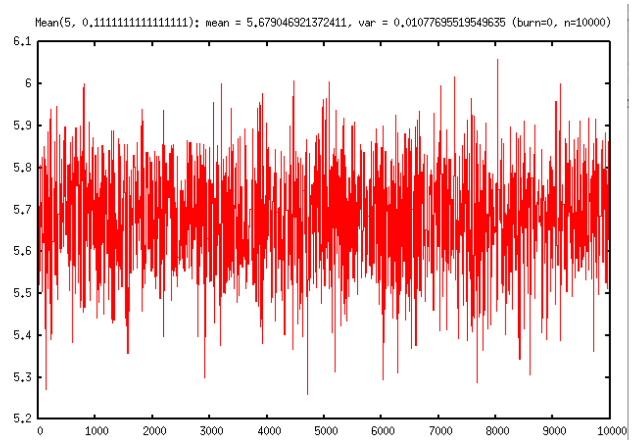
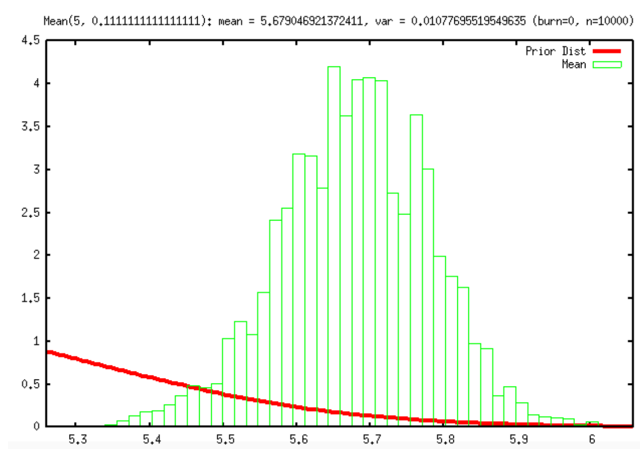
Nathan Hadfield
CS 677 (Spring 2014)

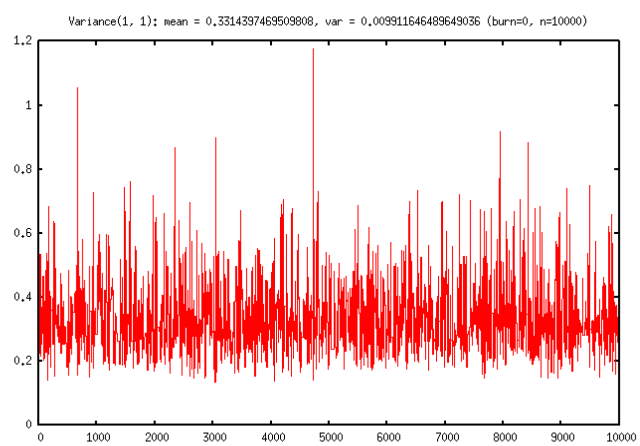
MCMC Lab #2

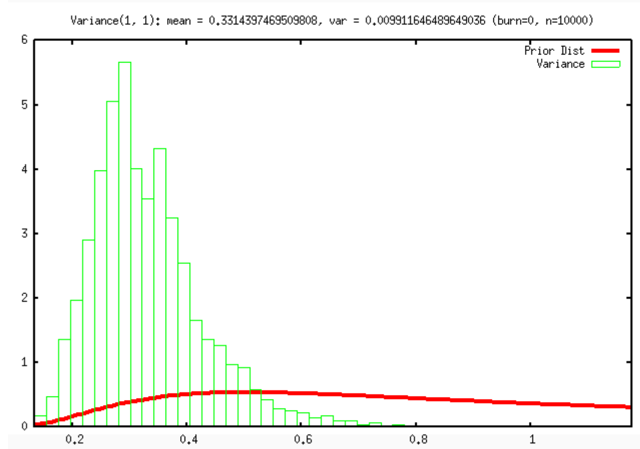
Code used set up each section of the lab is included below the charts. Code for implementing the nodes and the network is provided at the end.

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

I haven't yet been able to obtain the same results provided in the lab instructions. Perhaps I just need to let it run longer.

Here are results:

0 burn, 1000 samples:
1: MichaelBradley 68.506359; 90% interval: (68.168512, 71.818313)
2: MikeWeir 68.568669; 90% interval: (66.959685, 71.818313)
3: SteveFriesen 68.870070; 90% interval: (67.704821, 71.818313)
4: FrankBensel 68.932266; 90% interval: (68.868960, 71.065689)
5: UlyGrisette 68.988229; 90% interval: (67.596112, 71.818313)
6: DaisukeMaruyama 69.113726; 90% interval: (68.885119, 70.123087)
7: PaulAzinger 69.162171; 90% interval: (68.837082, 71.818313)
8: DavidLundstrom 69.181070; 90% interval: (66.685543, 71.818313)
9: HidemichiTanaka 69.278051; 90% interval: (69.116481, 71.818313)
10: ShingoKatayama 69.298395; 90% interval: (69.126480, 71.818313)
...
594: JamieElliott 74.107154; 90% interval: (71.818313, 74.116564)
595: RodPampling 74.201335; 90% interval: (71.818313, 75.632426)
596: AndreStolz 74.300154; 90% interval: (71.818313, 74.451293)
597: RodCurl 74.355775; 90% interval: (71.818313, 74.475204)
598: ScottDunlap 74.383805; 90% interval: (71.818313, 74.396361)
599: PatrickSheehan 74.435492; 90% interval: (71.818313, 76.893970)
600: KeithFergus 74.530361; 90% interval: (71.818313, 75.065474)
601: JasonDufner 74.566214; 90% interval: (71.818313, 74.786557)
602: JimmyGreen 74.684533; 90% interval: (71.818313, 77.636256)
603: AndyCrain 74.819675; 90% interval: (73.548603, 74.839618)
604: ChrisNallen 75.285112; 90% interval: (71.818313, 75.565003)

0 burn, 10,000 samples:
1: CharletonDechert 63.059197; 90% interval: (61.932321, 68.114029)
2: BrettQuigley 63.823470; 90% interval: (61.957505, 70.118672)
3: WilliamLinkIV 64.433450; 90% interval: (61.821228, 71.814321)
4: JeffBrehaut 64.435480; 90% interval: (61.659941, 70.291198)
5: BrendanJones 64.467350; 90% interval: (61.866006, 71.142384)
6: KevinDurkin 64.719385; 90% interval: (61.392781, 70.201949)
7: ChrisStroud 64.721479; 90% interval: (63.019291, 71.818313)
8: TommyTolles 64.784650; 90% interval: (62.335474, 71.682574)
9: PatPerez 64.878027; 90% interval: (63.038874, 69.531869)
10: PaulMcGinley 64.933665; 90% interval: (63.304477, 70.212636)
...
594: PierreFulke 78.028745; 90% interval: (71.996713, 82.943628)
595: TimClark 78.041545; 90% interval: (72.129058, 85.259594)
596: BrianHarman 78.588884; 90% interval: (71.818313, 83.232394)
597: BobAckerman 78.641958; 90% interval: (71.818313, 82.666595)
598: DennisColligan 78.665644; 90% interval: (71.818313, 82.519481)

599: RickFehr 78.864267; 90% interval: (73.065569, 82.801983)
600: ToshiIzawa 79.049189; 90% interval: (70.566723, 85.028566)
601: MichelleWie 79.086877; 90% interval: (72.267396, 82.618260)
602: JasonGore 79.319358; 90% interval: (71.818313, 87.248540)
603: TrippIsenhour 79.325457; 90% interval: (72.559681, 81.779186)
604: ThongchaiJaidee 79.962592; 90% interval: (71.818313, 86.828954)

10,000 burn, 100,000 samples
1: DavidFaught 35.567460; 90% interval: (28.085181, 64.012255)
2: JoeOgilvie 37.690226; 90% interval: (29.178827, 60.071037)
3: VanceVeazey 37.803001; 90% interval: (21.383958, 71.586654)
4: BrianDixon 40.833441; 90% interval: (26.970465, 65.075742)
5: KevinStadler 42.298471; 90% interval: (24.054469, 75.066460)
6: RobertDeruntz 42.535868; 90% interval: (31.368566, 63.595836)
7: BobTway 42.794322; 90% interval: (36.734639, 65.448897)
8: CaseyWittenberg 43.693175; 90% interval: (38.414358, 64.497341)
9: ToddBarranger 44.153859; 90% interval: (29.675191, 64.137654)
10: LorenPersonett 44.681982; 90% interval: (36.007485, 66.985873)
...
594: StephenAmes 98.209983; 90% interval: (68.929898, 114.151196)
595: EdFiori 98.758722; 90% interval: (74.667969, 105.211274)
596: BobbyKalinowski 99.448144; 90% interval: (75.693244, 115.008141)
597: MattHendrix 99.689896; 90% interval: (79.921170, 106.713186)
598: MiguelRivera 100.312942; 90% interval: (80.392490, 104.480465)
599: BoydSummerhays 101.314547; 90% interval: (80.844905, 111.848930)
600: FredCouples 101.725960; 90% interval: (65.362897, 113.869461)
601: TjaartvanderWalt 102.357989; 90% interval: (71.908438, 113.004750)
602: AndyMorse 103.448081; 90% interval: (79.315435, 118.139906)
603: HeathSlocum 108.544687; 90% interval: (80.742846, 122.449682)
604: EricAxley 111.032587; 90% interval: (84.755746, 121.812276)

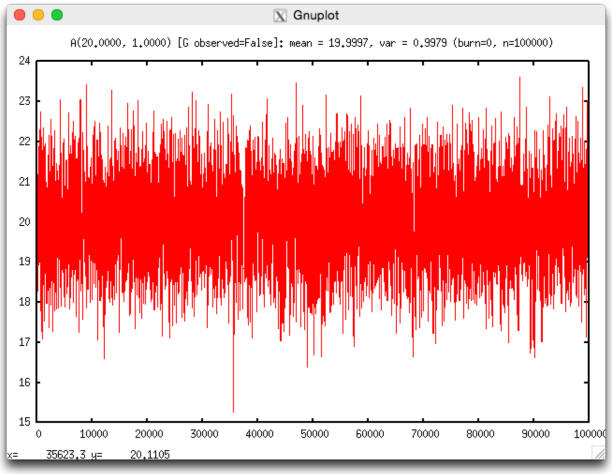
network_golfers.py

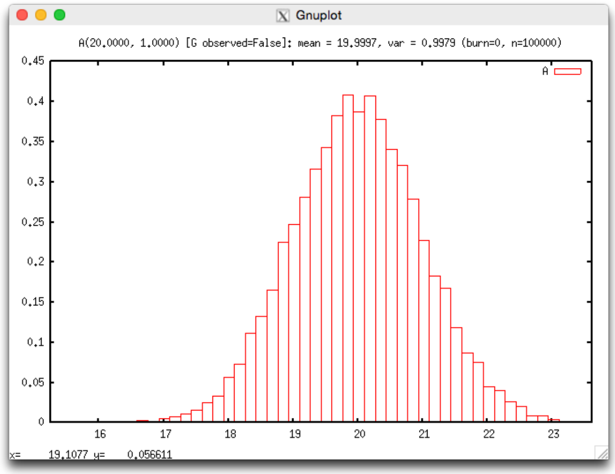
```
from node_normal import *
from node_invgamma import *
from network import *
from operator import itemgetter
import numpy

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

hypertournmean_candsd = 1  # variance
hypervar_candsd = 1  # variance
mean_candsd = 1  # variance
obsvar_candsd = 1  # variance

# tourns                    # list of tournament #s
# golfers                   # list of golfer names
# data                      # tuples of (name, score, tourn) from golfdataR.dat
# est_avg                   # estimated average score

data = []
for line in open('golfdataR.dat'):
    line_data = line.strip().split(' ')
    line_data[1] = float(line_data[1])
    data.append(line_data)

# data = [line.strip().split(' ') for line in open('golfdataR.dat')]

golfers = sorted(set([line[0] for line in data]))
scores = [float(line[1]) for line in data]
tourns = sorted(set([line[2] for line in data]), key=int)
```

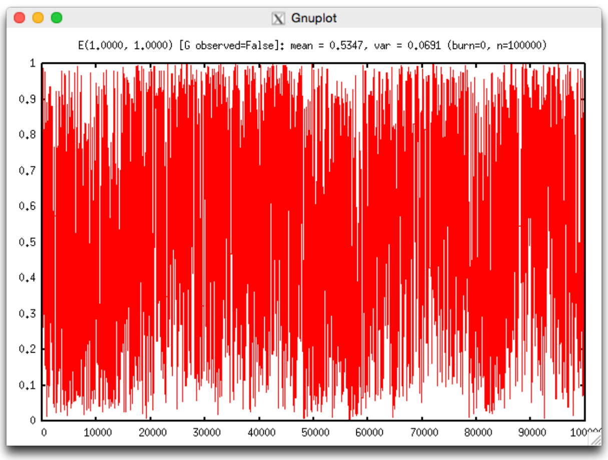
```python
est_avg = numpy.mean(scores)

#data = [item.strip() for item in (line.split(' ') for line in open('golfdataR.dat'))]


#data = (item.strip() for item in (line.split(' ')) for line in open('golfdataR.dat'))

hypertournmean = NormalNode(72.8, name='Tournament Hyper Mean', cand_var=hypertournmean_candsd, mean=72, var=2)
hypertournvar = InvGammaNode(3, name='Tournament Hyper Var', cand_var=hypervar_candsd, shape=18, scale=1 / .015)
tournmean = {}
for tourn in tourns:
    tournmean[tourn] = NormalNode(est_avg, 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(est_avg, name=golfer, cand_var=mean_candsd, mean=0, var=hypergolfervar)

obsvar = InvGammaNode(3.1, 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 = 100000

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

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
```

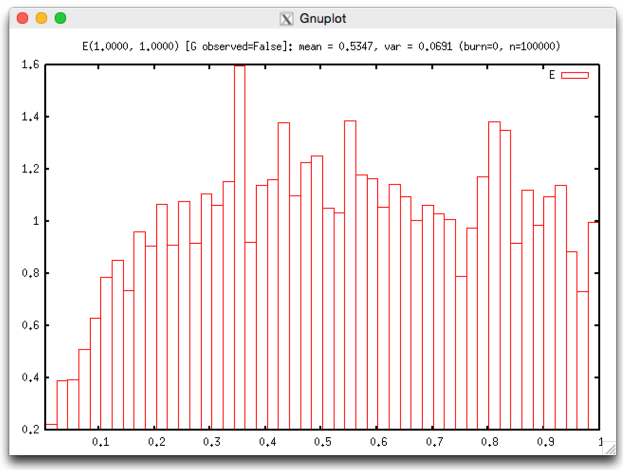
Wacky Network

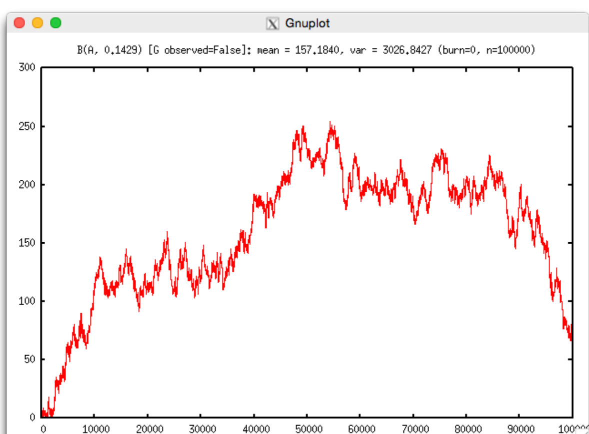
With no observations:

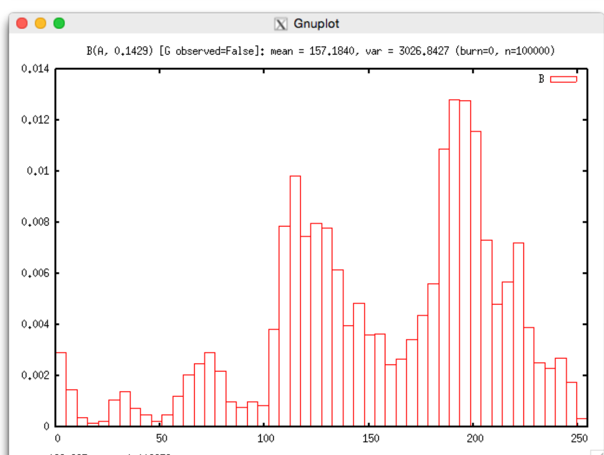


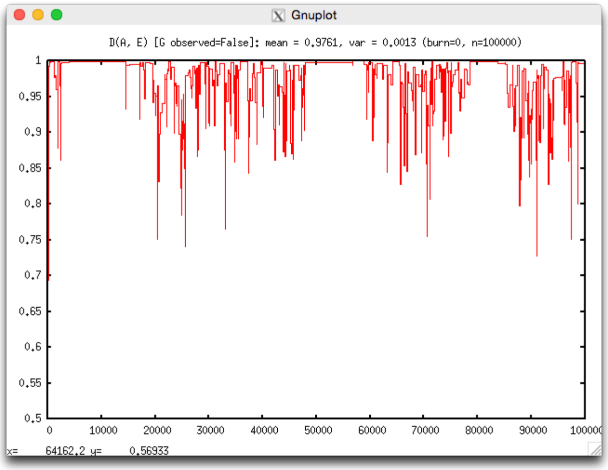


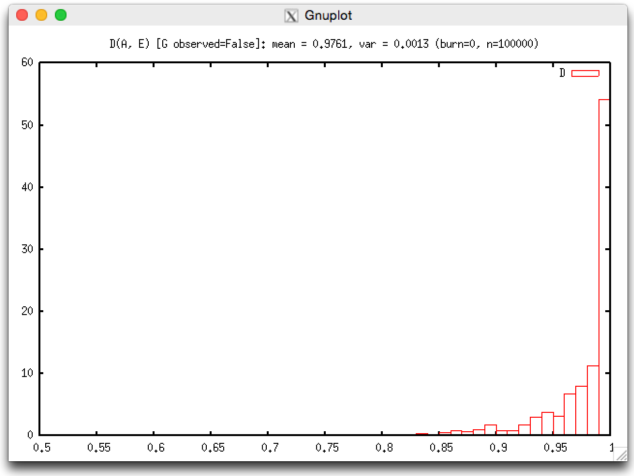


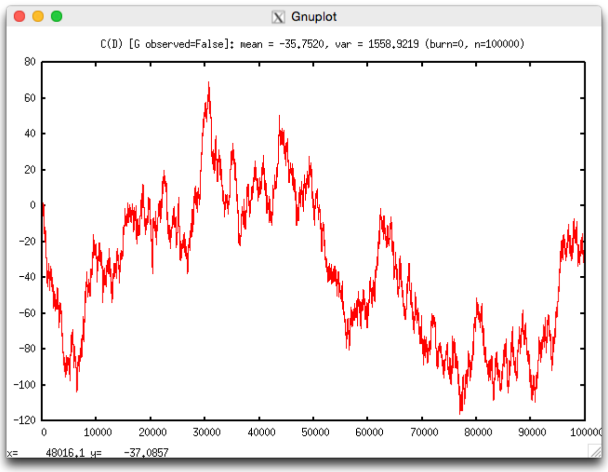


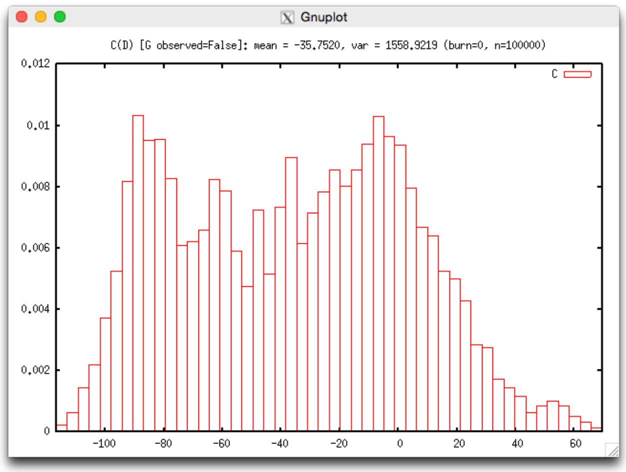


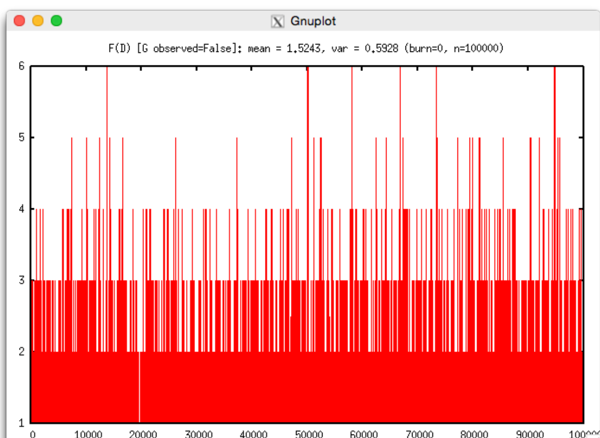


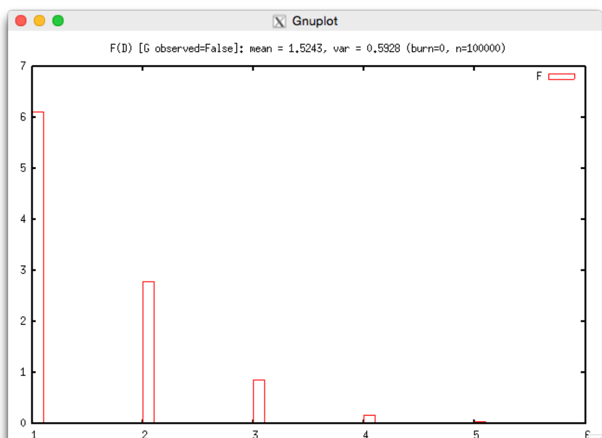

Gnuplot
D(A, E) [G observed=False]: mean = 0.9761, var = 0.0013 (burn=0, n=100000)
x= 64162.2 y= 0.56933

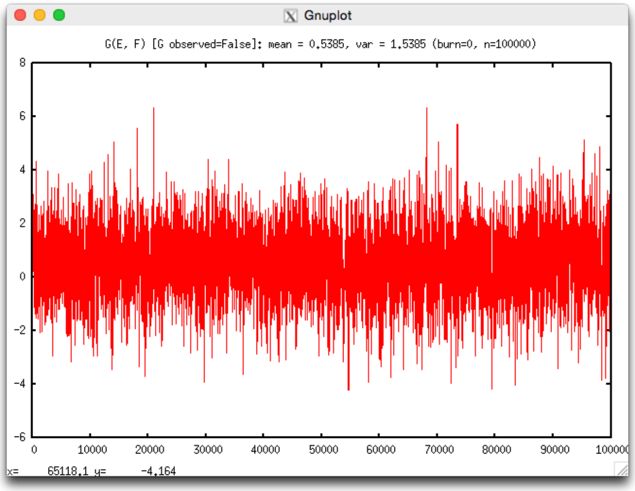

Gnuplot
D(A, E) [G observed=False]: mean = 0.9761, var = 0.0013 (burn=0, n=100000)
D

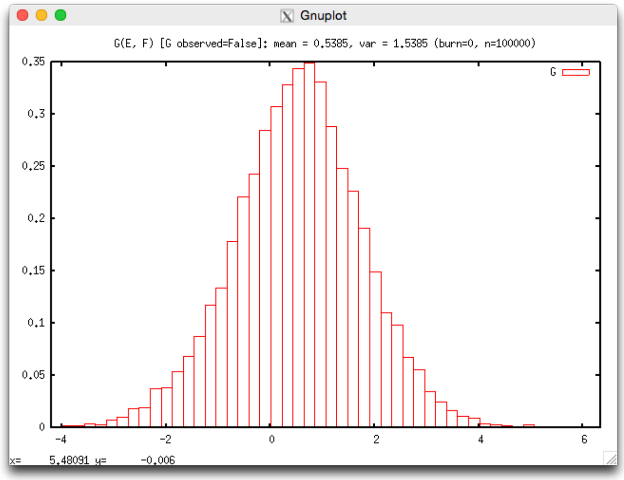

Gnuplot
C(D) [G observed=False]: mean = -35.7520, var = 1558.9219 (burn=0, n=100000)
x= 48016.1 y= -37.0857


Gnuplot
C(D) [G observed=False]: mean = -35.7520, var = 1558.9219 (burn=0, n=100000)
C

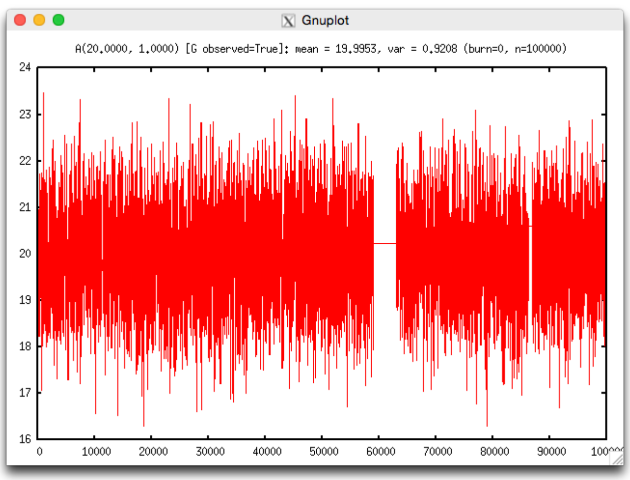

Gnuplot
F(D) [G observed=False]: mean = 1.5243, var = 0.5928 (burn=0, n=100000)

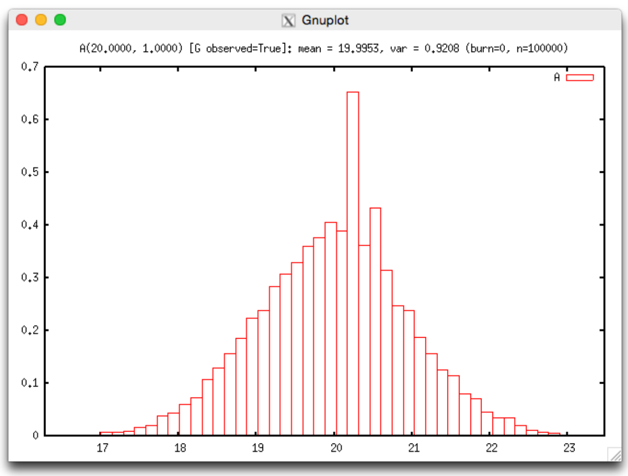

Gnuplot
F(D) [G observed=False]: mean = 1.5243, var = 0.5928 (burn=0, n=100000)
F

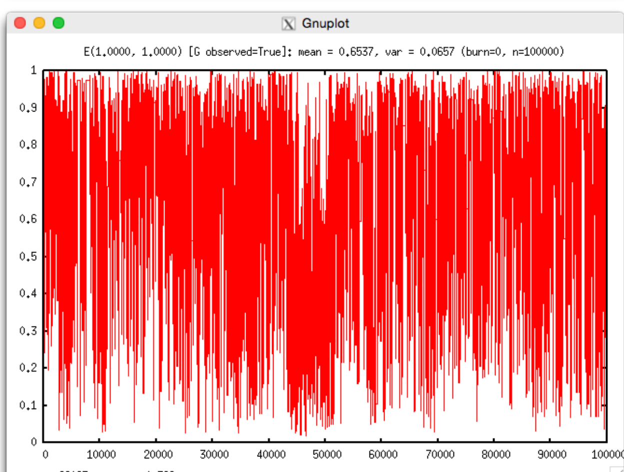


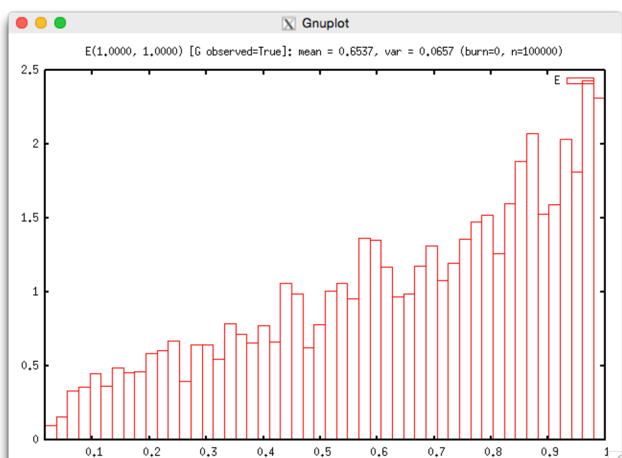


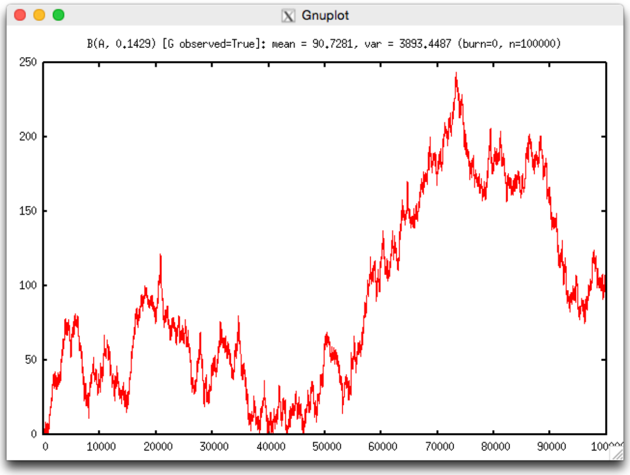
With G observed to be 5:

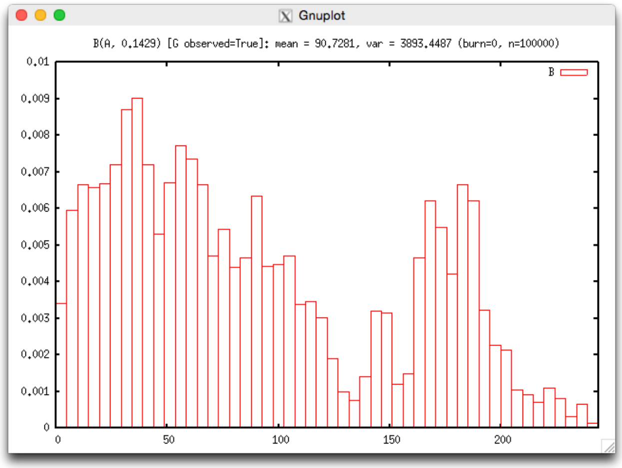


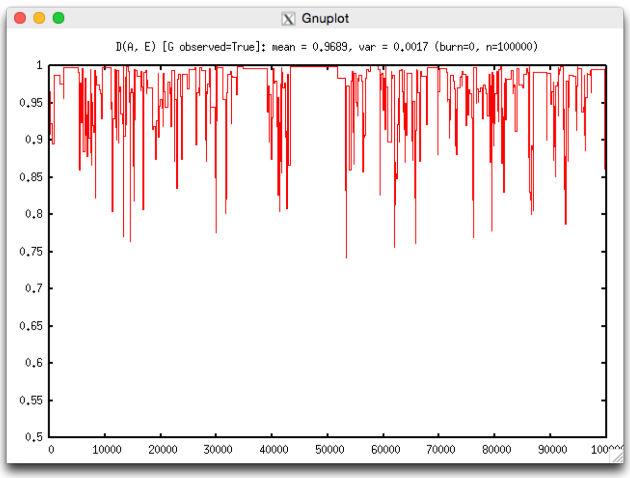


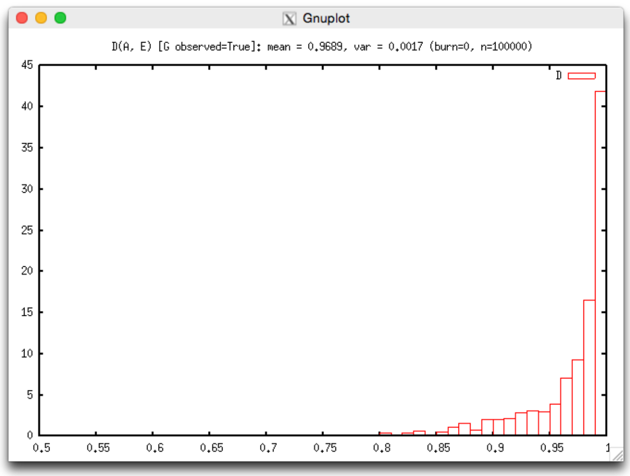


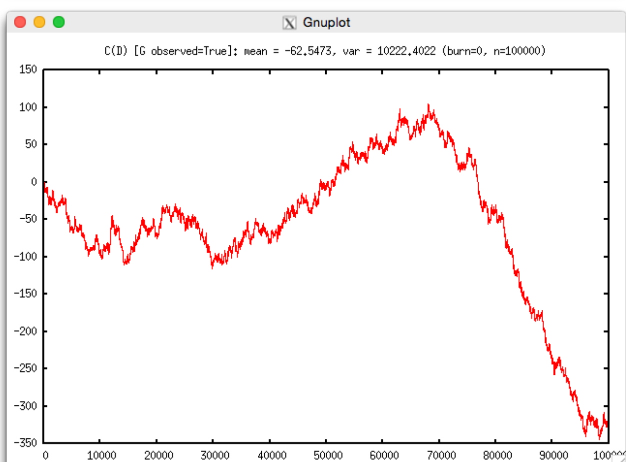


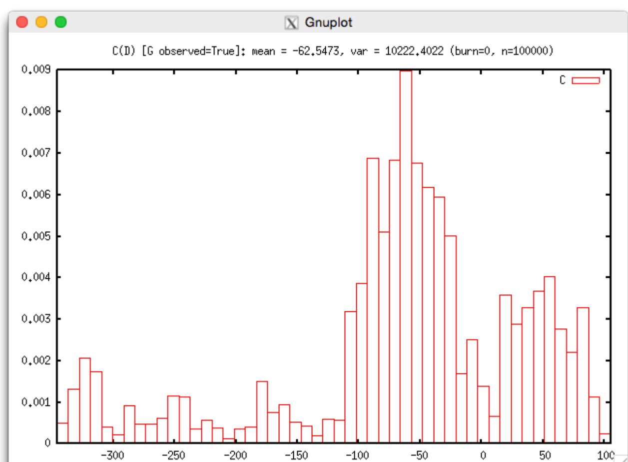

Gnuplot
B(A, 0.1429) [G observed=True]: mean = 90.7281, var = 3893.4487 (burn=0, n=100000)

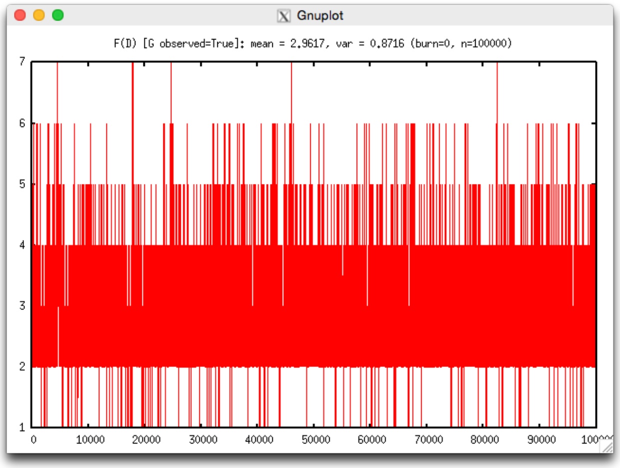

Gnuplot
B(A, 0.1429) [G observed=True]: mean = 90.7281, var = 3893.4487 (burn=0, n=100000)
B

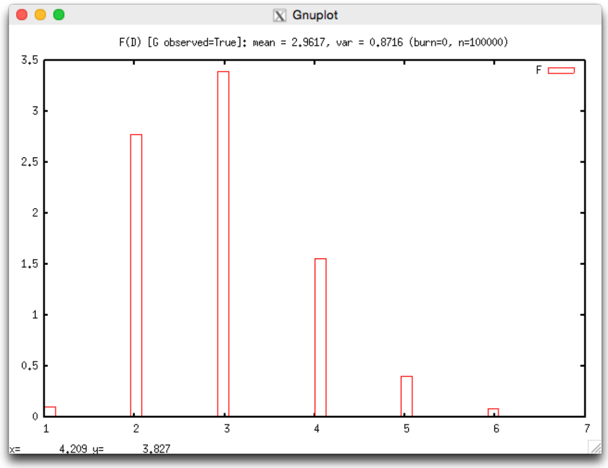

Gnuplot
D(A, E) [G observed=True]: mean = 0.9689, var = 0.0017 (burn=0, n=100000)

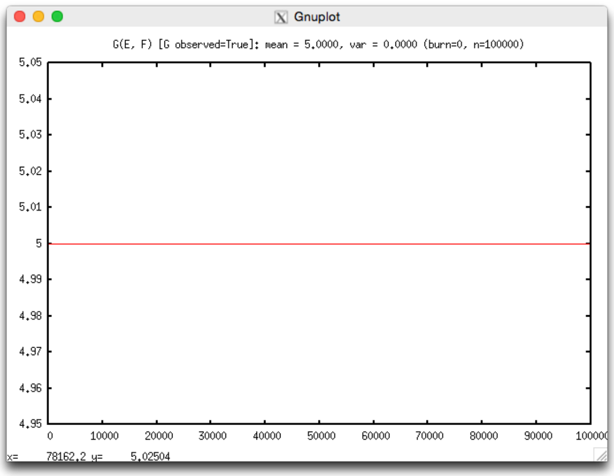

Gnuplot
D(A, E) [G observed=True]: mean = 0.9689, var = 0.0017 (burn=0, n=100000)
D


Gnuplot
C(D) [G observed=True]: mean = -62.5473, var = 10222.4022 (burn=0, n=100000)


Gnuplot
C(D) [G observed=True]: mean = -62.5473, var = 10222.4022 (burn=0, n=100000)
C







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

network_wacky.py

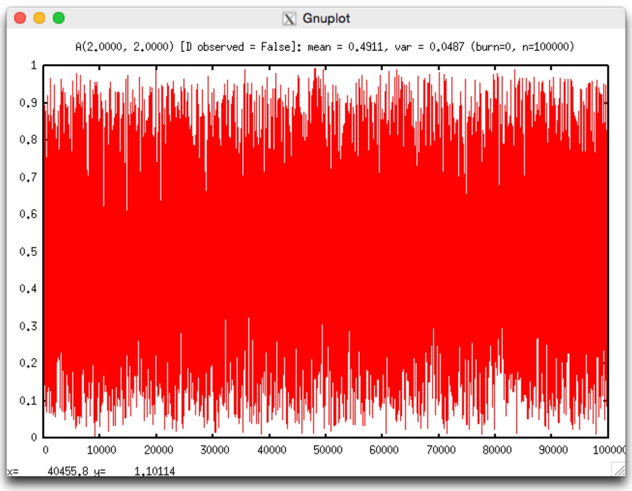
```
from node_normal import *
from node_beta import *
from node_gamma import *
from node_poisson import *
from node_bernoulli import *
from network import *
import numpy

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)
```

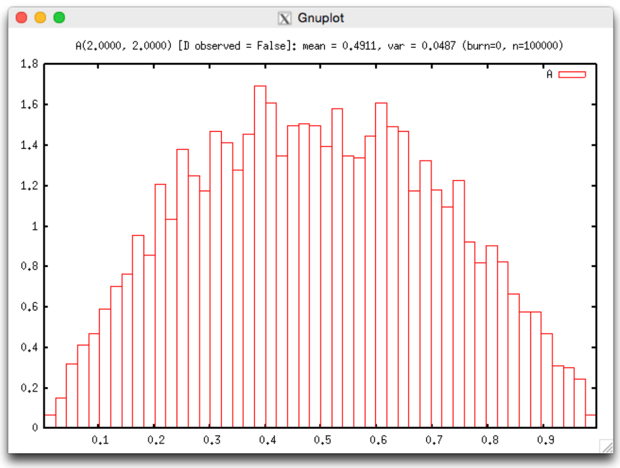
My Network

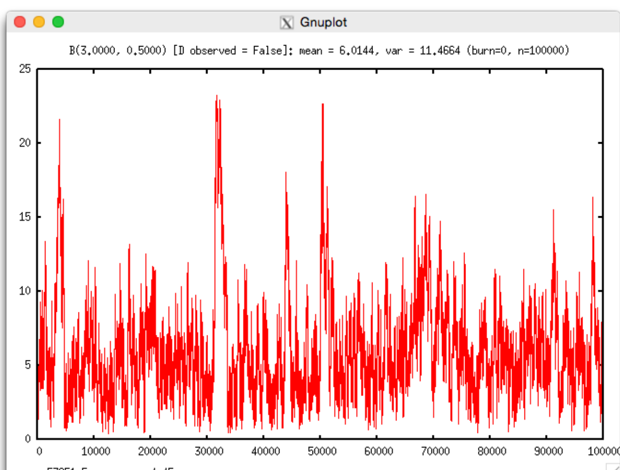
What happens to the network when D is observed at 5?

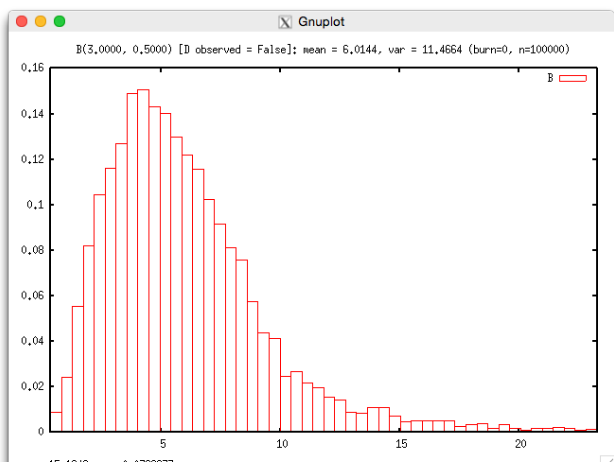
```
a = BetaNode(0.4, 'A', alpha=2, beta=2)
b = GammaNode(4, 'B', shape=3, scale=1/2)
c = NormalNode(0, 'C', mean=b, var=a)
d = PoissonNode(5, 'D', rate=b, observed=d_observed)
```

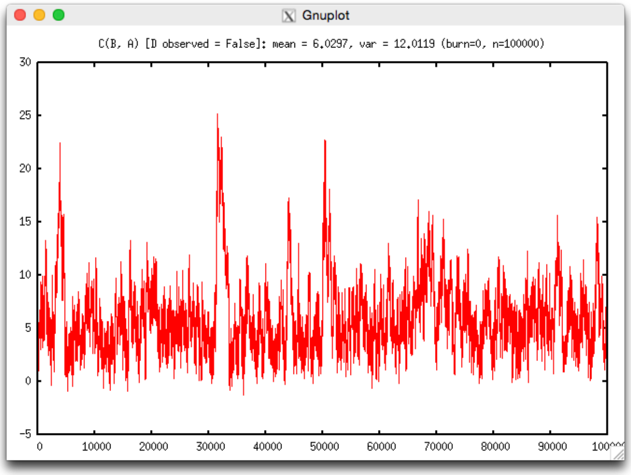
D not observed:

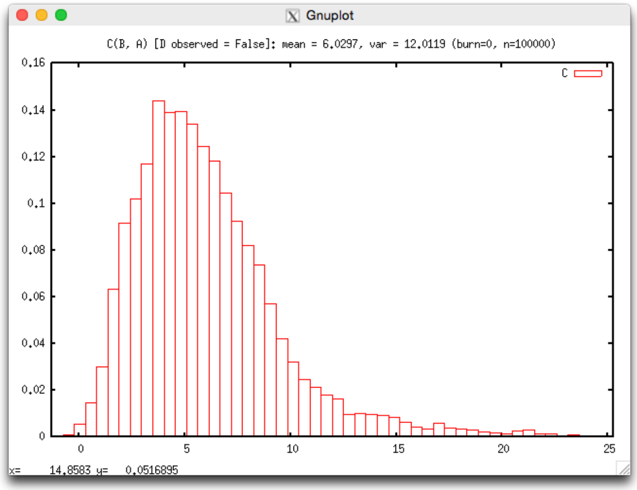


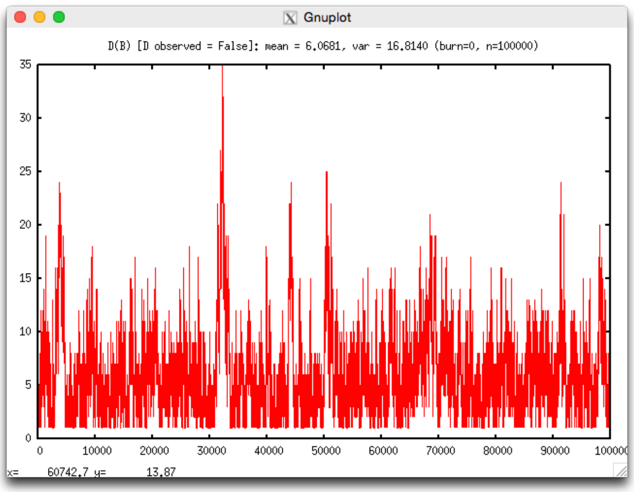


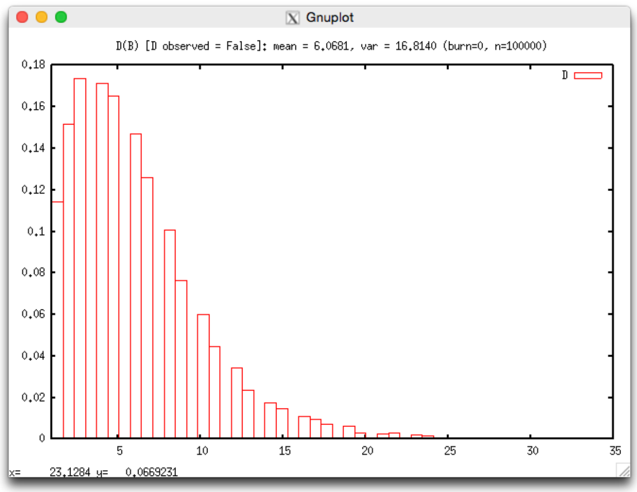




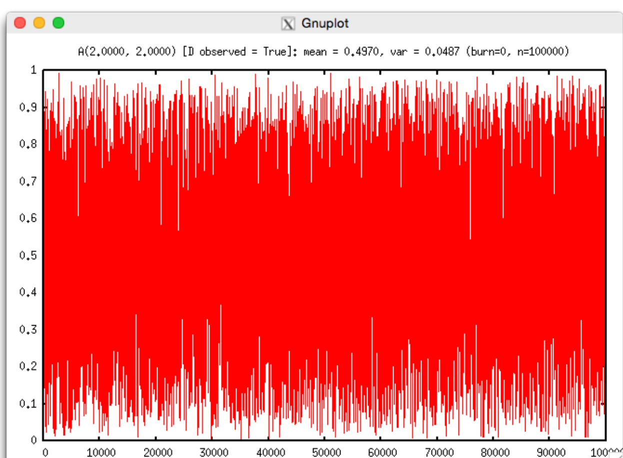


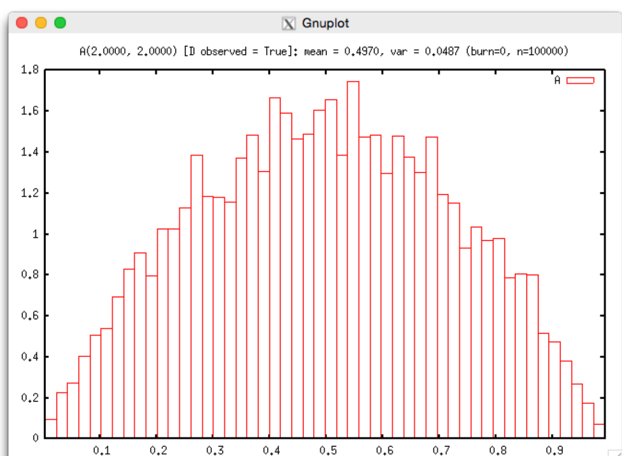


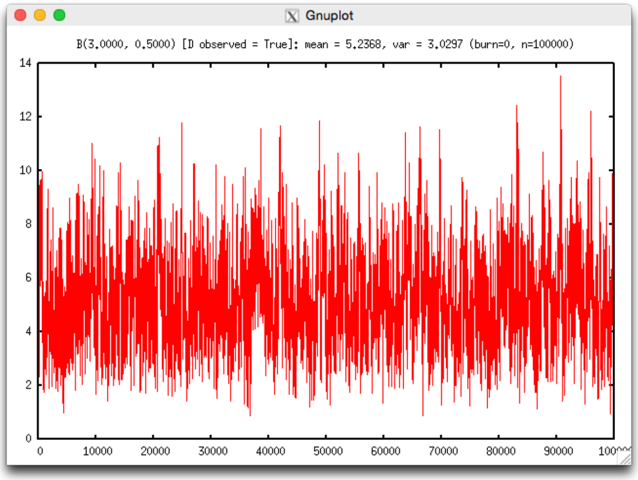


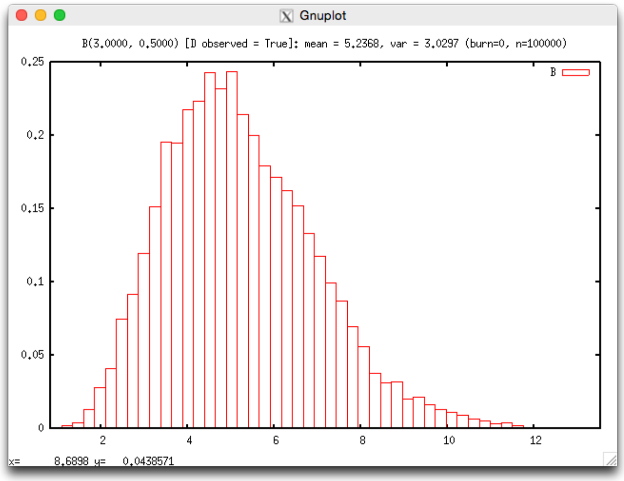


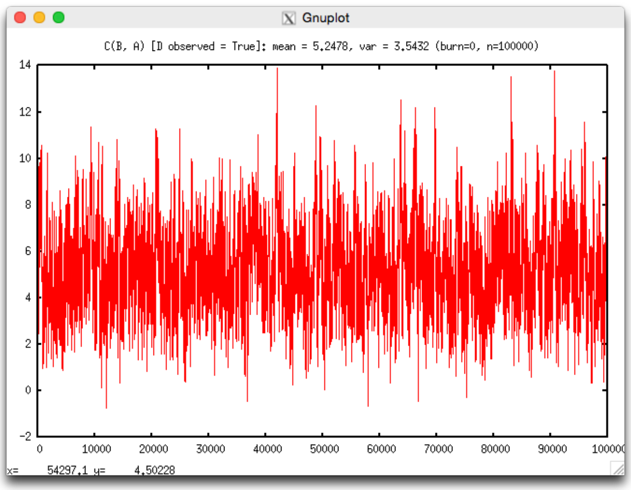
D observed at 5:

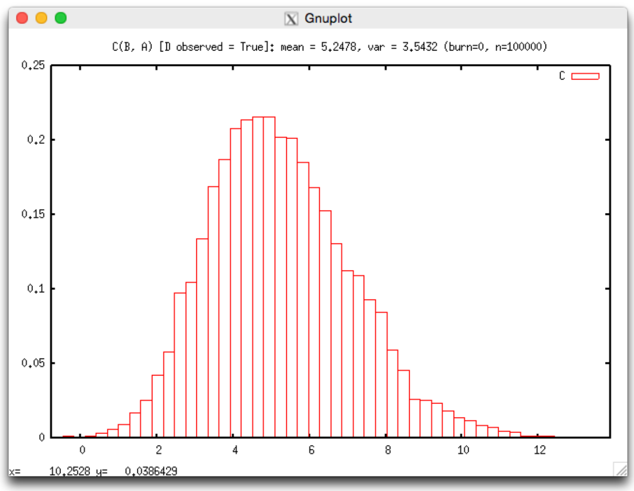


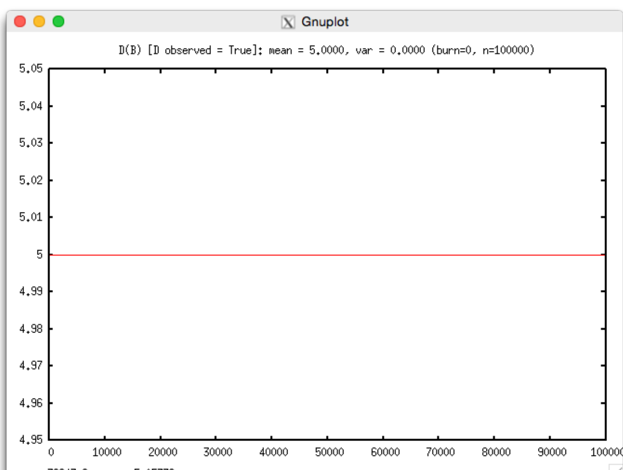












(Range is empty; can’t plot a histogram.)

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:
    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)
```

```python
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 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)

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

_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)
        self.p = p

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

        self.connect_to_parent_node(p)

    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 log_current_conditional_probability(self):
        """
        For Bernoulli/Binomial, sample directly instead of trying to use Metropolis.
        """

        p = Node.parent_node_value(self.p)
        sample = 1 if random.random() <= p else 0
        log_sample = (0 if sample == 0 else math.log(sample))

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