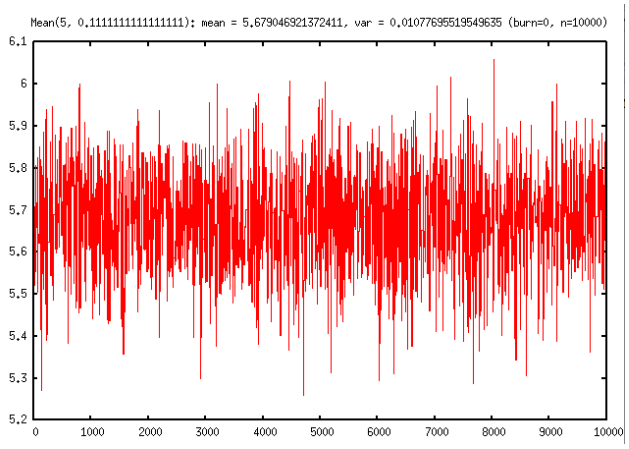
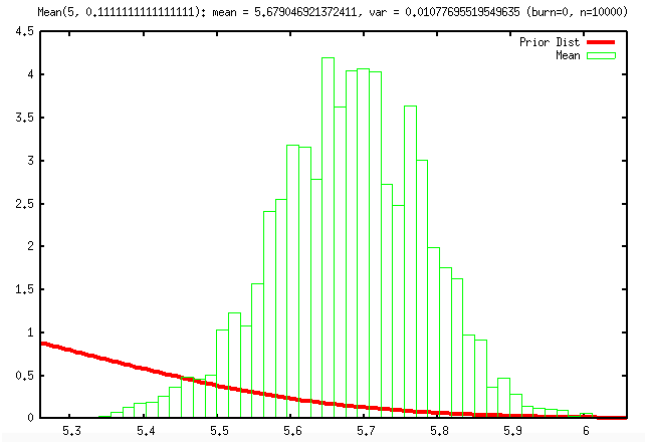
Nathan Hadfield
CS 677 (Spring 2014)

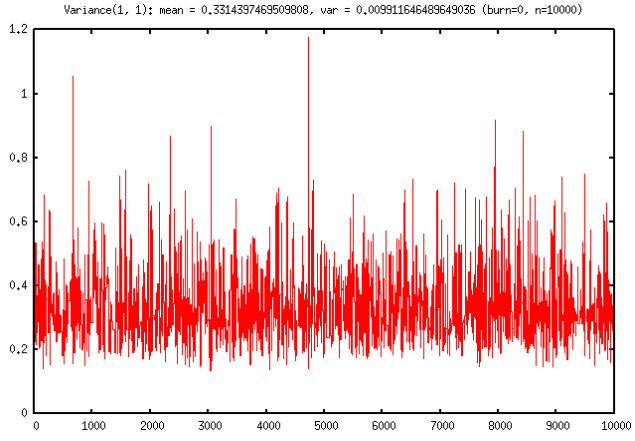
MCMC Lab #2

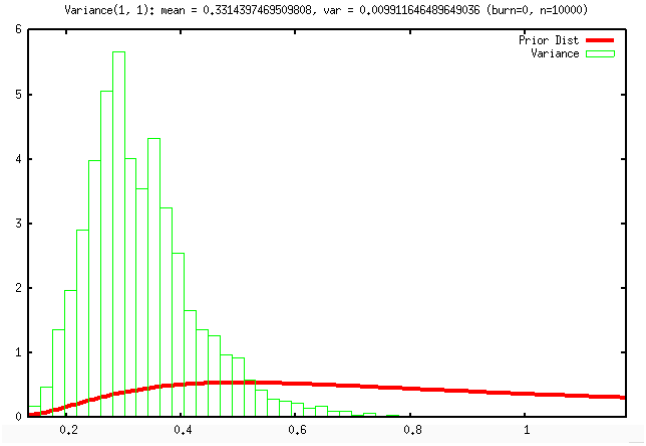
Code used set up each section of the lab is included below the charts. Code for implementing the nodes and the network is provided at the end.

Faculty Evaluations









network_faculty.py

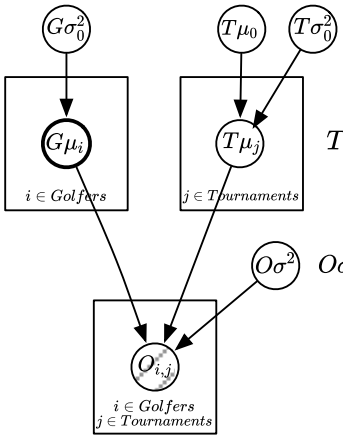
```
import numpy
from node_normal import *
from node_invgamma import *
from network import *

datafilename = 'faculty.dat'
nsamples = 10000
burn = 0
mean_candsd = 0.2
var_candsd = 0.15

# Read in Data
data = [float(line) for line in open(datafilename)]

# Use point estimators from the data to come up with starting values.
estimated_mean = numpy.mean(data)
estimated_var = numpy.var(data)

def MomentsInvGammaShape(mean, var):
    return 1
```

```
def MomentsInvGammaScale(mean,var):
    return 1

# Create Nodes and Links in Network
meannode = NormalNode(estimated_mean, name='Mean', cand_var=mean_candsd, mean=5, var=(1/3)**2)
varprior_mean = 1/4
varprior_stddev = 1/12
varprior_shape = MomentsInvGammaShape(varprior_mean, varprior_stddev**2)
varprior_scale = MomentsInvGammaScale(varprior_mean, varprior_stddev**2)
varnode = InvGammaNode(estimated_var, name='Variance', cand_var=var_candsd, shape=varprior_shape, scale=varprior_scale)
for datum in data:
    NormalNode(datum, observed=True, mean=meannode, var=varnode)

# Perform simulations and plot results
network = Network([meannode, varnode])
samples = network.collect_samples(burn, nsamples)


def mean_prior_pdf(x):
    return stats.norm.pdf(x, 5, 1/3)

def var_prior_pdf(x):
    return stats.invgamma.pdf(x, a=varprior_shape, scale=varprior_scale)

prior_pdfs = { meannode: mean_prior_pdf, varnode: var_prior_pdf }

results = {}
for node in [meannode, varnode]:
    params = {
        'mean': numpy.mean(samples.of_node(node)),
        'var': numpy.var(samples.of_node(node))
    }
    results[node] = params

    title = "{}: mean = {}, var = {} (burn={}, n={})". \
format(node.pdf_name, params['mean'], params['var'], burn, nsamples - burn)
    samples.plot_node(node, title=title)
    if params['var'] > 0:         # histogram fails if all values are the same
        samples.plot_histogram_for_node(node, title=title, prior_pdf=prior_pdfs[node])
```

Professional Golfers

$$T\mu_0^2 = N(72, 2)$$

$$G\sigma_0^2 = IGamma(18, 1/0.015) \qquad T\sigma_0^2 = IGamma(18, 1/0.015)$$



$$G\mu_i = N(0, G\sigma_0^2)$$

$$T\mu_j = N(T\mu_0, T\sigma_0^2)$$

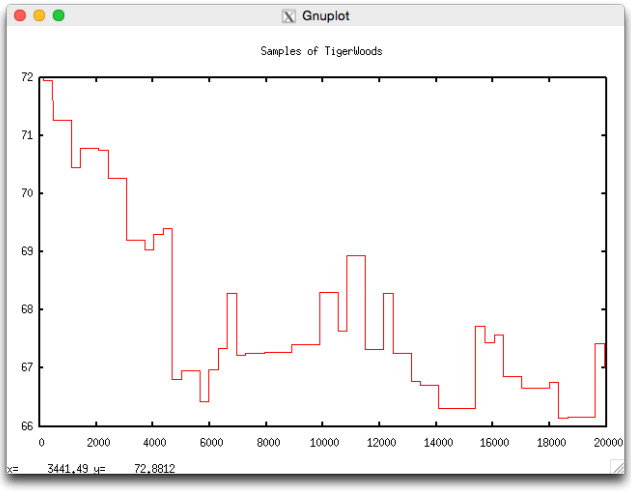
$$O\sigma^2 = IGamma(83, 1/0.0014)$$

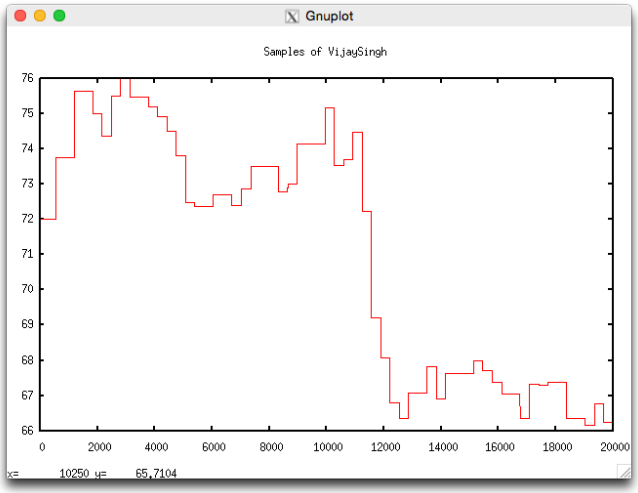
$$O_{i,j} = N(G\mu_i + T\mu_j, O\sigma^2)$$

$T\mu_j$ is the average score for Tournament j. $G\mu_i$ is the average skill of Golfer j (i.e., how much better they score than the average golfer). Given a set of golf scores for several tournaments, we are interested in obtaining the average skill of the golfers ($G\mu_i$).

Initially, my results were way off because I was using the same initial value for my golfer means as I was using for my tournament means (which I based blindly on the sample code provided, which used a variable named "est_avg" for both sets of nodes), whereas the golfer means should actually have been *differences* from the average, and an initial value close to 0 would have been much better. With an initial value for golfer means of 72, it would have taken a lot more samples than I was generating in order to get to a valid steady state, as illustrated by the following mixing plots of the golfer means of TigerWoods and VijaySingh:

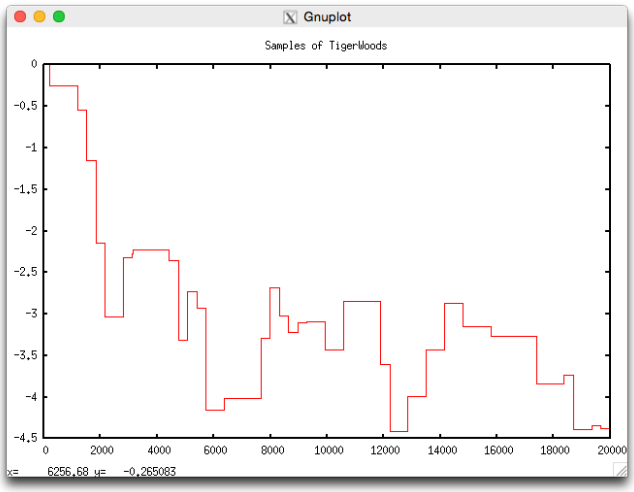
Initial golfermean value = 72:

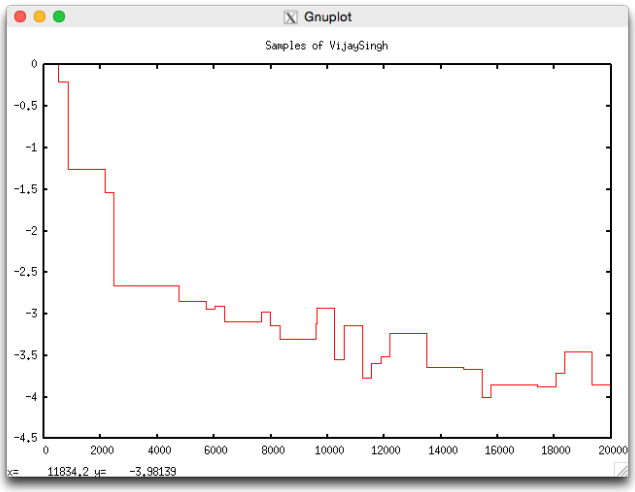




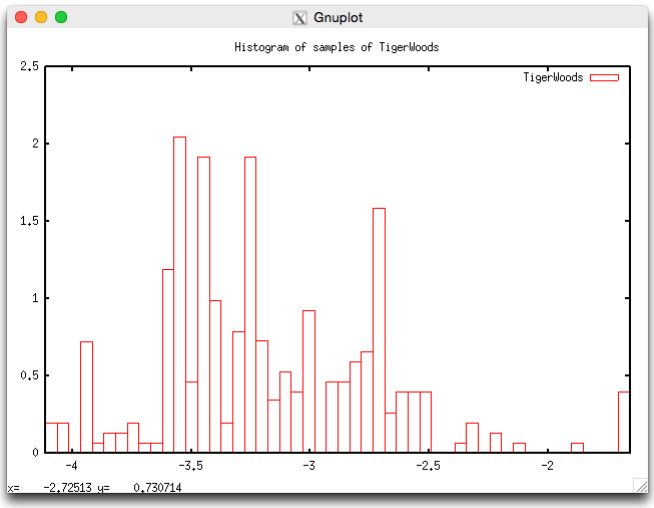
Setting the initial value of all golfer means to 0 significantly improved the results. With this initial value, the following mixing plots show that a good burn value is probably somewhere around 10,000.

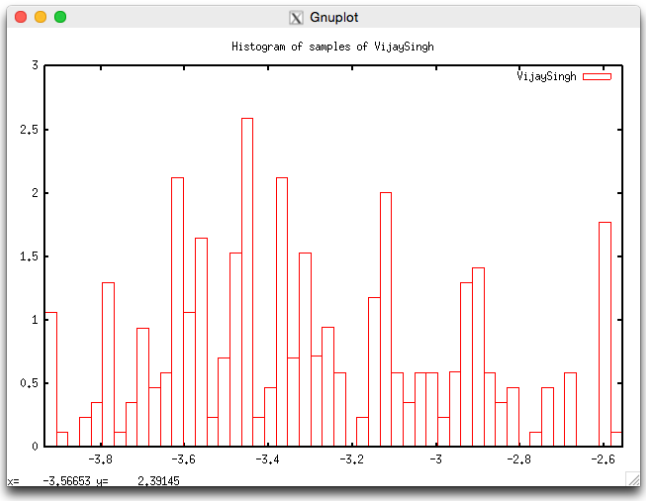
Initial golfermean value = 0:





Here are histograms of 100,000 samples of the golfer means for TigerWoods and VijaySingh, taken after 10,000 burned samples. The values are much better, and the distributions are beginning to look somewhat Gaussian. Still, there is a significant amount of variance in the samples.





Here are the rankings I obtained with 100,000 samples after a burn of 10,000:

1: VijaySingh -3.358989; 90% interval: (-3.787266, -2.602100)
2: TigerWoods -3.244745; 90% interval: (-3.942376, -2.531568)
3: PhilMickelson -2.800336; 90% interval: (-3.434281, -1.996117)
4: ErnieEls -2.773805; 90% interval: (-3.683183, -1.937580)
5: StewartCink -2.404671; 90% interval: (-3.085598, -1.819585)
6: SergioGarcia -2.318692; 90% interval: (-3.089609, -1.649872)
7: ScottVerplank -2.273680; 90% interval: (-2.977398, -1.656846)
8: JayHaas -2.231404; 90% interval: (-2.923095, -1.636209)
9: StephenAmes -2.148486; 90% interval: (-2.749085, -1.519959)
10: PadraigHarrington -2.141603; 90% interval: (-3.235765, -0.972317)
…

594: DavidCarr 3.576623; 90% interval: (1.619768, 5.912245)
595: MattLoving 3.577810; 90% interval: (1.700474, 5.970935)
596: CharlesCoody 3.738503; 90% interval: (1.513889, 5.674814)
597: DaveEichelberger 3.793106; 90% interval: (0.439146, 5.877481)
598: RobertDeruntz 3.839784; 90% interval: (1.803449, 6.202557)
599: LorenPersonett 3.866243; 90% interval: (2.116492, 5.170821)
600: TommyAaron 4.082453; 90% interval: (2.121717, 6.472951)
601: KevinSavage 4.763779; 90% interval: (2.779024, 6.713335)
602: JohnAber 4.804098; 90% interval: (2.951387, 6.441183)
603: DerekSanders 5.091835; 90% interval: (3.123192, 7.004599)
604: ArnoldPalmer 5.994113; 90% interval: (4.442667, 7.457914)

network_golfers.py

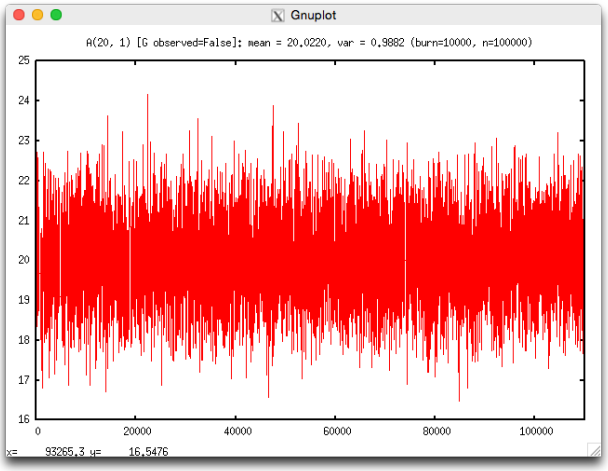
```
from node_normal import *
from node_invgamma import *
from network import *
from operator import itemgetter

data = []
for line in open('golfdataR.dat'):
    line_data = line.strip().split(' ')
    line_data[1] = float(line_data[1])              # parse the score value as a float
    data.append(line_data)

golfers = sorted(set([line[0] for line in data]))
tourns = sorted(set([line[2] for line in data]), key=int)

# For candidate distributions, we use a Normal with mean 0, variance 1
hypertournmean_candsd = 1  # variance
hypervar_candsd = 1  # variance
mean_candsd = 1  # variance
obsvar_candsd = 1  # variance

hypertournmean = NormalNode(72, name='Tournament Hyper Mean', cand_var=hypertournmean_candsd, mean=72, var=2)
hypertournvar = InvGammaNode(3.5, name='Tournament Hyper Var', cand_var=hypervar_candsd, shape=18, scale=1 / .015)
tournmean = {}
for tourn in tourns:
    tournmean[tourn] = NormalNode(72, name="Tournament {}".format(tourn), cand_var=mean_candsd,
                                  mean=hypertournmean, var=hypertournvar)

hypergolfervar = InvGammaNode(3.5, name='Golfer Hyper Var', cand_var=hypervar_candsd, shape=18, scale=1/.015)
golfermean = {}
for golfer in golfers:
    golfermean[golfer] = NormalNode(0, name=golfer, cand_var=mean_candsd, mean=0, var=hypergolfervar)

obsvar = InvGammaNode(8.5, name='Observation Var', cand_var=obsvar_candsd, shape=83, scale=1/.0014)
for (name, score, tourn) in data:
    NormalNode(score, observed=True, mean=[tournmean[tourn], golfermean[name]], var=obsvar)

# sample from nodes
burn = 10000
nsamples = 10000

network = Network(
    [hypertournmean, hypertournvar, hypergolfervar, obsvar] + list(tournmean.values()) + list(golfermean.values()))
samples = network.collect_samples(burn, nsamples)

samples.plot_node(golfermean['VijaySingh'])
samples.plot_node(golfermean['TigerWoods'])
samples.plot_histogram_for_node(golfermean['VijaySingh'])
samples.plot_histogram_for_node(golfermean['TigerWoods'])

ability = []
for golfer in golfermean:
    golfermean_samples = samples.of_node(golfermean[golfer])[:]
    golfermean_samples.sort()
    median = golfermean_samples[nsamples // 2]
    low = golfermean_samples[int(.05 * nsamples)]
    high = golfermean_samples[int(.95 * nsamples)]
    ability.append((golfer, low, median, high))

ability = sorted(ability, key=itemgetter(2))  # sort by median score
i = 1
for golfer, low, median, high in ability:
    print("{}: {} {:.6f}; 90% interval: ({:.6f}, {:.6f})".format(i, golfer, median, low, high))
    i += 1
```

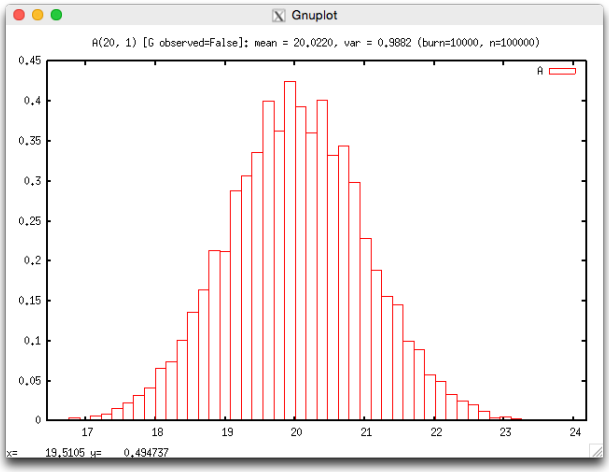
Wacky Network

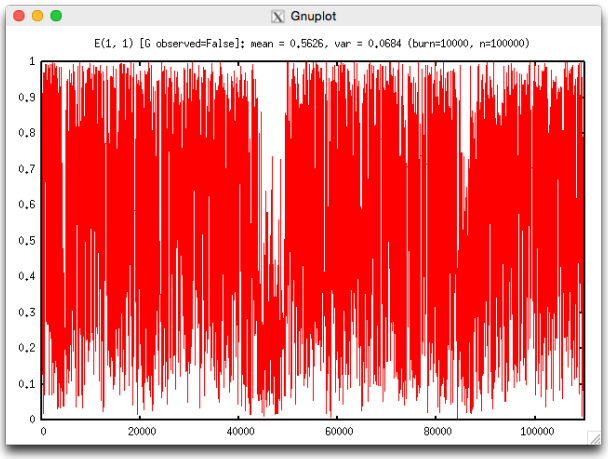
It took me a couple of tries to get my Bernoulli distribution implemented correctly, as I confused the sampling value used for the candidate with the probability of the candidate.

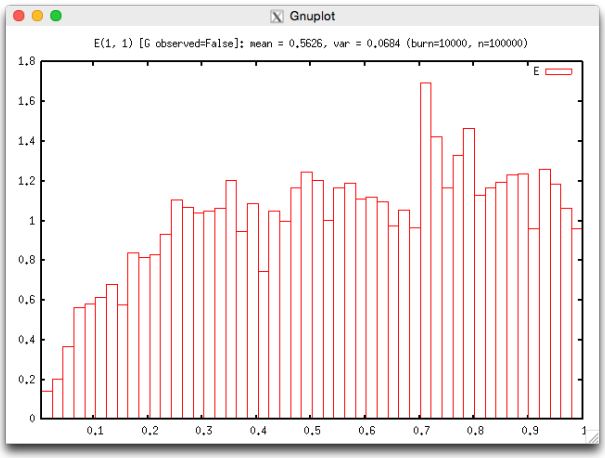
When G is observed, the most obvious difference is in the Poisson distribution of node F, which is much more likely to have values of 2, 3, and 4, and less likely to have a value of 1.

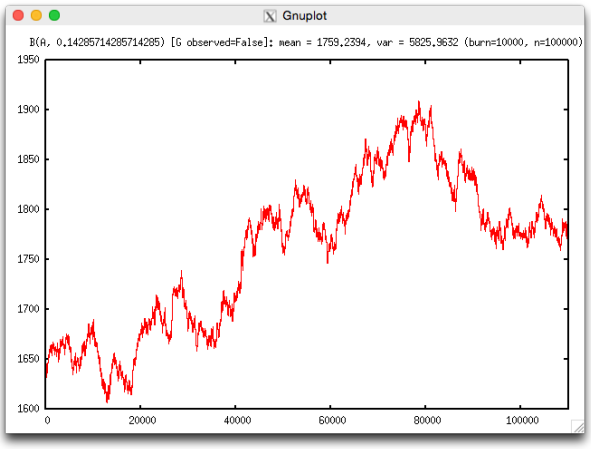
With no observations:

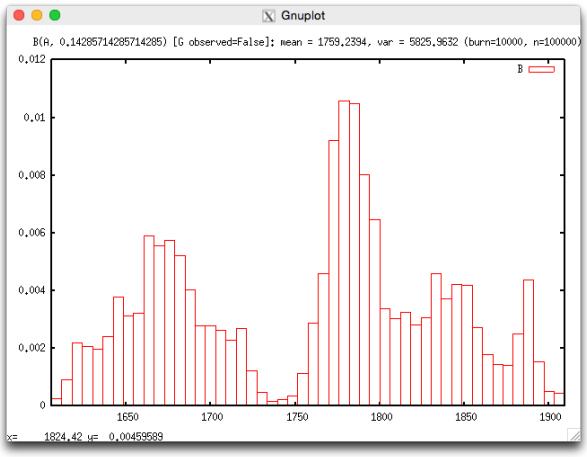


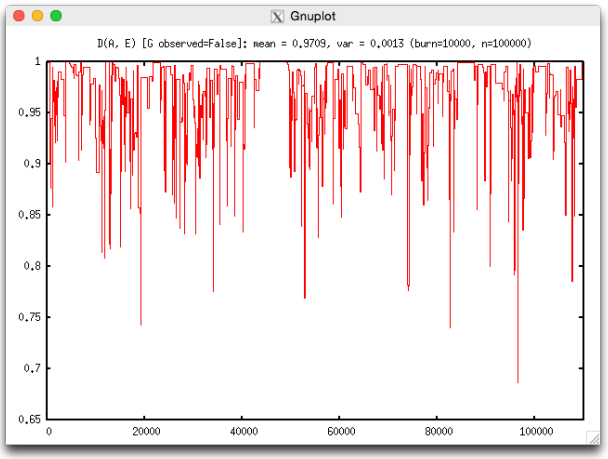


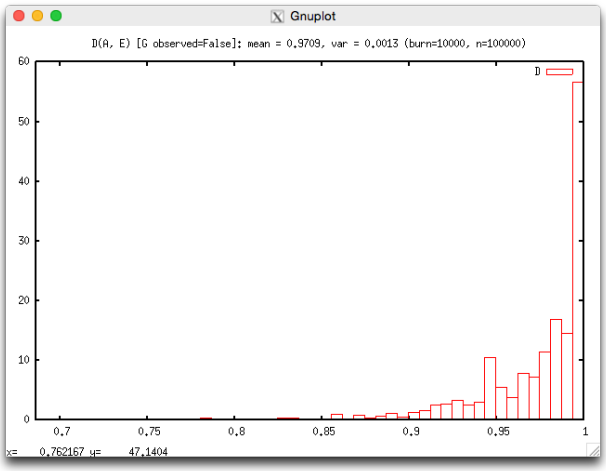


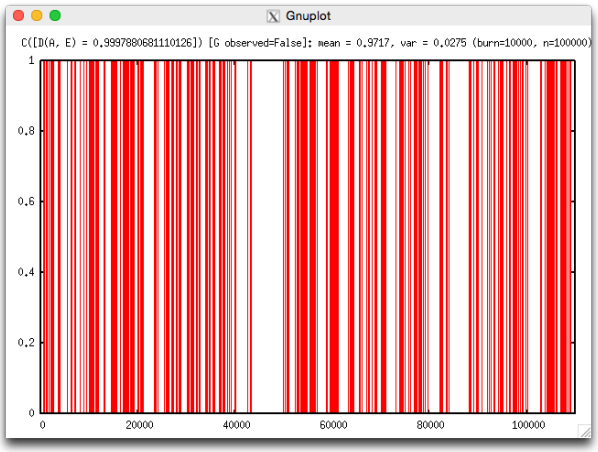


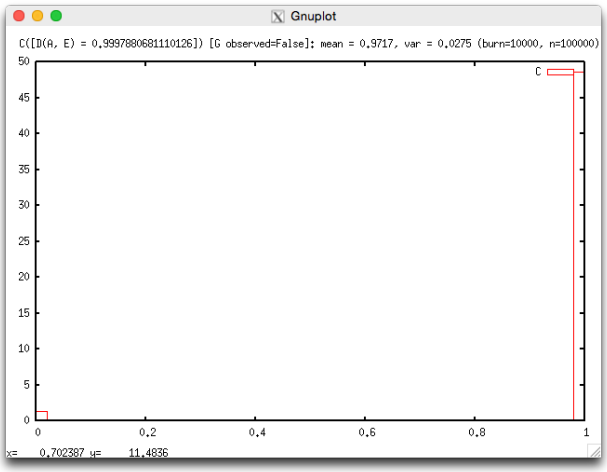


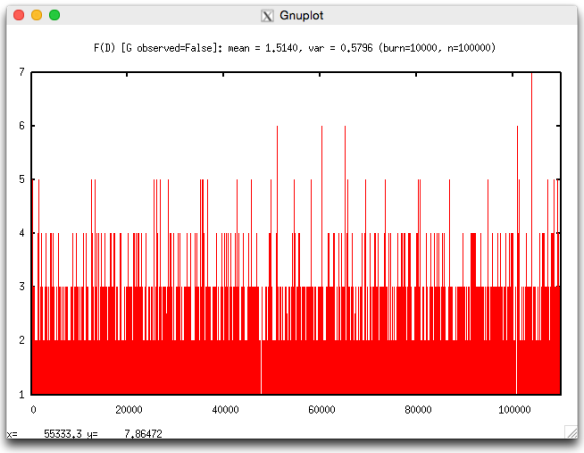


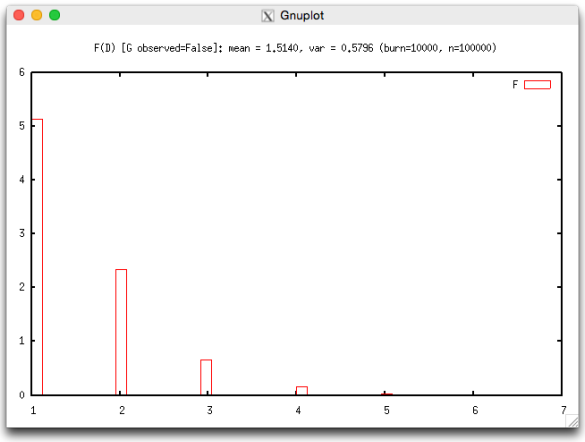

Gnuplot
D(A, E) [G observed=False]: mean = 0.9709, var = 0.0013 (burn=10000, n=100000)

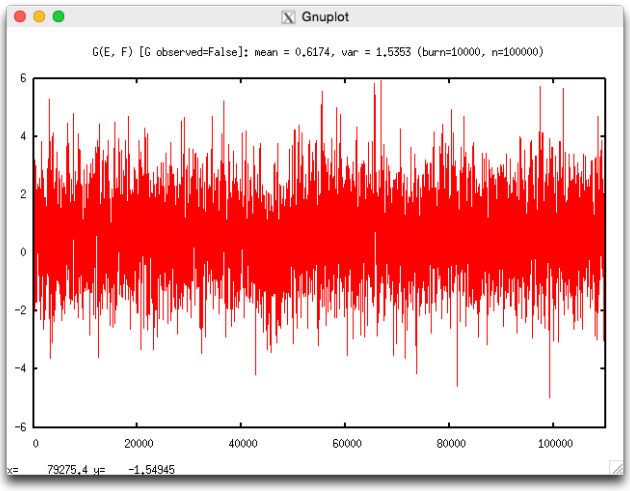

Gnuplot
D(A, E) [G observed=False]: mean = 0.9709, var = 0.0013 (burn=10000, n=100000)
D
x= 0.762167 y= 47.1404

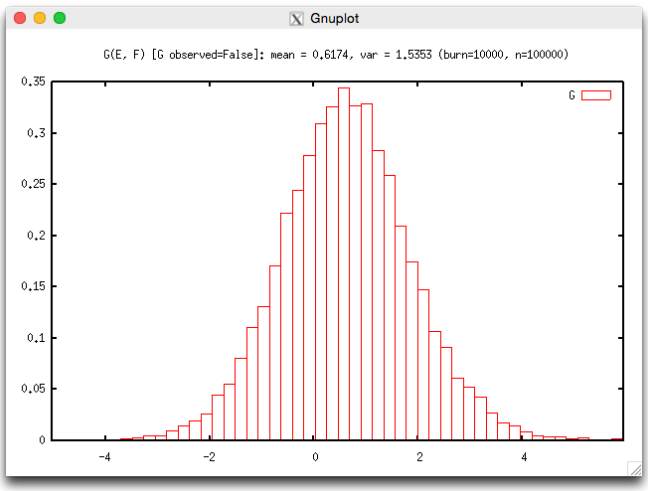

Gnuplot
C([D(A, E) = 0.9997880681110126]) [G observed=False]: mean = 0.9717, var = 0.0275 (burn=10000, n=100000)


Gnuplot
C([D(A, E) = 0.9997880681110126]) [G observed=False]: mean = 0.9717, var = 0.0275 (burn=10000, n=100000)
C
x= 0.702387 y= 11.4836

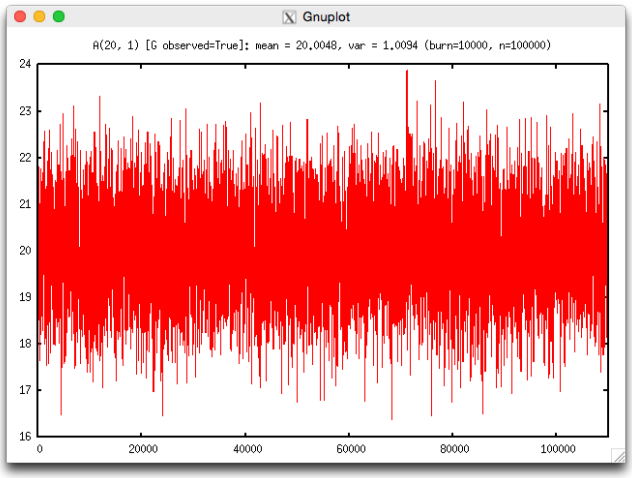

Gnuplot
F(D) [G observed=False]: mean = 1.5140, var = 0.5796 (burn=10000, n=100000)
x= 55333.3 y= 7.86472

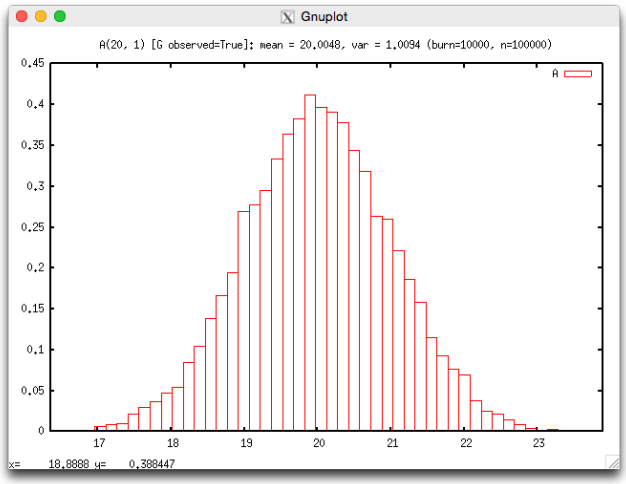

Gnuplot
F(D) [G observed=False]: mean = 1.5140, var = 0.5796 (burn=10000, n=100000)
F

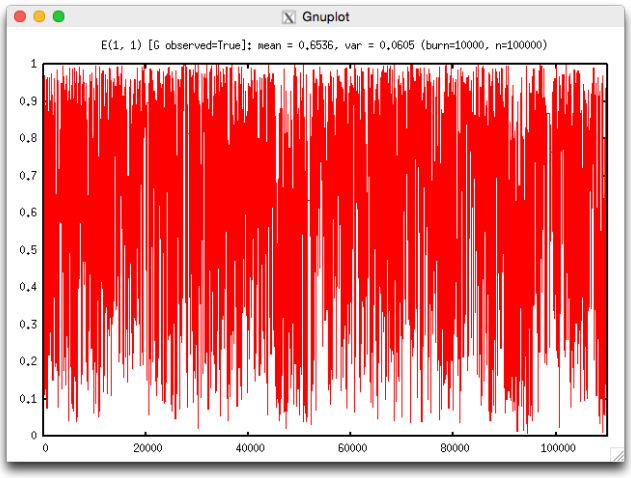


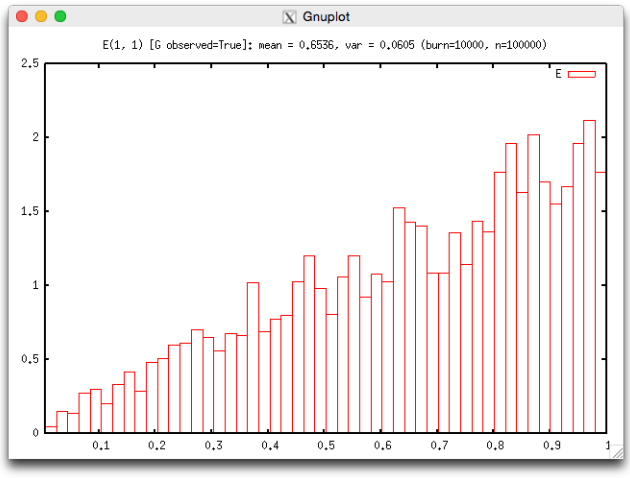


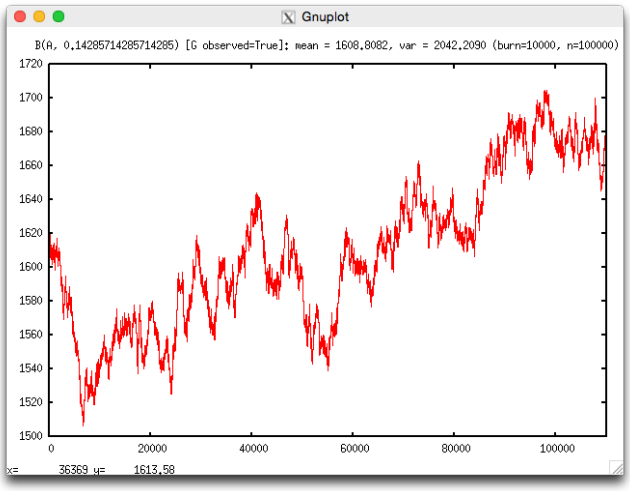
With G observed to be 5:

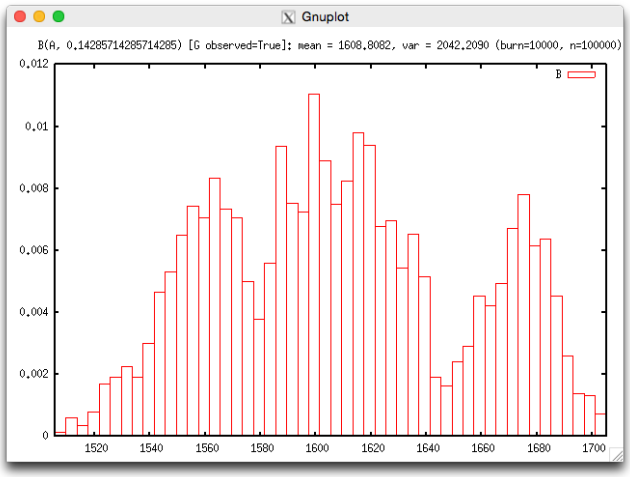


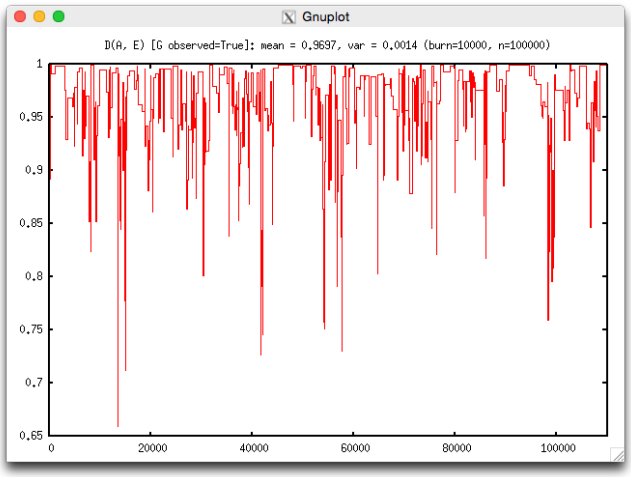


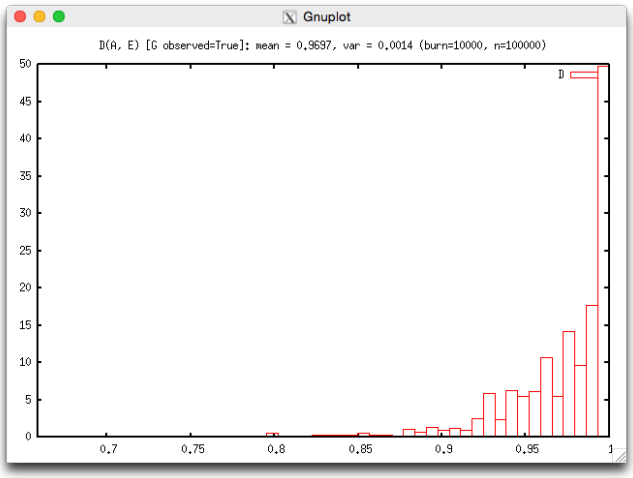


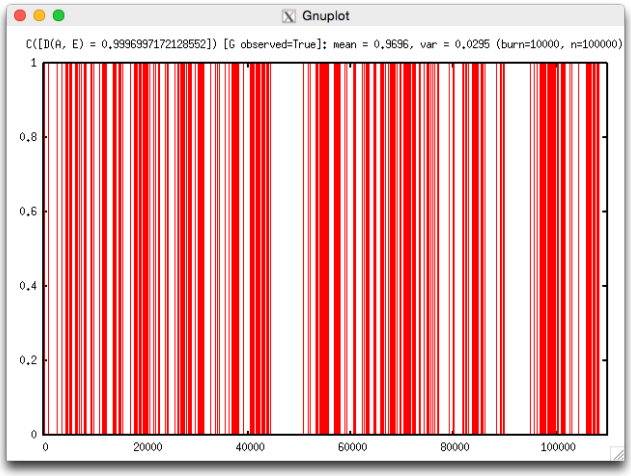


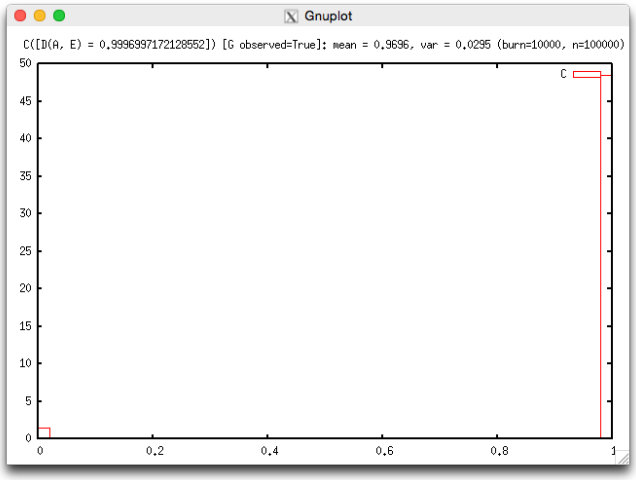

Gnuplot
B(A, 0.14285714285714285) [G observed=True]: mean = 1608.8082, var = 2042.2090 (burn=10000, n=100000)
x= 36369 y= 1613.58

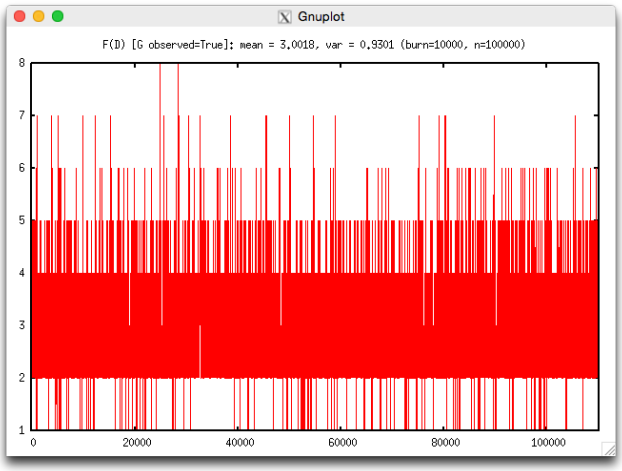

Gnuplot
B(A, 0.14285714285714285) [G observed=True]: mean = 1608.8082, var = 2042.2090 (burn=10000, n=100000)
B

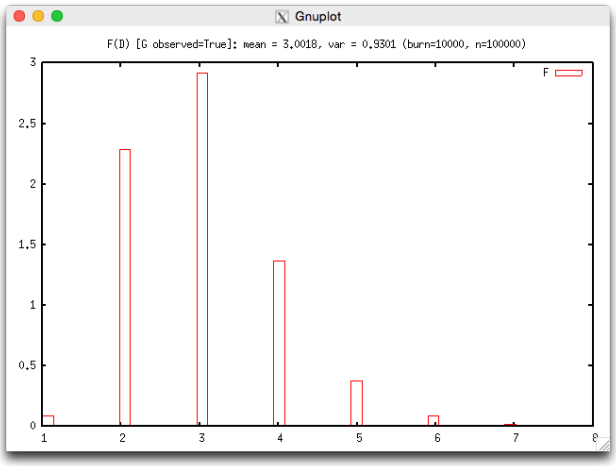

Gnuplot
D(A, E) [G observed=True]: mean = 0.9697, var = 0.0014 (burn=10000, n=100000)

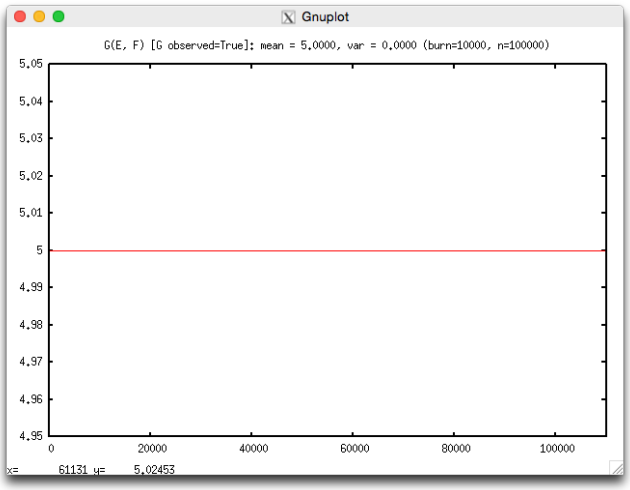

Gnuplot
D(A, E) [G observed=True]: mean = 0.9697, var = 0.0014 (burn=10000, n=100000)
D


Gnuplot
C([D(A, E) = 0.9996997172128552]) [G observed=True]: mean = 0.9696, var = 0.0295 (burn=10000, n=100000)


Gnuplot
C([D(A, E) = 0.9996997172128552]) [G observed=True]: mean = 0.9696, var = 0.0295 (burn=10000, n=100000)
C







(Range is empty; can't plot a histogram.)

network_wacky.py

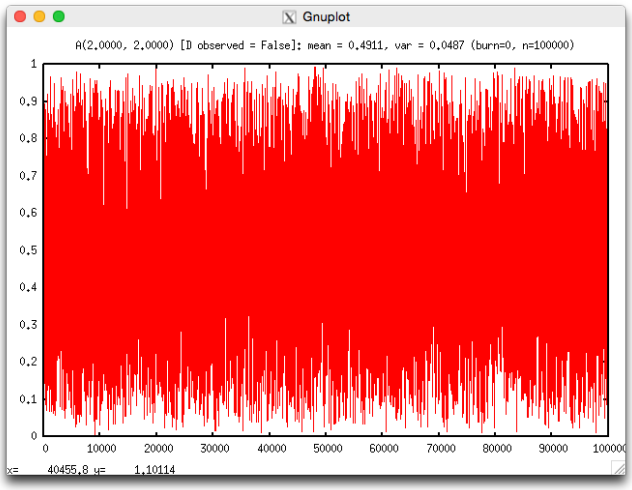
```
from node_normal import *
from node_beta import *
from node_gamma import *
from node_poisson import *
from node_bernoulli import *
from network import *
import numpy

logging.basicConfig(level=logging.WARNING,
     format='[%(levelname)s] %(module)s %(funcName)s(): %(message)s')


burn = 0
num_samples = burn + 100000

for g_observed in [False, True]:
    a = NormalNode(20, 'A', mean=20, var=1)
    e = BetaNode(0.5, 'E', alpha=1, beta=1)
    b = GammaNode(0.2, 'B', shape=a, shape_modifier=lambda x: x ** math.pi, scale=1/7)
    d = BetaNode(0.5, 'D', alpha=a, beta=e)
    c = BernoulliNode(0, 'C', p=d)
    f = PoissonNode(4, 'F', rate=d)
    g = NormalNode(5, 'G', mean=e, var=f, observed=g_observed)

    network = Network([a, e, b, d, c, f, g])
    samples = network.collect_samples(burn=burn, n=num_samples)

    for node in network.nodes:
        mean = numpy.mean(samples.of_node(node))
        var = numpy.var(samples.of_node(node))
        title = "{} [G observed={}]: mean = {:.4f}, var = {:.4f} (burn={}, n={})"
.format(node.pdf_name, g_observed, mean, var, burn, num_samples-burn)
        samples.plot_node(node, title=title)
        samples.plot_histogram_for_node(node, title=title)
```

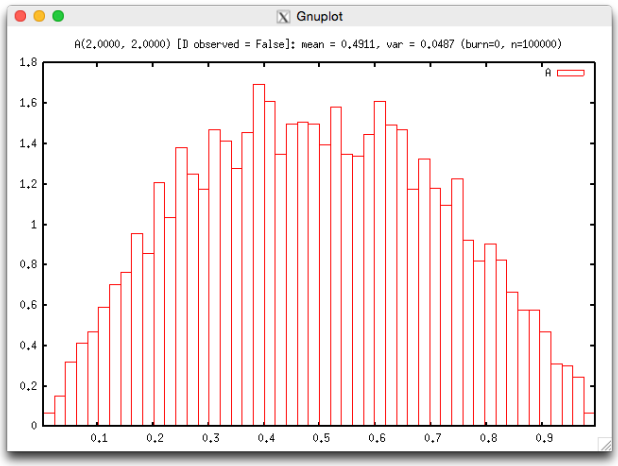
My Network

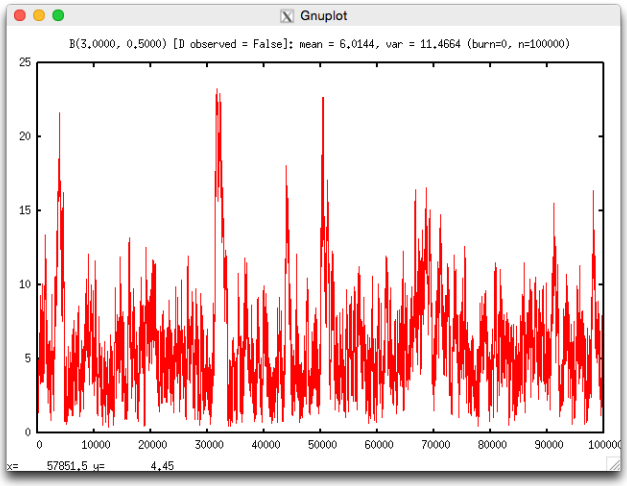
What happens to the network when D is observed at 5?

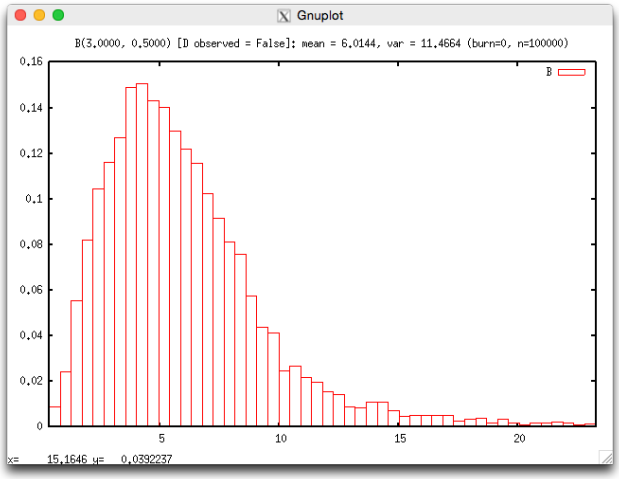
```
a = BetaNode(0.4, 'A', alpha=2, beta=2)
b = GammaNode(4, 'B', shape=3, scale=1/2)
c = NormalNode(0, 'C', mean=b, var=a)
d = PoissonNode(5, 'D', rate=b, observed=d_observed)
```

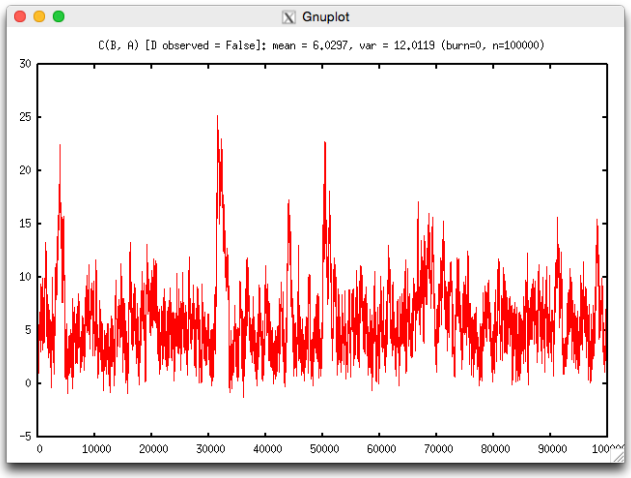
D not observed:

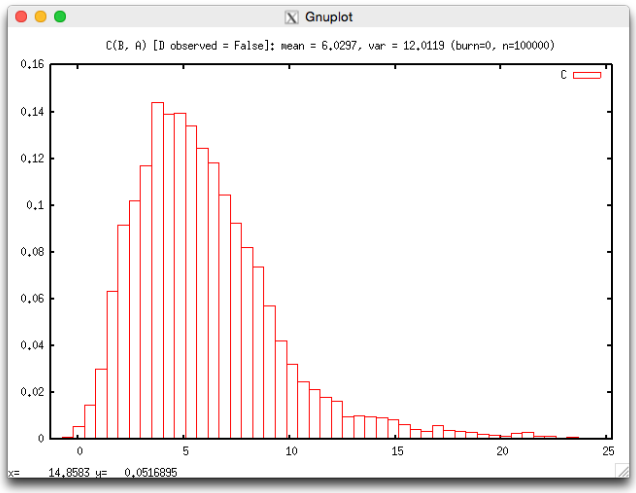


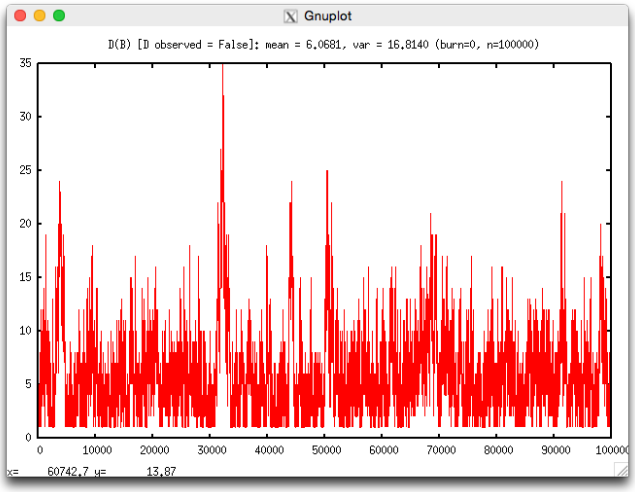


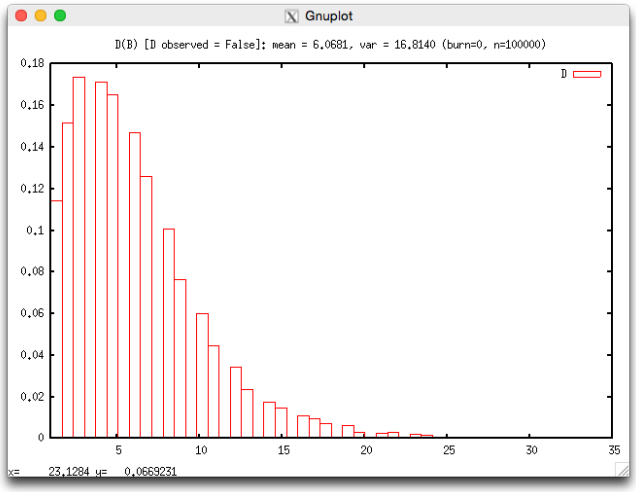




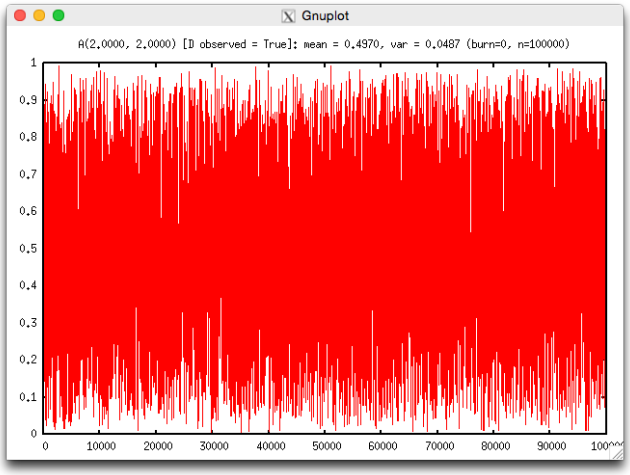


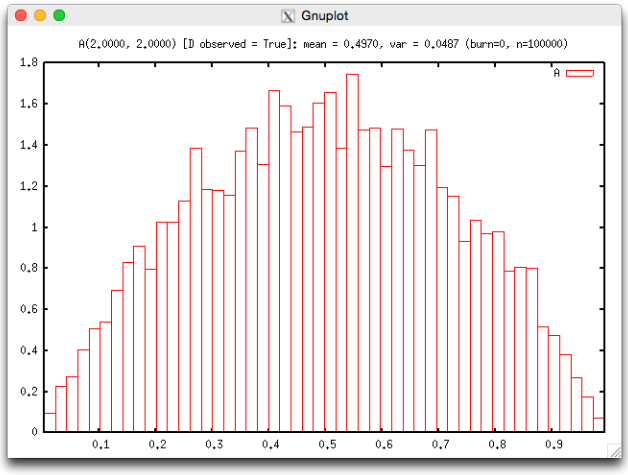


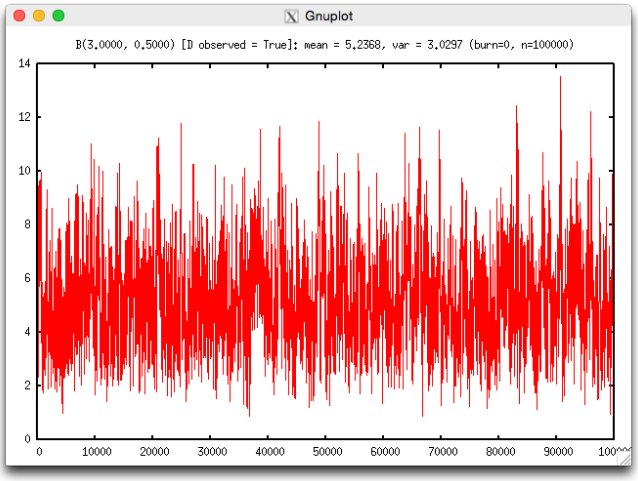


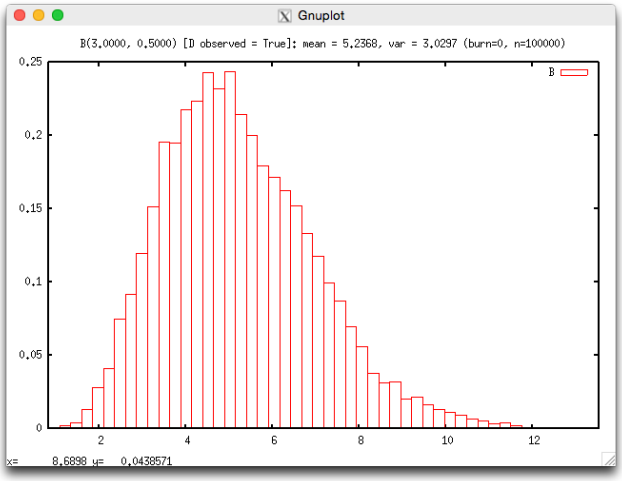


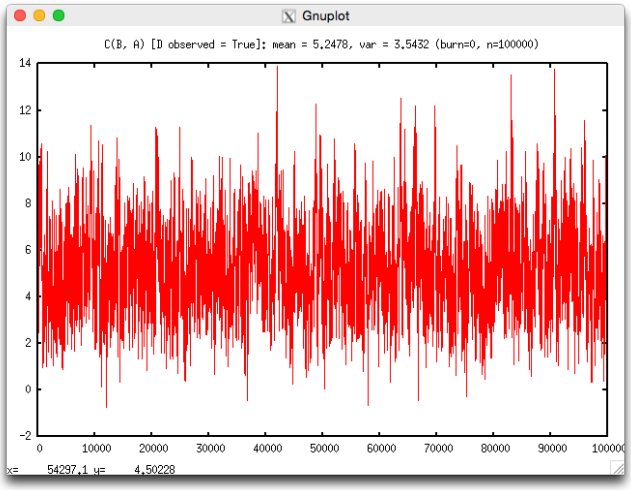
D observed at 5:

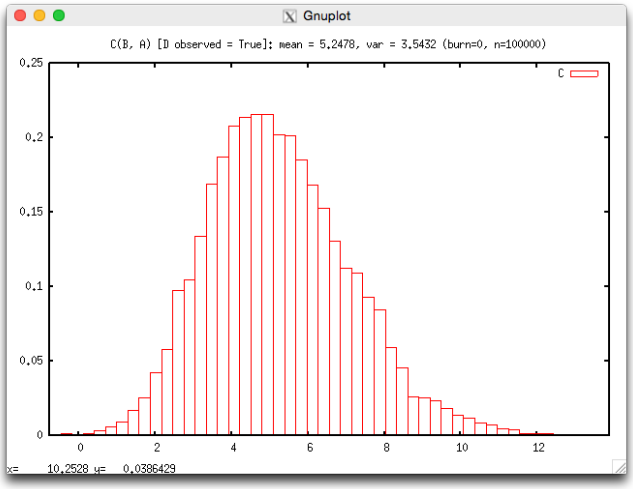


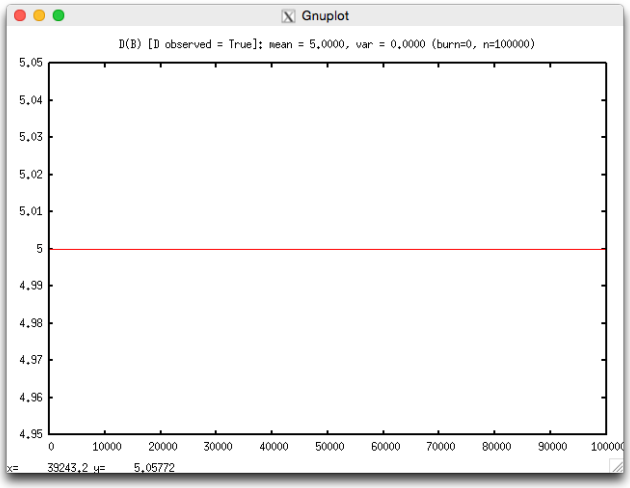












(Range is empty; can’t plot a histogram.)

Metropolis Implementation

network.py

The Network class stores the nodes and initiates the sampling process. When it is finished, it returns the results in a SampleProcessor object, which can be used to compute statistics and generate plots.

```
import logging
import evilplot

log = logging.getLogger("network")


class Network(object):

    def __init__(self, nodes=None):
        self.nodes = [] if nodes is None else nodes

    def __str__(self):
        pass

    def metropolis_sample_generator(self):
        """Create samples from the given nodes using the Metrpolis algorithm."""

        while True:
            for test_node in self.nodes:
                test_node.sample_with_metropolis()

                network_state = []
                for node in self.nodes:
                    network_state.append(node.current_value)
                yield network_state

    def collect_samples(self, burn, n, generator=None):
        """Run burn iterations, then collect n samples"""

        mcmc = generator
        if mcmc is None:
            mcmc = self.metropolis_sample_generator()

        progress_step = (burn + n) / 10
        cur_sample = 0

        log.info("Burning...")
        for i in range(burn):
            next(mcmc)
            cur_sample += 1
            if cur_sample % progress_step == 0:
                log.warning("{:.0%}... ".format(cur_sample/(burn+n)))


        log.info("Sampling...")
        samples = []
        for i in range(n):
            sample = next(mcmc)
            log.debug("Sample: " + str(sample))
            samples.append(next(mcmc))
            cur_sample += 1
            if cur_sample % progress_step == 0:
                log.warning("{:.0%}... ".format(cur_sample/(burn+n)))

        return SamplesProcessor(self.nodes, samples)


class SamplesProcessor(object):

    def __init__(self, nodes, samples):
        if not type(nodes) is list:
            raise AssertionError("'nodes' argument is not a list (type = " + type(nodes).__name__ + ")")
        self.nodes = nodes
        self.samples = samples

    def __str__(self):
        samples_str = ", ".join([node.name for node in self.nodes]) + "\n"
        samples_str += "\n".join([", ".join(map(str, sample)) for sample in self.samples])
        return samples_str

    def of_node(self, node):
        """Returns samples for the given node"""

        samples = [sample[self.nodes.index(node)] for sample in self.samples]
        return samples

    def plot_node(self, node, title=None):
        if title is None:
            title = u"Samples of {0:s}".format(node.display_name)
        p = evilplot.Plot(title=title)

        points = evilplot.Points(list(enumerate(self.of_node(node))))
        points.style = 'lines'
        points.linewidth = 1
```

```
        p.append(points)
        p.show()

    def plot_histogram_for_node(self, node, title=None, prior_pdf=None):
        if title is None:
            title = u"Histogram of samples of {0:s}".format(node.display_name)
        p = evilplot.Plot(title=title)

        if not prior_pdf is None:
            priord = evilplot.Function(prior_pdf)
            priord.title = "Prior Dist"
            p.append(priord)

        hist = evilplot.Histogram(self.of_node(node), 50, normalize=True)
        hist.title = node.display_name
        p.append(hist)
        p.show()
```

node.py

All of the nodes inherit the Node class, which provides common functionality for Metropolis sampling. The heart of the class is sample_with_metropolis(), which implements the core of the the Metropolis algorithm. Subclasses implement log_current_conditional_probability(), which returns a probability for the given node type conditional upon the values of its parents.

```
import random
import logging
import math

_log = logging.getLogger("nodes")


class Node:
```

IMPOSSIBLE = math.log(0.000000000001)

```
    def __repr__(self):
        return self.__str__()

    def __init__(self, value=None, name=None, cand_var=1, observed=False):
        self.name = name
        self.current_value = value
        self.cand_std_dev = math.sqrt(cand_var)     # std_dev of Gaussian distribution used to generate candidates
        self.is_observed = observed

        self._children = []  # subclass init methods should add self to parents' children
        self._log_p_current_value = None  # log of the last sample

    def __str__(self):
        return self.display_name()

    @property
    def pdf_name(self):
        return self.display_name()

    @property
    def node_type(self):
        return self.__class__.__name__

    @property
    def display_name(self):
        return self.name if not self.name is None else self.node_type

    @staticmethod
    def parent_node_str(node):
        return "{:.4f}".format(node) if not isinstance(node, Node) else node.display_name

    @staticmethod
    def parent_node_value(node):
        """
        If node is a list of parent nodes, returns the sum of their values.
        """
        if isinstance(node, Node):
            return node.current_value
        elif isinstance(node, list):
            return sum([Node.parent_node_value(a_node) for a_node in node])
        else:
            return node

    def connect_to_parent_node(self, parent):
        """
        If parent is a list nodes, connects to each of them.
        """
        if isinstance(parent, Node):
            parent._children.append(self)
        elif isinstance(parent, list):
            for parentnode in parent:
                self.connect_to_parent_node(parentnode)
```

```
def current_conditional_probability(self):
    """Provided for testing; use log_current_conditional_probability instead."""
    return math.exp(self.log_current_conditional_probability())

def log_current_conditional_probability(self):
    """Compute the conditional probability of this node given its parents"""
    raise NotImplementedError

def current_unnormalized_mb_probability(self):
    """Provided for testing; use log_current_unnormalized_mb_probability instead."""
    return math.exp(self.log_current_unnormalized_mb_probability())

def log_current_unnormalized_mb_probability(self):
    p = 0.0
    for node in self._children + [self]:
        p += node.log_current_conditional_probability()
    return p

def probability_of_current_value_given_other_nodes(self):
    return math.exp(self.log_probability_of_current_value_given_other_nodes())

def log_probability_of_current_value_given_other_nodes(self):
    """
    Needed only for Gibbs sampling. Metropolis sampling only requires
    a probability that is proportional to the actual probability, which
    saves us from having to determine the integral for the marginal
    probability.
    """
    raise NotImplementedError

def is_candidate_in_domain(self, cand):
    """Overridden by subclasses to reject samples that are outside the domain of the probability function."""
    return True

def select_candidate(self):
    """Can be overridden by subclasses in order to provide custom distributions. Default is Gaussian."""
    return random.gauss(self.current_value, self.cand_std_dev)

def sample_with_gibbs(self):
    """
    Samples boolean values.
    """
    if not self.is_observed:
        p = self.probability_of_current_value_given_other_nodes()

        r = random.random()
        self.current_value = (r < p)
        _log.debug("P(" + self.name + ") = " + str(p))

def sample_with_metropolis(self):
    """Sample this node using Metropolis."""

    _log.debug("Sampling {}...".format(self))

    if not self.is_observed:

        # Metropolis:
        # 1 - Use the candidate distribution to select a candidate.
        # 2 - Compare the (proportionate) probability of the candidate with the
        # (proportionate) probability of the current value.
        # 3 - If the probability of the candidate is greater, use it.
        # Otherwise, determine whether to use it as a random selection with
        # probability proportionate to the probability of the current value.

        # 1 - Select a candidate. (Since we're not using Metropolis-Hastings,
        # we use a Gaussian normal with variance provided by parameter 'cand_var'.)

        cand = self.select_candidate()
        _log.debug("last: {}, cand: {}".format(self.current_value, cand))

        # If the candidate falls outside the domain of the probability function,
        # we can skip it immediately.
        if self.is_candidate_in_domain(cand):
            # 2 - Compare the probability of the candidate with that of the current value

            # log_p_cand = candidate probability
            saved_value = self.current_value
            self.current_value = cand
            log_p_cand = self.log_current_unnormalized_mb_probability()
            self.current_value = saved_value

            # log_p_current_value = current probability
            if self._log_p_current_value is None:
                self._log_p_current_value = self.log_current_unnormalized_mb_probability()


            log_r = log_p_cand - self._log_p_current_value
            log_u = math.log(random.random())

            _log.debug("log_r = {}, log_u = {}".format(log_r, log_u))

            # 3 - Use candidate with probability proportionate to the ratio of
            # its likelihood over the likelihood of the current value.

            if log_u < log_r:
                self.current_value = cand
                self._log_p_current_value = log_p_cand
```

node_normal.py

```
from node import Node
import logging
import scipy.stats as stats
import math

_log = logging.getLogger("node_normal")


class NormalNode(Node):
    def __init__(self, value=0, name=None, mean=0, var=1, cand_var=1, observed=False):
        super().__init__(value=value, name=name, cand_var=cand_var, observed=observed)
        self.mean = mean
        self.var = var

        self.connect_to_parent_node(mean)
        self.connect_to_parent_node(var)

    def __str__(self):
        return "{} = {}".format(self.pdf_name, self.current_value)

    @property
    def pdf_name(self):
        return "{}({}, {})".format(self.display_name, Node.parent_node_str(self.mean), Node.parent_node_str(self.var))

    def is_candidate_in_domain(self, cand):
        return Node.parent_node_value(self.var) > 0

    def log_current_conditional_probability(self):
        """
        Return probability given current values of 'mean' and 'var'.
        (If 'mean' and 'var' are parent nodes, get their current_value.)
        """

        mean = Node.parent_node_value(self.mean)
        var = Node.parent_node_value(self.var)

        if var == 0:
            _log.debug("Node " + str(self) + ": Cannot compute a normal probability when variance is 0.")
            return Node.IMPOSSIBLE

        p = stats.norm.pdf(self.current_value, mean, math.sqrt(var))
        _log.debug("  p = {}".format(p))
        log_p = (Node.IMPOSSIBLE if p == 0 else math.log(p))

        _log.debug("p({}={}) = {}".format(self.display_name, self.current_value, p))
        return log_p
```

node_invgamma.py

```
from node import Node
import logging
import scipy.stats as stats
import math

_log = logging.getLogger("node_invgamma")


class InvGammaNode(Node):
    def __init__(self, value=1, name=None, shape=1, scale=1, cand_var=1, observed=False):
        super().__init__(value=value, name=name, cand_var=cand_var, observed=observed)
        self.shape = shape
        self.scale = scale

        if shape is None:
            raise ValueError("Parameter 'shape' is required")

        if value <= 0:
            raise ValueError("Parameter 'value' must be greater than 0.")

        self.connect_to_parent_node(shape)
        self.connect_to_parent_node(scale)

    def __str__(self):
        return "{} = {}".format(self.pdf_name, self.current_value)

    @property
    def pdf_name(self):
        return "{}({}, {})".format(self.display_name, Node.parent_node_str(self.shape),
Node.parent_node_str(self.scale))

    def is_candidate_in_domain(self, cand):
        return cand > 0

    def log_current_conditional_probability(self):

        assert(self.current_value > 0)

        shape = Node.parent_node_value(self.shape)
        scale = Node.parent_node_value(self.scale)

        p = stats.invgamma.pdf(self.current_value, a=shape, scale=scale)
        log_p = (Node.IMPOSSIBLE if p == 0 else math.log(p))

        _log.debug("p({}={}) = {}".format(self.display_name, self.current_value, p))
```

```
        return log_p
```

node_gamma.py

```
from node_invgamma import *

_log = logging.getLogger("node_gamma")


class GammaNode(InvGammaNode):
    def __init__(self, value=1, name=None, shape=1, scale=1, shape_modifier=None, cand_var=1, observed=False):
        super().__init__(value=value, name=name, shape=shape, scale=scale,
                         cand_var=cand_var, observed=observed)

        self.shape_modifier = shape_modifier

    def log_current_conditional_probability(self):

        assert(self.current_value > 0)

        shape = Node.parent_node_value(self.shape)
        scale = Node.parent_node_value(self.scale)

        if not self.shape_modifier is None:
            shape = self.shape_modifier(shape)

        p = stats.gamma.pdf(self.current_value, a=shape, scale=1/scale)
        log_p = (Node.IMPOSSIBLE if p == 0 else math.log(p))

        _log.debug("p({}={}) = {}".format(self.display_name, self.current_value, p))
        return log_p
```

node_poisson.py

```
from node import Node
import logging
import scipy.stats as stats
import math
import random

_log = logging.getLogger("node_poisson")


class PoissonNode(Node):
    def __init__(self, value=1, name=None, rate=1, cand_var=1, observed=False):
        super().__init__(value=value, name=name, cand_var=cand_var, observed=observed)
        self.rate = rate

        if value <= 0:
            raise ValueError("Parameter 'value' must be greater than 0.")

        self.connect_to_parent_node(rate)

    def __str__(self):
        return "{} = {}".format(self.pdf_name, self.current_value)

    @property
    def pdf_name(self):
        return "{}({})".format(self.display_name, Node.parent_node_str(self.rate))

    def is_candidate_in_domain(self, cand):
        return cand > 0

    def select_candidate(self):
        """For Poisson, use Metropolis with a candidate distribution that rounds samples from a normal."""
        return round(random.gauss(self.current_value, self.cand_std_dev), 0)


    def log_current_conditional_probability(self):

        assert(self.current_value > 0)

        rate = Node.parent_node_value(self.rate)

        p = stats.poisson.pmf(self.current_value, mu=rate)
        log_p = (Node.IMPOSSIBLE if p == 0 else math.log(p))

        _log.debug("p({}={}) = {}".format(self.display_name, self.current_value, p))
        return log_p
```

node_beta.py

```
from node import Node
import logging
import scipy.stats as stats
import math

_log = logging.getLogger("node_beta")


class BetaNode(Node):
    def __init__(self, value=1, name=None, alpha=1, beta=1, cand_var=1, observed=False):
        super().__init__(value=value, name=name, cand_var=cand_var, observed=observed)
```

```
        self.alpha = alpha
        self.beta = beta

        if value < 0 or value > 1:
            raise ValueError("Parameter 'value' must be greater than 0 and less than 1.")

        self.connect_to_parent_node(alpha)
        self.connect_to_parent_node(beta)

    def __str__(self):
        return "{} = {}".format(self.pdf_name, self.current_value)

    @property
    def pdf_name(self):
        return "{}({}, {})".format(self.display_name, Node.parent_node_str(self.alpha),
Node.parent_node_str(self.beta))

    def is_candidate_in_domain(self, cand):
        return 0 <= cand <= 1

    def log_current_conditional_probability(self):

        assert(self.current_value > 0)

        alpha = Node.parent_node_value(self.alpha)
        beta = Node.parent_node_value(self.beta)

        p = stats.beta.pdf(self.current_value, a=alpha, b=beta)
        log_p = (Node.IMPOSSIBLE if p == 0 else math.log(p))

        _log.debug("p({}={}) = {}".format(self.display_name, self.current_value, p))
        return log_p
```

node_bernoulli.py

```
from node import Node
import logging
import math
import random
import scipy.stats as stats

_log = logging.getLogger("node_bernoulli")


class BernoulliNode(Node):
  def __init__(self, value=1, name=None, p=0.5, observed=False):
    super().__init__(value=value, name=name, cand_var=1, observed=observed)
    if not isinstance(p, list):
      p = [p]

    self.p = p

    if value < 0 or value > 1:
      raise ValueError("Parameter 'value' must be between 0 and 1.")

    for parent in self.p:
      self.connect_to_parent_node(parent)

  def __str__(self):
    return "{} = {}".format(self.pdf_name, self.current_value)

  @property
  def pdf_name(self):
    return "{}({})".format(self.display_name, Node.parent_node_str(self.p))

  def select_candidate(self):
    # p = Node.parent_node_value(self.p)
```

```
    sample = 1 if random.random() <= 0.5 else 0
    return sample

def log_current_conditional_probability(self):
    """
    For Bernoulli/Binomial, sample directly instead of trying to use Metropolis.
    """
    param_p = Node.parent_node_value(self.p)

    p = stats.bernoulli.pmf(self.current_value, param_p)
    log_p = (Node.IMPOSSIBLE if p == 0 else math.log(p))

    _log.debug("p({}={}) = {}".format(self.display_name, self.current_value, p))
    return log_p
```