Efficient Inference in Probabilistic Computing

M.Eng Propsal

Jeff Wu

1 Introduction

Probabilistic programming languages are generalizations of programming languages, in which procedures are replaced with random procedures that induce distributions. By allowing for easy description and manipulation of distributions, probabilistic programming languages let one describe classical AI models in compact ways, and provide a language for very richer expression.

A core operation of probabilistic programming languages is inference, which is, in general, a difficult computational problem [**?**]. However, Markov chain Monte Carlo (MCMC) methods converge quite quickly to the posterior distribution for a large class of inference problems. Much effort has been devoted to studying in which cases this inference is efficient, and how to make it so.

We introduce a Python implementation of a programming language much like Church [**?**]. We implement inference in a relatively naive way, first, using a database to store all random choice. We then improve on this by introducing traces, letting us represent the program structure in a way that allows us to do minimal re-computation.

We plan to further explore making inference more efficient, by

1. Using just-in-time techniques on the compiler
2. Using a "reduced" traces implementation, in which edges of the traces graph are compressed
3. Potentially exploring some "adaptive" generalizations of Gibbs sampling

We'd like to be able to recover the efficiency of special-cased algorithms

2 Language description

2.1 Values

A **value** is a valid binding for a variable, in our language. For example, integers, floats, and booleans are valid values. We call these primitive values. Another type of value is a **procedure**, a function which gets applied to arguments and returns (not necessarily primitive) values.

These procedures should be non-deterministic. To allow for this, we can imagine that there is a special procedure, call it `bernoulli`, which takes an argument p, and outputs 0 or 1, with probabilities $1 - p$ and p, respectively. Arbitrary random procedures can then be built up using this one. We will see later that we actually can and will allow for something far more general than random procedures.

2.2 Expressions and Environments

Expressions are syntactic building blocks, which specified as being one of the following:

- Atomic expressions:
  - A value itself.
  - A variable name.
- Non-atomic expressions
  - An application of an expression to a list of argument expressions.
  - A function, consisting of some variable arguments, and a body expression.
  - An if statement, consisting of a branching expression, and two child expressions.
  - An operator statement, consisting of an operator (e.g. $==$, $<$, $+$, $\times$, `AND`, `OR`, `NOT`) and a list of argument expressions.

Expressions are **evaluated**, so that they take on some value. Their evaluation is relative to an **environment**, in which variable names may be bound to values.

Evaluation happens recursively. If an invalid expression is ever given (e.g. the first expression in an application isn't a procedure, or we call the equality operator with three arguments), an error is thrown.

2.3 Directives

There are four primitive operations, called **directives**, which the language supports. They are:

1. `sample(expr)` :

   Evaluates with respect to the global environment to sample a value for the expression `expr`.

2. `assume(name, expr)` :

   Samples a value for the expression `expr`, and then binds `name`, a string variable name, to the resulting value in the global environment.

3. `observe(expr, val)` :

   Observes the expression `expr` to have been evaluated to the value `val`.

4. `infer()` :

   Runs a single step of the Markov Chain.

The result of calling `infer()` sufficiently many times, should be that sampling gives the posterior distribution for expressions, conditioned on all the observed values.

3 Inference

Inference is by far the most complicated of the directives, and also the most important. Performing inference is the reason we are interested in probabilistic programming languages in the first place.

3.1 High-level description

We now describe the way in which inference is implemented, at a high level. The idea is to run Metropolis-Hastings on the space of all possible executions of the program.

Our proposal density, which chooses the new state (program execution) to potentially transition to, does the following:

1. Pick, at random, one of the random choices made, at any point in the program's execution history. Make a new choice for this randomness.
2. Rerun the entire program, changing only this choice. If this causes us to evaluate new sections of the code (a different side of an if statement, or a procedure called with different arguments), simply run these new sections. Also "unevaluate" the evaluation of previously-evaluated sections which are no longer evaluated.

We then use the update rule of Metropolis-Hastings to decide whether we enter this new state, or whether we keep the old one. It is simple to assign probabilities proportional to the posterior: If the new execution history doesn't agree with our observations, we assign it a probability of 0 (so we never transition to such a state). Otherwise, we look at all the choices of randomness in the program, see how likely those choices were, and take the product over all these choices.

3.2 Adding noise to observations

Unfortunately, as it has been described, this random walk doesn't converge to the posterior! Consider the following example, in which I flip two weighted coins and observe that they came up differently:

```
>>> p, q = 0.6, 0.4
>>> assume('a', bernoulli(p))
>>> assume('b', bernoulli(q))
>>> assume('c', var('a') ^ var('b'))
>>> observe('c', True)
```

It is easy to verify that the posterior distribution should have the state {`a:True, b:False`} with probability $\frac{9}{13}$ and {`a:False, b:True`} with probability $\frac{4}{13}$. However, we can see that if we start in the state {`a:True, b:False`}, we will never walk to either of the two adjacent states {`a:True, True`} and {`a:False, b:False`}. So the random walk stays in that state forever! The same argument applies for {`a:False, b:True`}. Thus if we draw a state from the prior, and run any number of steps of the random walk, the resulting distribution is identical to the prior.

To fix this problem, we should allow some small probability of transitioning to states where observations are false. After all, a Bayesian should never trust their eyes 100%. Thus suppose whenever we had an observation

```
>>> observe(expr, val)
```

we instead always used:

```
>>> observe(bernoulli(ifelse(expr == val, 1, noise_level)), True)
```

Now, when we run our walk, if the expression `expr == val` evaluates to `False` (so the original observation would've failed), we force the outermost `bernoulli` application to be `True`. If the noise level is small, this should not affect results much, since the execution histories in which the original observation was false are weighted against heavily.

Now, the space of states is always connected in the random walk and we should get convergence.

We provide the function `noisy(obs, noise_level)` as shorthand for `bernoulli(ifelse(obs, 1, noise_level))`, where you have observed some expression `obs` to be true.

3.3 XRPs

Evaluating and unevaluating sections of code is very easy when all random choices are made independently of one another. But actually, it is easy to evaluate and unevaluate so long as the sequence of random choices is **exchangeable sequence**, meaning roughly that the probabilities of making a set of choices is the same, regardless of what order we chose to make the choices.

Thus instead of merely being able to flip independent coins as a source of randomness for procedures, the most general thing we are able to allow what is called an **exchangeable random procedure (XRP)**. An XRP is a type of random procedure, but different applications of it are not guaranteed to be independent; they are guaranteed merely to be exchangeable.

For example, if f is an XRP taking one argument, we may find that $f(1) = 3$, and later that $f(2) = 4$. Unlike a typical procedure, the result $f(2) = 4$ was not necessarily independent of the earlier result $f(1) = 3$. However, the probability that this execution history happens should be the same as the probability that we first apply $f(2)$ and get 4, and then later apply $f(1)$ and get 3. [1]

XRPs used in our probabilistic programs are formally specified by the following four functions:

- `init()`, which returns an initial state.
- `get(state, args)`, which returns a value.
- `prob(state, value, args)`, which returns a probability.
- `inc(state, value, args)`, which returns a new state.
- `rem(state, value, args)`, which returns a new state.

The state of the XRP is a sufficient state, meaning it alone lets you determine the XRP's behavior. Minimally, the state may be a history of all previous applications of the XRP, although in most cases, it is much more succinct. Whenever we apply the XRP, we use `get` to determine the resulting value, and we may use `prob` to determine the probability that we got that value, conditioning on all previous applications. This value can then incorporated into the XRP using `inc`, and the state is updated. Finally, `rem` lets you remove a value, undoing an incorporate.

Notice that if applications are independent, then we do not need a state, and both incorporate and remove can do nothing. Thus this recovers the special case of normal random procedures.

[1] Notice that $f(2)$ can be replaced everywhere with $f(1)$ in this paragraph — multiple applications to the same arguments can also be different and non-independent, so long as they are exchangeable.

We can see why this exchangeability condition still lets us do inference. During inference, when we unevaluate, we may want to remove a value from the XRP's state, undoing the corresponding incorporate. But we may have incorporated new values since then. The key is that the exchangeability condition still lets us pretend the XRP application in question was the most recent one, and thus we can safely remove the value as if it were the most recent one.

XRPs are quite a general object, and replacing random procedures with them significantly increase the expressive power of our language. They also come in a variety of flavors, and we will see some examples later. Of course, we now allow defining procedures which use and return other XRPs, since XRPs are valid values.

Lastly, we allow for something much more general than the noisy observations above. Suppose that I observe that the sky is pink, and also that the ocean is yellow. Chances are, either my vision has gone crazy, or the world has underwent drastic changes. The correctness of my observations is highly correlated, and so we should allow for non-independent noise. Luckily, XRPs allow for this. We now simply require that all observed expressions have an outermost XRP application. When running inference, we always force the XRP to give the observed value by using `inc` (without using `get`). The probability of getting this observed value should always be non-zero.

The noisy expressions obtained with `noisy` are a special case of this, so long as `noise_level` is non-zero.

4 Implementations

4.1 The randomness database

In order to perform inference, we must store all random choices (XRP applications) in the execution of our program, so that we can evaluate and unevaluate.

The first challenge is figuring out how to specify points in the execution history. Essentially, we use a call stack, augmented with things telling us the location being considered within an expression.

Then, our database supports the following operations:

- `insert(stack, xrp, value, args, obs)`: Records that the `xrp` applied to `args`, at execution point specified by `stack`, returned `value`. The argument `obs` specifies whether this was an outermost "noise" XRP application in an observation.
- `remove(stack)`: Removes the random choice at the execution point specified by `stack`.
- `has(stack)`: Checks if the current execution history reaches the execution point specified by `stack`.
- `get(stack)`: Gets a tuple `(xrp, args, value, prob, obs)` containing all the relevant data corresponding to the random choice stored at `stack`. The elements of the tuple are, in order: the XRP applied, the list of arguments, the resulting value, the probability of getting that value, and whether it was an outermost "noise" application in an observation.
- `random_stack()`: Return a random stack in the database such that `obs = False`.

- `unevaluate(stack, args)`: Unevaluates everything in the database which was evaluated as a result of reaching the execution point `stack`. If `args` is provided, then we unevaluate anything evaluated when applying the procedure at `stack` to arguments different from `args`.
- `save()`: Saves the current state of the database.
- `restore()`: Restores the saved version of the database.
- `reset()`: Resets the database, as if the program has not been executed at all.

Our database also maintains the following variables:

- `count`: The number of XRP applications which aren't outermost noise applications.
- `prob`: The product of all probabilities stored, i.e. the probability of the current execution history.
- `eval_prob`: The product of all probabilities which were newly evaluated since the last save.
- `uneval_prob`: The product of all probabilities which were unevaluated since the last save.

Implementation of this database is not important. The main difficulties are in keeping track of location correctly. We can now give code for `infer`, which runs a single step in the Markov chain:

```
def infer():
  stack = randomDB.random_stack()
  (xrp, args, val, prob, obs) = randomDB.get(stack)

  old_p = randomDB.prob
  old_count = randomDB.count

  randomDB.save()

  randomDB.remove(stack)
  new_val = xrp.apply(args)
  randomDB.insert(stack, xrp, new_val, args)

  rerun()

  new_p = randomDB.prob
  new_to_old_q = randomDB.uneval_prob / randomDB.count
  old_to_new_q = randomDB.eval_prob / old_count

  if new_p * new_to_old_q < old_p * old_to_new_q * random.random():
    randomDB.restore()
```

Here, `rerun()` simply reruns the entire program, using the new randomness in the DB, evaluating new things and unevaluating neglected things as needed. It is not hard to verify that this uses the correct probability of transitioning, according to Metropolis-Hastings.

Technically, we use log probabilities instead of probabilities, as done here, to avoid precision errors. In fact, we require that the `prob()` function of XRPs return the log probability.

4.2 Traces

5 Test Cases

In order to ensure the language was working properly, a suite of test cases was developed.

One of the primary techniques used for testing was "following the prior", using the function `follow_prior(variable, niters, burnin)`. To follow the prior, we draw `niters` samples from the prior, and run each of them `burnin` steps in the random walk. We then view the resulting distribution on `variable`.

Here are some of the more illustrative and interesting test cases.

5.1 Testing a tricky coin

Consider the following scenario: There is a coin that is fair with probability 50%. The rest of the time, its weight is drawn uniformly from the interval [0, 1]. The coin is flipped, and we observe it to be heads `nheads` times. We then infer the posterior probability that the coin was fair.

```
>>>  noise_level = .001
>>>  nheads = 1
>>>
>>>  assume('weight', beta(1, 1))
>>>  assume('tricky-coin', function([], bernoulli('weight')))
>>>  assume('fair-coin', function([], bernoulli(0.5)))
>>>  assume('is-fair', bernoulli(0.5))
>>>  assume('coin', ifelse('is-fair', 'fair-coin', 'tricky-coin'))
>>>
>>>  for i in xrange(nheads):
>>>    observe(bernoulli_noise(apply('coin'), noise_level), True)
>>>
>>>  follow_prior('is-fair', 10000, 1000)
```

We should be able to predict the results of running this program for different values of `nheads`. First we compute the probability that the coin comes up heads n times, given that it is tricky. The probability is simply $\beta(1, n+1) = \frac{1}{n+1}$. Thus the probability the coin is fair (not tricky), given it came up heads n times, is, by Baye's law

$$\frac{\frac{1}{2} \cdot \frac{1}{2^n}}{\frac{1}{2} \cdot \frac{1}{n+1} + \frac{1}{2} \cdot \frac{1}{2^n}} = \frac{n+1}{2^n + n + 1}.$$

Here are our results, when running `follow_prior('is-fair', 10000, 1000)`, for the percentage of times the coin was fair:

`nheads`	Predicted	Actual
0	0.5	0.4982
1	0.5	0.4934
2	0.4286	0.4289
3	0.3333	0.3346
4	0.2381	0.2712
5	0.1579	0.2528

As `nheads` grows, we should expect the mixing time to grow. This is because

5.2 Bayes nets

We first test that inference works, by considering a number of test cases. A classic inference problem which is well understood, is inference in Bayesian networks. Bayesian networks are incredibly easy to describe in our language. Here are some simple examples of inference giving the correct answer in a Bayesian network.

5.2.1 Sprinkler net

Let's start with a simple example, to get familiar with our language. Here's a definition of a bayesian network with just two nodes.

```
>>> assume('cloudy', bernoulli(0.5))
>>> assume('sprinkler', ifelse('cloudy', bernoulli(0.1), bernoulli(0.5)))
```

We then observe that the sprinkler is on. Worlds in which the sprinkler is on are weighted as 100 times more likely than ones in which the sprinkler is off.

```
>>> noise_level = .01
>>> sprinkler_ob = observe(bernoulli_noise('sprinkler', noise_level), True)
```

Now, let's try inferring the weather. We follow the prior 10000 times, going 50 steps each time.

```
>>> follow_prior('cloudy', 10000, 50)}
{False: 0.82010000000000005, True: 0.1799}
```

This is close to the value of $\frac{5}{6} = 0.8\overline{333}$ False, which is what we'd get if there was noise in our observation. However, there is still a number of worlds in which the sprinkler was actually on. In those worlds, the weather was 50/50. Thus we should've expect the answer to be on the order of 0.01 lower.

Now, let's suppose we didn't observe the sprinkler being on, after all.

```
>>> forget(sprinkler_ob)
```

So it should be the case that it's cloudy exactly half the time.

```
>>> follow_prior('cloudy', 10000, 50)}
{False: 0.50429999999999997, True: 0.49569999999999997}
```

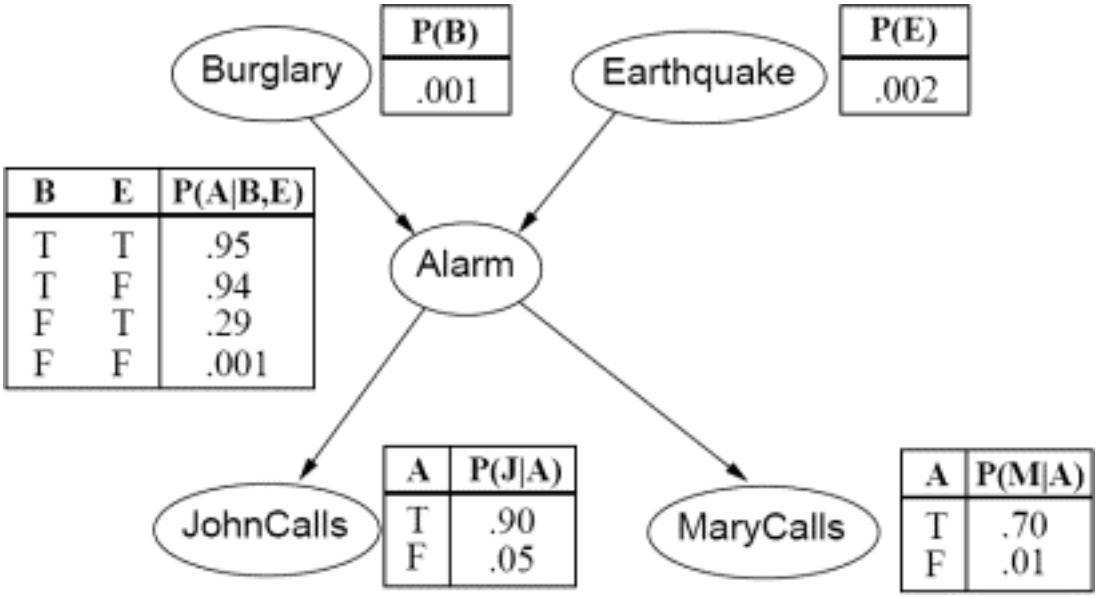
Now, we re-observe the sprinkler to be on. This time, we are much more sure.

```
>>> noise_level = .001
>>> sprinkler_ob = observe(bernoulli_noise('sprinkler', noise_level), True)
>>> follow_prior('cloudy', 10000, 50)}
{False: 0.83999999999999997, True: 0.16}
```

We see that our answer is even closer to the value of $\frac{5}{6} = 0.8\overline{333}$ False, now.

5.2.2 Alarm net

Here's a more complicated Bayesian network, which is given as an example in the classic AI textbook Artificial Intelligence (A Modern Approach).



Let's define this Bayesian network.

```
>>> assume('burglary', bernoulli(0.001))
>>> assume('earthquake', bernoulli(0.002))
>>> assume('alarm', ifelse('burglary', ifelse('earthquake', bernoulli(0.95), bernoulli(0.94)), \
...                                    ifelse('earthquake', bernoulli(0.29), bernoulli(0.001))))
>>> assume('johnCalls', ifelse('alarm',  bernoulli(0.9), bernoulli(0.05)))
>>> assume('maryCalls', ifelse('alarm',  bernoulli(0.7), bernoulli(0.01)))
```

Let's try a couple inference problems.

```
>>> follow_prior('alarm', 1000, 100) # should give 0.002516 True
{False: 0.99739999999999995, True: 0.0025999999999999999}
>>>
>>> noise_level = .001
>>> mary_ob = observe(bernoulli_noise('maryCalls', noise_level), True)
>>> follow_prior('johnCalls', 1000, 100) # should give 0.177577 True
{False: 0.91620000000000001, True: 0.083799999999999999}
>>>
>>> forget(mary_ob)
>>> burglary_ob = observe(bernoulli_noise(negation('burglary'), noise_level), True)
>>> follow_prior('johnCalls', 1000, 100) # should give 0.051343 True
{False: 0.94710000000000005, True: 0.052900000000000003}
```

The first and third inferences were on the mark, but the second one was off by a factor of two! The explanation is that Mary calls very rarely calls. Indeed, she only calls about 0.01 of the time, since the alarm almost never sounds. This is rare enough that our observation that she called is

reasonably likely to be wrong. In worlds where she doesn't call, John also tends not to call, thus accounting for the large dip.

To verify that this is the explanation, we can alter the probabilities so that Mary calling is more likely. Here is an example of this being done.

```
>>> reset()
>>>
>>> assume('burglary', bernoulli(0.1))
>>> assume('earthquake', bernoulli(0.2))
>>> assume('alarm', ifelse('burglary', ifelse('earthquake', bernoulli(0.95), bernoulli(0.94)), \
...                                    ifelse('earthquake', bernoulli(0.29), bernoulli(0.10))))
>>> assume('johnCalls', ifelse('alarm',  bernoulli(0.9), bernoulli(0.5)))
>>> assume('maryCalls', ifelse('alarm',  bernoulli(0.7), bernoulli(0.1)))
>>>
>>> follow_prior('alarm', 1000, 100) # should give 0.218400 True
{False: 0.77959999999999996, True: 0.22040000000000001}
>>>
>>> noise_level = .001
>>> mary_ob = observe(bernoulli_noise('maryCalls', noise_level), True)
>>> follow_prior('johnCalls', 1000, 100) # should give 0.764681 True
{False: 0.2432, True: 0.75680000000000003}
>>>
>>> forget(mary_ob)
>>> burglary_ob = observe(bernoulli_noise(negation('burglary'), noise_level), True)
>>> follow_prior('johnCalls', 1000, 100) # should give 0.561333 True
{False: 0.44290000000000002, True: 0.55710000000000004}
```

Success!

5.3 Testing xor

Consider the following program, where we simply draw two booleans `a` and `b`, and observe that their xor is `True`:

```
>>> p, q = 0.6, 0.4
>>> noise_level = .01
>>>
>>> assume('a', bernoulli(p))
>>> assume('b', bernoulli(q))
>>> assume('c', var('a') ^ var('b'))
>>>
>>> follow_prior('a', 10000, 100) # should be 0.60 True
{False: 0.3977, True: 0.60229999999999995}
>>>
>>> xor_ob = observe(bernoulli_noise('c', noise_level), True)
>>> follow_prior('a', 10000, 100) # should be 0.69 True
{False: 0.3342, True: 0.66579999999999995}
```

This second inference is significantly off, and it's not the case that our observation is particularly unlikely. Here, the burn-in was not enough. Notice that in order to mix, the program must walk over states which contradict the observed values.

Suppose we are in the state {a:True, b:True}. Then, the random walk is highly discouraged from entering either of the two adjacent states {a:True, b:False} and {a:False, b:True}, since worlds in which a⊕b is False are weighted against, by a factor of 100.

Let's estimate the amount of burn-in it should take for this random walk to mix. Suppose we are in a state where a⊕b is True. About half the time, we will attempt to enter one of the two adjacent states (the other half of the time, we will keep the same value for that flip). Entering this state will occur with probability roughly $\frac{1}{100}$. Once we are in such a state, we are given an opportunity to enter either of the two a⊕b being True states, so we have successfully mixed.

Thus we can roughly model this as having a $\frac{1}{200}$ chance of mixing properly, for each iteration. Thus if we have a burn-in of $200 \cdot x$, there is roughly a $1 - \frac{1}{e^x}$ chance of mixing. Here are empirical results of burnin against follow_prior('a', 10000, burnin) results:

burnin	follow_prior('a', 10000, burnin) percentage True
100	0.66579999999999995
200	0.67869999999999997
300	0.68200000000000005
400	0.68489999999999995
500	0.68759999999999999

These results are slightly better than our prediction, but confirm the overall phenomenon. Because of the noise, we can't expect it to ever converge to exactly 0.69. Instead, we may hope for it to converge to something like $0.01 \cdot 0.60 + 0.99 \cdot 0.69 = 0.6891$.

5.4 Testing a decaying atom

Suppose we are observing a radioactive atom in discrete time intervals, e.g. we look at it each second. Each time we look at the atom, there is some chance p that it decays, called its decay rate.

Suppose there is an atom whose decay rate we don't know. We then observe that it decays in the nth time interval. Suppose that the prior over decay rates is the distributed as $\text{Be}(\alpha, \beta)$, the beta distribution with parameters α and β. Then a simple calculation shows the posterior distribution for decay rates should now be $\text{Be}(\alpha + 1, \beta + n - 1)$.

```
>>> assume('decay', beta(1, 1))
>>> assume('geometric', function('x', ifelse(bernoulli('decay'), 'x', apply('geometric', var('x') + 1))))
>>> observe(bernoulli_noise(apply('geometric', 0) == 10, .01), True)
  print get_pdf(follow_prior('decay', 100, 100), 0, 1, .1)
```

5.5 Testing mem

Here, we test that the memoization procedure works.
Let's first write the fibonacci function, in the naive way:

```
>>> fibonacci_expr = function('x', ifelse(var('x')<=1, 1, \
...                   apply('fibonacci', var('x')-1) + apply('fibonacci', var('x')-2)))
>>> assume('fibonacci', fibonacci_expr)
```

Of course, evaluating fibonacci in this manner is an exponential time operation:

```
>>> t = time(); sample(apply('fibonacci', 20)); time() - t
10946
1.76103687286
```

Mem is an XRP which, when applied to (possibly probabilistic) functions, returns a version of the function which is memoized. That is, function calls are remembered, so that if a function is called with the same arguments, it does not need to recompute. Here is an example of mem being used.

```
>>> assume('bad_mem_fibonacci', mem('fibonacci'))
>>>
>>> t = time(); sample(apply('bad_mem_fibonacci', 20)); time() - t
10946
1.90201187134
>>> t = time(); sample(apply('bad_mem_fibonacci', 20)); time() - t
10946
0.00019097328186
```

Notice that the second call to this mem'd fibonacci is much faster, since mem remembers the value. However, the first call is just as slow as before. Since Fibonacci is recursive, we really want to memoize all the recursive subcalls as well, making the Fibonacci function a linear time computation. We can write the program without much trouble, using mem:

```
>>> mem_fibonacci_expr = function('x', ifelse(var('x')<=1, 1, \
...                     apply('mem_fibonacci', var('x')-1) + apply('mem_fibonacci', var('x')-2)))
>>> assume('mem_fibonacci', mem(mem_fibonacci_expr))
>>>
>>> t = time(); sample(apply('mem_fibonacci', 20)); time() - t
10946
0.0271570682526
>>> t = time(); sample(apply('mem_fibonacci', 20)); time() - t
10946
0.000293016433716
```

5.6 Testing DPMem

Let's now implement the Dirichlet process.

```
>>> assume('DP', \
...        function(['concentration', 'basemeasure'], \
...                 let([('sticks', mem(function('j', beta(1, 'concentration')))),
...                      ('atoms',  mem(function('j', apply('basemeasure')))),
...                      ('loop', \
...                       function('j', \
...                                ifelse(bernoulli(apply('sticks', 'j')), \
...                                       apply('atoms', 'j'), \
...                                       apply('loop', var('j')+1))))], \
...                     function([], apply('loop', 1)))))
```

We can now use the Dirichlet process to create something we call DPmem. DPmem is a generalization of mem which sometimes returns memoized values, and sometimes resamples new values.

```
  """DEFINITION OF DPMEM"""

>>>  assume('DPmem', \
...         function(['concentration', 'proc'], \
...                  let([('restaurants', \
...                        mem(function('args', \
...                                     apply('DP', ['concentration',
...                                                  function([], apply('proc', 'args'))])))])], \
...                      function('args', apply('restaurants', 'args')))))


  """TESTING DPMEM"""

  concentration = 1 # when close to 0, just mem.  when close to infinity, sample
  assume('DPmemflip', apply('DPmem', [concentration, function(['x'], bernoulli(0.5))]))
  print [sample(apply('DPmemflip', 5)) for i in xrange(10)]

  print "\n TESTING GAUSSIAN MIXTURE MODEL\n"
  assume('alpha', gaussian(1, 0.2)) # use vague-gamma?
  assume('expected-mean', gaussian(0, 1))
  assume('expected-variance', gaussian(0, 1))
  assume('gen-cluster-mean', apply('DPmem', ['alpha', function(['x'], gaussian(0, 1))]))
  assume('get-datapoint', mem( function(['id'], gaussian(apply('gen-cluster-mean', 222), 1.0))))
  assume('outer-noise', gaussian(1, 0.2)) # use vague-gamma?

  observe(gaussian(apply('get-datapoint', 0), 'outer-noise'), 1.3)
  observe(gaussian(apply('get-datapoint', 1), 'outer-noise'), 1.2)
  observe(gaussian(apply('get-datapoint', 2), 'outer-noise'), 1.1)
  observe(gaussian(apply('get-datapoint', 3), 'outer-noise'), 1.15)

  t = time()
  print format(get_pdf(follow_prior('expected-mean', 100, 30), -4, 4, .5), '%0.2f')
  print 'time taken', time() - t

  #concentration = 1
  #uniform_base_measure = uniform_no_args_XRP(2)
  #print [sample(apply('DP', [concentration, uniform_base_measure])) for i in xrange(10)]
  #expr = beta_bernoulli_1()
```

DPmem can be used, for example, to do mixture modeling. The idea is that we should sometimes

References