should now be clearer from the details of algorithms what its computational drawbacks might be. In our model above, any given observation is *a priori* exremely unlikey, since our target can appear anywhere. For obvious reasons, rejection sampling will work poorly, since the chance that a random sample from a Gaussian will take on exactly the value `x` is negligible. Thus, randomly sampling and only retaining the samples where the Gaussian did take on the value `x` is an inefficient strategy. MCMC similarly has difficulty when the vast majority of possible parameter settings have probability 0. (Why?) In contrast, `factor` and `observe` provide a much softer constraint: parameter values that do not give rise to our observations are lower-probability, but not impossible.

Incremental inference based on incremental evidence

When a particle filter encounters new evidence, it updates its collection of particles (estimates). 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. This makes them particularly well-suited for programs that interleave inference and observation.

Below, we extend the the radar detection example to infer the trajectory of a moving object, rather than the position of a static one–the program receives a sequence of noisy observations and must infer the underlying sequence of true object locations. 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.

The code below generates observations from a randomly-sampled underlying trajectory (notice that we only have one observation per time step):

```
///fold: helper functions for drawing

...

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])
  }
};

var 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
```

run ▼

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 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])
  }
};

var numSteps = 80;
var 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
```

run ▼

Again, the actual trajectory is in green, the observations are in grey, and the inferred trajectory is in green. Try increasing or decreasing the number of particles to see how this affects inference.

Here (http://dritchie.github.io/web-procmod/) is a more complex example of using SMC to generate a 3D model that matches a given volumetric target (Note: this demo uses a much older version of WebPPL, so some of the syntax is different / not compatible with the code we've been working with).

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 the posterior distribution. This is a *non-parametric* (https://en.wikipedia.org/wiki/Nonparametric_statistics) representation of the posterior. Non-parametric methods are highly flexible but can require a very many expensive samples.

On the other side of the same coin, we have *parametric* (https://en.wikipedia.org/wiki/Parametric_statistics) representations–that is, we can try to design and fit a parameterized density function to approximate the posterior distribution. 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 acheive reasonabe result much more quickly. Unlike Monte Carlo methods, however, if the true posterior is badly fit by the family we will never get good results.

Thus, if we believe we can fit the distribution of interest reasonably well parametrically, there are a number of advantages to doing so. This is the approach taken by the family of variational inference (http://docs.webppl.org/en/master/inference.html#optimization) methods, and WebPPL provides a version of these algorithms via the `optimize` inference option (the name 'optimize' comes from the fact that we're optimizing the parameters of a density function to make it as close as possible to the true posterior). Below, we use `optimize` to fit the hyperparameters of a Gaussian distribution from data:

```
var trueMu = 3.5

var 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)) // ensure sigma > 0
  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
// Also try using MCMC and seeing how many samples it takes to converge
//   method: 'MCMC',
//   onlyMAP: true,
//   samples: 5000
}, gaussianModel)

viz.marginals(post)
```

run ▼

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

How does `optimize` work? By default, it takes the given arguments of random choices in the program (in this case, the arguments `(0, 20)` and `(0, 1)` to the two `gaussian` random choices used as priors) and replaces with them with free parameters which it then optimizes to bring the resulting distribution as close as possible to the true posterior. This approach is also known as *mean-field variational inference*: approximating the posterior with a product of independent distributions (one for each random choice in the program).

However, 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:

```
var 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)
```

run ▼

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 family, or *guide*. First let’s look at the above mean field approach, written with explicit guides:

```

var 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)
```

run ▼

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

```
var 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(), 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)
```

run ▼

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?

Test your knowledge: Exercises (/exercises/inference-algorithms.html)

Reading & Discussion: Readings (/readings/inference-algorithms.html)

Next chapter: 8. Rational process models (/chapters/process-models.html)