CS6730 : Assignment 2

Instructor and TAs

Release: 27th Mar 2019; **Due: Apr 5th, 11.59pm**

- Submit to **GradeScope a single LaTeX-generated pdf file** containing your solutions. Please type your answers in the solutions blocks in the source LaTeX file of this assignment.
- You are encouraged to collaborate/discuss with other students on this assignment, but write your solutions/code in your own words.
- We tried our best to keep coding questions simple so that each coding question should require no more than a page of code (in R say).

---

0. [MANDATORY: TIME TO RE-SEARCH] The research paper assignment will be done collaboratively in teams of 4 students each - its deadline will be later in the course, and its evaluation will be based on a critique written by each member in his/her own words (70%) and a team presentation (30%). For now, please provide your

   (a) team members and team name, and

   Solution: Team members:

   - Harshavardhan P K (CS15B061)
   - Monisha J CS15B053
   - Chandrashekhar Dhulipala (CS15B009)
   - Mohammad Sohail Shariff (CS15B025)

   (b) the single research paper your team has chosen.

   Some example papers are in the course moodle, but feel free to choose any other research paper with some component of probabilistic graphical modeling.

1. (5 points) [JUNCTION TREE GETS A TUNE-UP] Let $H = (\mathcal{X}, E)$ be a chordal Markov network, and let $T = (C, F)$ be a junction tree for H. Let β_c for $c \in C$ be a set of (locally) calibrated potentials, i.e. for all $(i, j) \in F$, we have that

$$\sum_{C_i \setminus S_{i,j}} \beta_i(C_i) = \sum_{C_j \setminus S_{i,j}} \beta_j(C_j),$$

where $S_{i,j} = C_i \cap C_j$. Show that for all $i, k \in C$ (not necessarily neighbors in T), we have that if $a \in C_i$ and $a \in C_k$,

$$\sum_{C_i \setminus \{a\}} \beta_i(C_i) = \sum_{C_k \setminus \{a\}} \beta_k(C_k)$$

Solution: We saw in class that $\beta_i(C_i) = \sum_{X \setminus C_i} \prod_{c \in C} \phi_c(C_c) = \phi_i(C_i) \prod_{j \in N(i)} \delta_{j \to i}(S_{ij})$ is the unnormalized marginal distribution for variables in C_i.

$$P(C_i) = \frac{1}{Z}\beta_i(C_i) = \frac{1}{Z} \sum_{X \setminus C_i} \prod_{c \in C} \phi_c(C_c) \tag{1}$$

where $Z = \sum_{C_i} \sum_{X \setminus C_i} \prod_{c \in C} \phi_c(C_c) = \sum_X \prod_{c \in C} \phi_c(C_c)$ is independent of C_i.
It is given that $a \in C_i \cap C_k$.
Thus, $\sum_{C_i \setminus \{a\}} \beta_i(C_i) = \sum_{C_i \setminus \{a\}} ZP(C_i) = ZP(a)$.
Similarly, $\sum_{C_k \setminus \{a\}} \beta_k(C_k) = ZP(a)$. Thus,

$$\sum_{C_i \setminus \{a\}} \beta_i(C_i) = \sum_{C_k \setminus \{a\}} \beta_k(C_k)$$

Alternate Proof: If $(i, j) \in F$ and $a \in S_{i,j}$:

$$\sum_{C_i \setminus \{a\}} \beta_i(C_i) = \sum_{S_{i,j} \setminus \{a\}} \sum_{C_i \setminus S_{i,j}} \beta_i(C_i) = \sum_{S_{i,j} \setminus \{a\}} \sum_{C_j \setminus S_{i,j}} \beta_j(C_j) = \sum_{C_j \setminus \{a\}} \beta_j(C_j) \tag{2}$$

For non-neighbors i, k, by running intersection property there exists a single path $i — v_1 — v_2 — \cdots — v_n — k$ such that for all edges in path the separation set contains a. And hence:

$$\sum_{C_i \setminus \{a\}} \beta_i(C_i) = \sum_{C_{v_1} \setminus \{a\}} \beta_i(C_{v_1}) = \cdots = \sum_{C_{v_n} \setminus \{a\}} \beta_i(C_{v_n}) = \sum_{C_k \setminus \{a\}} \beta_k(C_k) \tag{3}$$

(Note: After a junction tree algorithm is run, we can compute $P(X_i)$ by choosing **any** clique c containing X_i and marginalizing out all other variables from β_c – this works as the β_c of different cliques c are each valid (unnormalized) marginals of the same (original) joint distribution. This question shows that this behavior also holds true for any set of general factors/potentials β_c as long as they are calibrated (including *pseudo-marginal* potentials β_c that are not marginals for any valid joint distribution over $\mathcal{X}$, and which have applications in variational inference).)

2. (10 points) [TOYING AROUND WITH MCMC::MH] Use normal distribution centered at the current state of the chain as a proposal distribution in the Metropolis-Hastings algorithm to sample from the $\mathrm{Gamma}(\theta, 1)$ distribution when θ is a non-integer that is at least 2.

   (a) (3 points) Write down the acceptance probability (after all simplifications). What properties do you need to verify to confirm your method reaches the right stationary distribution after running for a sufficiently long time? Show your verification.

Solution: The proposal distribution is

$$Q(x'|x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(\frac{-(x-x')^2}{2\sigma^2}\right) = Q(x|x')$$

Also, $\mathrm{Gamma}(\theta, 1)$ distribution has

$$P(x) \propto x^{\theta-1} \exp(-x) \text{ if } x > 0$$

If $x \leqslant 0, P(x) = 0$.
Thus the acceptance probability of moving from x to x' is

$$A(x'|x) = \begin{cases} \min(1, \frac{P(x')Q(x|x')}{P(x)Q(x'|x)}) = \min(1, \frac{P(x')}{P(x)}) = \min(1, (\frac{x'}{x})^{\theta-1} e^{x-x'}) & \text{if } x' > 0 \\ 0 & \text{if } x' \leqslant 0 \end{cases}$$

Verification: Firstly, the markov chain is irreducible and ergodic since we can transition from any point x to any other point x' in range $(0, \infty)$ in one step with a finite probability since proposal distribution $Q(x'|x)$ is positive and $P(x')/P(x) = \exp(x - x')$ is positive. The markov chain is aperiodic since we can remain in same state with non-zero probability as $Q(x|x)$ is positive. We need to verify the detailed balance condition:

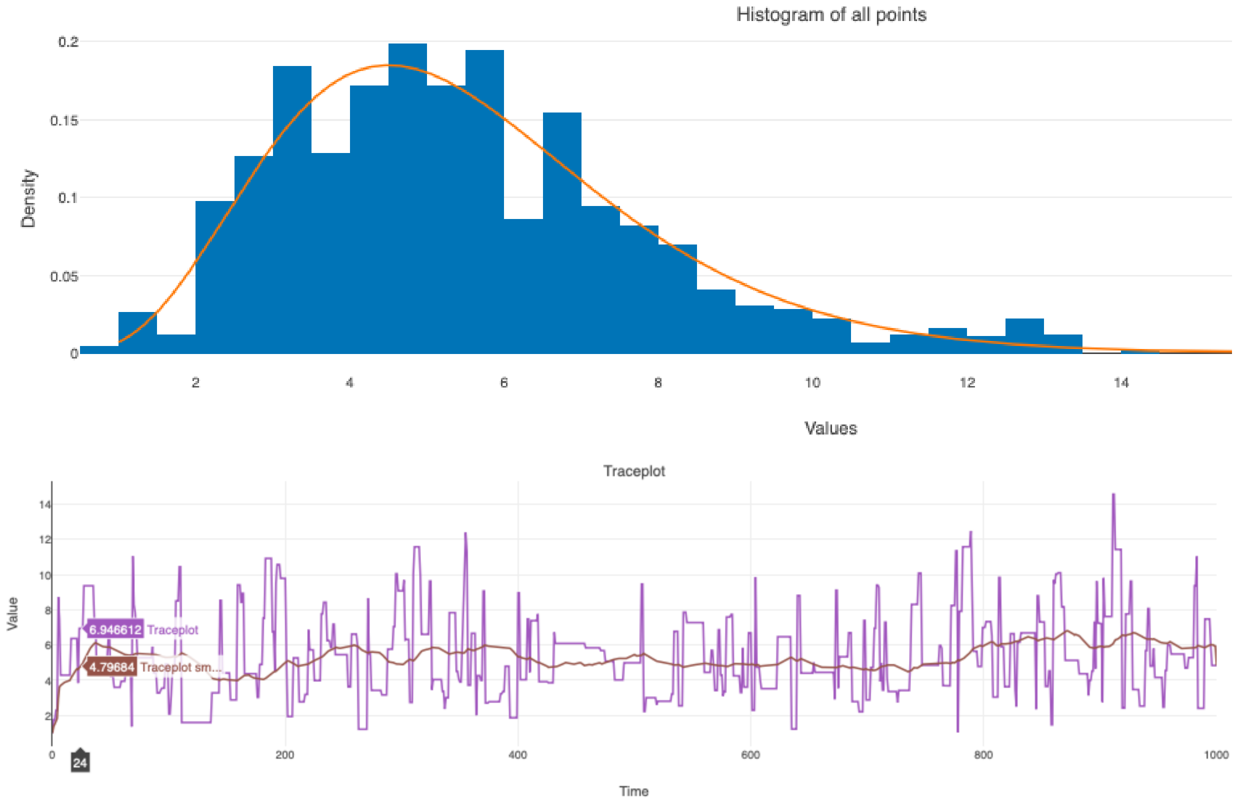
$$P(x)Q(x'|x)A(x'|x) = P(x')Q(x|x')A(x|x')$$

Without loss of generality let $A(x'|x) = 1$.
Then $P(x')Q(x|x')A(x|x') = P(x')Q(x|x')\frac{P(x)Q(x'|x)}{P(x')Q(x|x')} = P(x)Q(x'|x) = P(x)Q(x'|x)A(x'|x)$

(b) (4 points) Provide your code that simulates 1000 values from $\mathsf{Gamma}(5.5, 1)$, and show the trace plots, and the histogram of X_n (with the gamma density overlaid).

Solution: Initially we chose **standard deviation of normal distribution œ = 10** for proposal distribution. The starting point is 1.

```python
import numpy as np
from numba import jit, njit
from scipy.stats import gamma

import plotly.graph_objs as go
from plotly import tools
from plotly.offline import download_plotlyjs, init_notebook_mode, plot, iplot


@njit
def mp_gamma_sample(k=1., theta=2., T=10000, sigma=1., init = 1.):
    if init <= 0:
        init = np.random.rand() *5.
    curr_state = init

    states = []
    states.append(curr_state)
    accept_r1 = []
    accept_r = []

    #For each iteration of sampling
    for _ in range(T):
        #calculate the displacement to next proposed state X'
        gap = np.random.normal(0.0,sigma)
        next_state = curr_state + gap
        #Check if P(X')>0 else remain in X
        if next_state <= 0:
            states.append(curr_state)
            accept_r.append(0)
            accept_r1.append(0)
            continue

        #Calculate the accepatance proabbility
        accept_prob = ((next_state/curr_state)**(k-1))*np.exp((curr_state-next_state)/theta)
        #Sample from [0,1] uniformly and check if Acceptance probability is grater than that number
        accept_r1.append(int(np.random.rand() < accept_prob))
        accept_r.append(min(1,accept_prob))

        #Change next state to X' if accepted
        curr_state = next_state if accept_r1[-1] else curr_state
        states.append(curr_state)
    return states, accept_r

'''
A smoothing function for plots
'''
@njit
def get_accept(accept_r, states, lag=5000):
    accept = np.copy(accept_r)
    st = np.copy(states[:])
    for i in prange(lag):
        accept[i] = np.mean(accept_r[:i+1])
        st[i] = np.mean(states[:i+1])
    for i in prange(lag, len(states)-1):
        accept[i] = np.mean(accept_r[i-lag:i])
        st[i] = np.mean(states[i-lag:i])
    return accept, st
if __name__ == "__main__":
    T=1000
    sigma = 10
    lag = 100
    last = 1000

    #Sample T times
    states, accept_r = mp_gamma_sample(T=T, sigma=sigma, k=5.5, theta=1)

    #We sample last 1000 from states
    states_samples = states[-last:]
    print("Done")

    #Calculate acceptance rate
    accept, st = get_accept(np.array(accept_r, dtype = float), np.array(states), lag)
    #accept1 = np.cumsum(accept_r)/np.arange(1, len(accept_r)+1)
    accept1 = accept_r
    print("Done")

    #Calculate Gamma pdf function
    vals = np.arange(1,20,0.01)
    gvals = gamma.pdf(vals, 5.5)

    #Plot
    trace_gamma = go.Scatter(
        x=vals,
        y=gvals,
        mode='lines',
        name='Gamma(5.5,1)'
    )

    hist_full = go.Histogram(
        x=states,
        histnorm='probability density',
        name='histogram of '+str(T)+' runs',
        xaxis='x1',
        yaxis='y1'
    )

    hist_first = go.Histogram(
        x=states[:1000],
        histnorm='probability density',
        name='histogram of first 1000 samples',
        xaxis='x2',
        yaxis='y2'
    )

    hist_last = go.Histogram(
        x=states[-1000:],
        histnorm='probability density',
        name='histogram of last 1000 samples',
        xaxis='x2',
        yaxis='y2'
    )

    trace_plot = go.Scatter(
        x=np.arange(len(states)),
        y=states,
        mode='lines',
        name='Traceplot',
        xaxis='x3',
        yaxis='y3'
    )

    trace_plot_mean = go.Scatter(
        x=np.arange(len(st)),
        y=st,
        mode='lines',
        name='Traceplot smoothened (average of '+str(lag)+' samples)',
        xaxis='x3',
        yaxis='y3'
    )
    accept_plot = go.Scatter(
        x=np.arange(len(accept1)),
        y=accept1,
        mode='lines',
        name='Acceptance rate',
        xaxis='x4',
        yaxis='y4'
    )

    accept_plot_mean = go.Scatter(
        x=np.arange(len(accept)),
        y=accept,
        mode='lines',
        name='Acceptance rate over next '+str(lag)+' samples',
        xaxis='x4',
        yaxis='y4'
    )

    fig = tools.make_subplots(rows=4, cols=1, subplot_titles=('Histogram of all points', 'Histogram of last 1000 points', 'Traceplot', 'Acceptance Rate'))
    fig.append_trace(hist_full, 1, 1)
    fig.append_trace(trace_gamma, 1, 1)
    fig.append_trace(hist_last, 2, 1)
    fig.append_trace(trace_gamma, 2, 1)
    fig.append_trace(trace_plot, 3, 1)
    fig.append_trace(trace_plot_mean, 3, 1)
    fig.append_trace(accept_plot_mean, 4, 1)
    fig.append_trace(accept_plot, 4, 1)

    fig['layout']['xaxis1'].update(title='Values')
    fig['layout']['xaxis2'].update(title='Values')
    fig['layout']['xaxis3'].update(title='Time')
    fig['layout']['xaxis4'].update(title='Time')

    fig['layout']['yaxis1'].update(title='Density')
    fig['layout']['yaxis2'].update(title='Density')
    fig['layout']['yaxis3'].update(title='Value')
    fig['layout']['yaxis4'].update(title='Acceptance Rate')
    fig['layout'].update(title='Metropolis hasting for Gamma distribution', height=2000)

    plot(fig, filename='plots/MH_Summary.html')
```

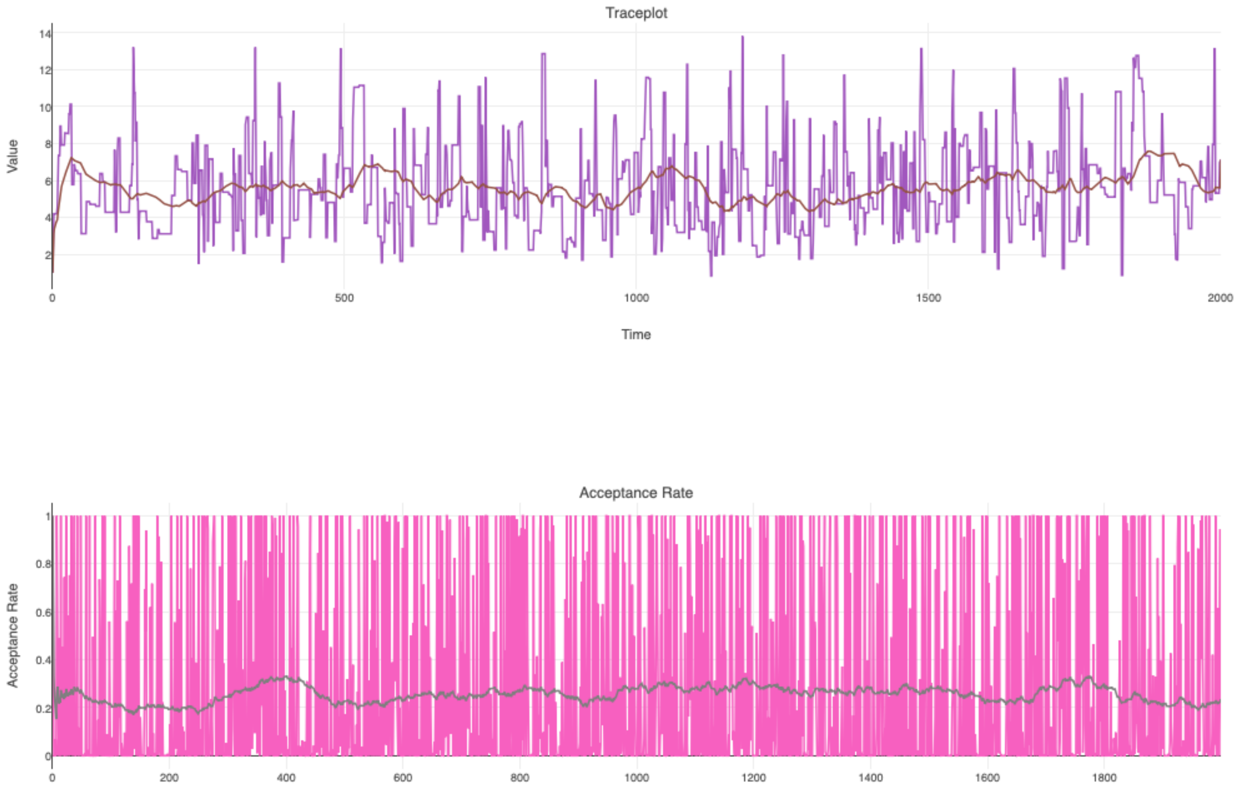


Pink: Trace Plot
Brown: Smoothed average for 100 samples

(c) (3 points) How did you choose your burn-in time? Report it along with your acceptance rate during the burn-in vs. sample collection periods. How did these values change as a function of the variance parameter of your normal distribution, and what do you think is the optimal variance for fast convergence?

Solution: We chose to sample after the trace plots begun to develop similar pattern over time and when acceptance probabilities average over 100 consecutive samples show lesser variance and stabilize.

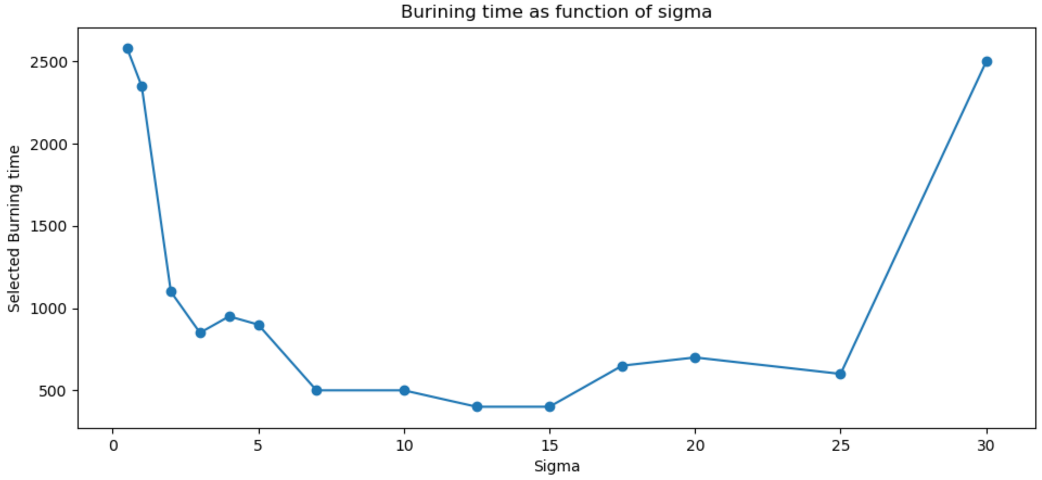
For $\sigma = 10$, we have for 2000 steps:

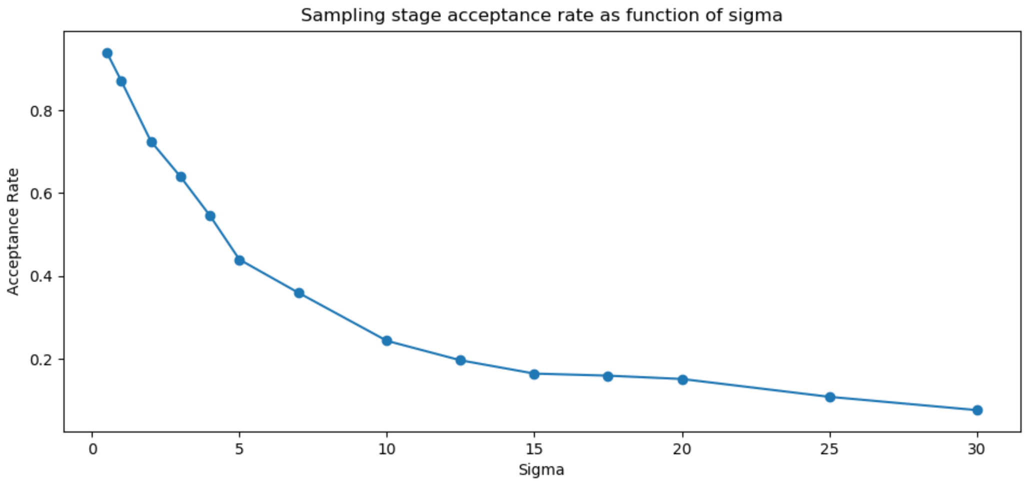


$\sigma = 10$

So the smoothed acceptance rates curve (averaged curve in grey) is already pretty mush stable at around 0.27 after $t = 500$. The trace plots stabilize after $t = 250$ similar patterns emerge. So burning period is set at $t = 500$.

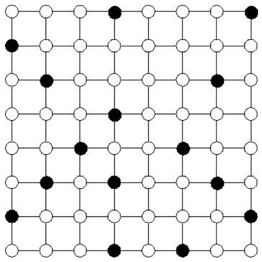
Now we plot burning rate and stabilized acceptance as function of σ. In all cases acceptance rate during burning varies highly with each iteration and starting point.





We see that optimal value of σ lies in between 10 and 15(lowest we got is at 15) where acceptance rate ranges from 0.25 to 0.18. The acceptance rate decreases with increases with sigma since as sigma increases proposal distribution tends to propose points at larger distance from current state including negative values leading to larger probability of being rejected (since the current state is most probably close to mean at 5.5).

3. (10 points) [HARD-CORE WITH MCMC::GIBBS]



Consider a (non-complete) connected graph *G = (V, E)* such as the one shown with $n = |V|$. Now each vertex in V gets either mapped to 0 or 1, where we only consider the following set $C \subset \{0,1\}^n$ of admissible configurations characterized by the property that pairs of adjacent vertices **cannot both** take the value 1 (see figure where black denotes 1).

Now, we want to pick one of the admissible configurations $\mathbf{x} \in C$ "at random". That is, we consider the (discrete) uniform distribution π on C, i.e. $\pi_{\mathbf{x}} = \frac{1}{|C|} \;\; \forall \, \mathbf{x} \in C$.

(a) (3 points) Write down the edge potentials of the undirected graphical model for this problem, and a Gibbs sampling algorithm for sampling from this model (including how you derived the associated conditional $P(X_i|X_{-i})$). Does your algorithm need to know the partition function $|C|$?

Solution: If $(X_i, X_j) \in E$, then

$$\phi_{ij}(x_i, x_j) = \begin{cases} 0 & x_i = x_j = 1 \\ 1 & \text{otherwise} \end{cases}$$

We have $P(X_i|X_{-i}) = P(X_i|N(i))$ where $N(i)$ are neighbouring nodes of X_i. Thus if atleast one of $N(i)$ currently has value 1 $P(X_i = 1|X_{-i}) = 0$. If all neighbors has value 0,

$$\frac{P(X_i = 0|N(i) = 0)}{P(X_i = 1|N(i) = 0)} = \frac{\prod_{X_k \in N(i)} \phi_{ik}(X_i = 0, X_k = 0)}{\prod_{X_k \in N(i)} \phi_{ik}(X_i = 1, X_k = 0)} = 1$$

Hence, with equal probability X_i takes values 0 or 1. We don't need $|C|$.

Algorithm 1: Hardcore-Gibbs

```
Initialize $\mathbf{x}^{(0)}$;
$t \leftarrow 0$ for $B + N$ steps do
    for $i \leftarrow [|V|]$ do
        if any neighbors of $x_i$ has value 1 then
            $x_i^{(t+1)} \leftarrow 0$;
        end
        else
            Toss a coin and if heads assign 1 to $x_i^{(t+1)}$ else 0;
        end
    end
    $t \leftarrow t + 1$;
end
Return $\{\mathbf{x}^{(B+1)}, \ldots, \mathbf{x}^{(B+N)}\}$;
```

(b) (3 points) Is the Markov chain you set up irreducible and aperiodic? Does your chain admit a distribution that satisfies detailed balance? If it simplifies your proof, assume here that your Gibbs sampling routine employs "random scan" (pick a random i from $1, \ldots, n$ and then make a move based on $P(X_i|X_{-i})$ in each epoch) instead of "systematic scan" (cycle through all i from 1 to n in each epoch).

Solution: The Markov chain is **aperiodic** since from a feasible state it can stay in the same feasible state with non-zero probability. If one of chosen x_i's neighbors have 1 its value is same. Else with probability 0.5 its value is same.

Suppose the maximum number of ones at any given state is K. We can reach a state containing all variables set to 0 in K steps (with non-zero probability) using random scan if random nodes selected are ones with values one and they are changed to 0 (with probability 0.5). We can reach any state S from this state of all zeroes within K steps (and hence in exactly K steps by staying in same states) with non-zero probability. Thus in $2K$ steps we can reach from any state to any other state. Thus the markov chain is **irreducible**.

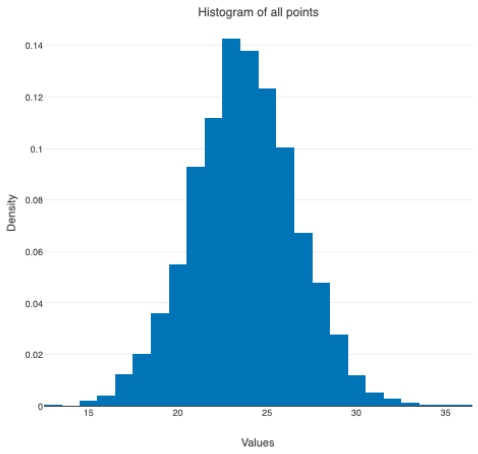
(c) (4 points) Provide your code and trace plots (of some functions that each map a configuration to a real value that helps visualize how well the chain is mixing). Plot the burn-in time you chose as a function of the size of the grid graphs you used. We didn't ask about the empirical acceptance rate here as it is always 100% for Gibbs sampling - prove that it is so under the same "random scan" epoch assumption above.

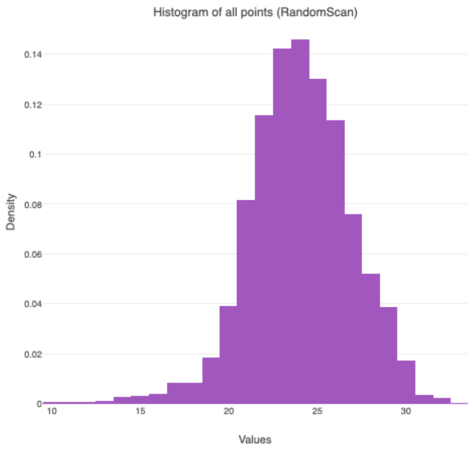
Solution: We choose the number of ones in given state as function for analyzing traceplots and choose burning time. In the code we also compare random and systematic scan.

```python
import numpy as np
from numba import jit, njit, prange
import plotly.graph_objs as go
from plotly import tools
from plotly.offline import download_plotlyjs, init_notebook_mode, plot, iplot
```

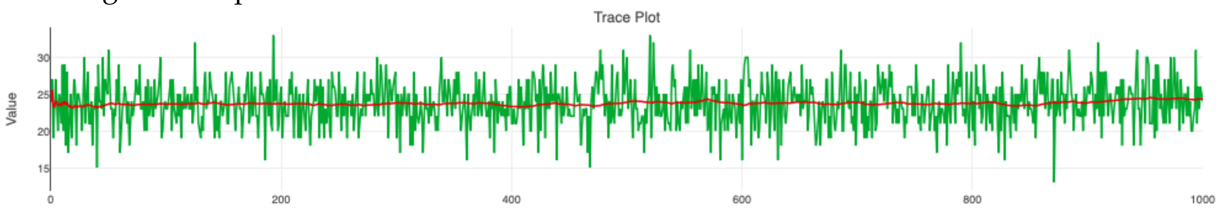
```python


@njit
def gibbs_sample(curr_state, T=100000, grid_len=10, rand = False):
    states = []
    states.append(curr_state)
    t=0
    while t<T:
        #For each i,j
        for i_ in range(grid_len):
            for j_ in range(grid_len):
                #If random scan choos i,j at random
                if rand:
                    i = np.random.randint(grid_len)
                    j = np.random.randint(grid_len)
                else:
                    i,j = i_, j_
                toss = np.random.randint(2)
                is_valid = True
                #Check if neighbors have value 1
                if toss==1:
                    if i-1>= 0 and curr_state[i-1,j]==1: is_valid=False
                    if i+1<grid_len and curr_state[i+1,j] == 1: is_valid=False
                    if j-1>= 0 and curr_state[i,j-1]==1: is_valid=False
                    if j+1<grid_len and curr_state[i,j+1] == 1: is_valid=False
                if is_valid: curr_state[i,j] = toss
                if rand:
                    states.append(np.copy(curr_state))
                    t+= 1
                    if t>=T: break
        if not rand:
            states.append(np.copy(curr_state))
            t+= 1

    return states
'''
A smoothing function for plots
'''
@njit
def get_smooth(states, lag):
    st=[]
    for i in prange(lag):
        st.append(np.mean(states[:i+1]))
    for i in prange(lag, len(states)):
        st.append(np.mean(states[i-lag:i]))
    return st

if __name__ == "__main__":
    grid_len = 10
    T=1000000

    #Get all states
    states_ = gibbs_sample(curr_state = np.eye(grid_len, dtype=np.bool), grid_len=grid_len, T=T)
    states1_ = gibbs_sample(curr_state = np.eye(grid_len, dtype=np.bool), grid_len=grid_len, T=T, rand=True)
    print("Done")

    #Number of ones in each state
    states = np.sum(states_, axis=(1,2))
    states1 = np.sum(states1_, axis=(1,2))


    lag = 100
    #We choose last 5000
    last = 5000
    st = get_smooth(states, lag)
    st1 = get_smooth(states, lag)
    print("Done")

    hist_full = go.Histogram(
        x=states,
        histnorm='probability density',
        name='histogram of '+str(T)+' runs',
        xaxis='x1',
        yaxis='y1'
    )

    hist_last = go.Histogram(
        x=states[-last:],
        histnorm='probability density',
        name='histogram of last '+str(last)+' samples',
        xaxis='x2',
        yaxis='y2'
    )

    trace_plot = go.Scatter(
        x=np.arange(len(states)),
        y=states,
        mode='lines',
        name='Traceplot',
        xaxis='x3',
        yaxis='y3'
    )

    trace_plot_mean = go.Scatter(
        x=np.arange(len(st)),
        y=st,
        mode='lines',
        name='Traceplot smoothened (average of '+str(lag)+' samples)',
        xaxis='x3',
        yaxis='y3'
    )

    hist_full1 = go.Histogram(
        x=states1,
        histnorm='probability density',
        name='histogram of '+str(T)+' runs',
        xaxis='x1',
        yaxis='y1'
    )

    hist_last1 = go.Histogram(
        x=states1[-last:],
        histnorm='probability density',
        name='histogram of last '+str(last)+' samples',
        xaxis='x2',
        yaxis='y2'
    )

    trace_plot1 = go.Scatter(
        x=np.arange(len(states1)),
        y=states1,
        mode='lines',
        name='Traceplot',
        xaxis='x3',
        yaxis='y3'
    )

    trace_plot_mean1 = go.Scatter(
        x=np.arange(len(st1)),
        y=st,
        mode='lines',
        name='Traceplot smoothened (average of '+str(lag)+' samples)',
        xaxis='x3',
        yaxis='y3'
    )

    fig = tools.make_subplots(rows=6, cols=2, subplot_titles=('Histogram of all points', 'Histogram of last '+str(last)+' points','Trace Plot',
                                                                              'Histogram of all points (RandomScan)', 'Histogram of last '+str(last)+' points(RandomScan)','Trace Plot(RandomScan)'),
                              specs=[
                                     [{'rowspan': 2}, {'rowspan':2}],
                                     [None, None],
                                     [{'colspan':2}, None],
                                     [{'rowspan': 2}, {'rowspan':2}],
                                     [None, None],
                                     [{'colspan':2}, None]],
                              print_grid=True)

    fig.append_trace(hist_full, 1, 1)
    fig.append_trace(hist_last, 1, 2)
    fig.append_trace(trace_plot, 3, 1)
    fig.append_trace(trace_plot_mean, 3, 1)
    fig.append_trace(hist_full1, 4, 1)
    fig.append_trace(hist_last1, 4, 2)
    fig.append_trace(trace_plot1, 6, 1)
    fig.append_trace(trace_plot_mean1, 6, 1)


    fig['layout']['xaxis1'].update(title='Values')
    fig['layout']['xaxis2'].update(title='Values')
    fig['layout']['xaxis3'].update(title='Time')


    fig['layout']['yaxis1'].update(title='Density')
    fig['layout']['yaxis2'].update(title='Density')
    fig['layout']['yaxis3'].update(title='Value')

    fig['layout']['xaxis4'].update(title='Values')
    fig['layout']['xaxis5'].update(title='Values')
    fig['layout']['xaxis6'].update(title='Time')


    fig['layout']['yaxis4'].update(title='Density')
    fig['layout']['yaxis5'].update(title='Density')
    fig['layout']['yaxis6'].update(title='Value')

    fig['layout'].update(title='Gibbs Sampling on Hardcore model on grid graph n='+str(grid_len), height=2000)

    plot(fig, filename='plots/Hardcore_Summary.html')

```

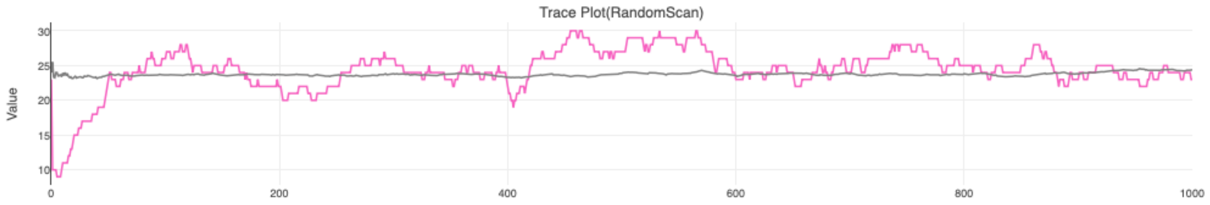
For 10 × 10 grid we give histogram for 10000 samples:





Next we give trace plots:

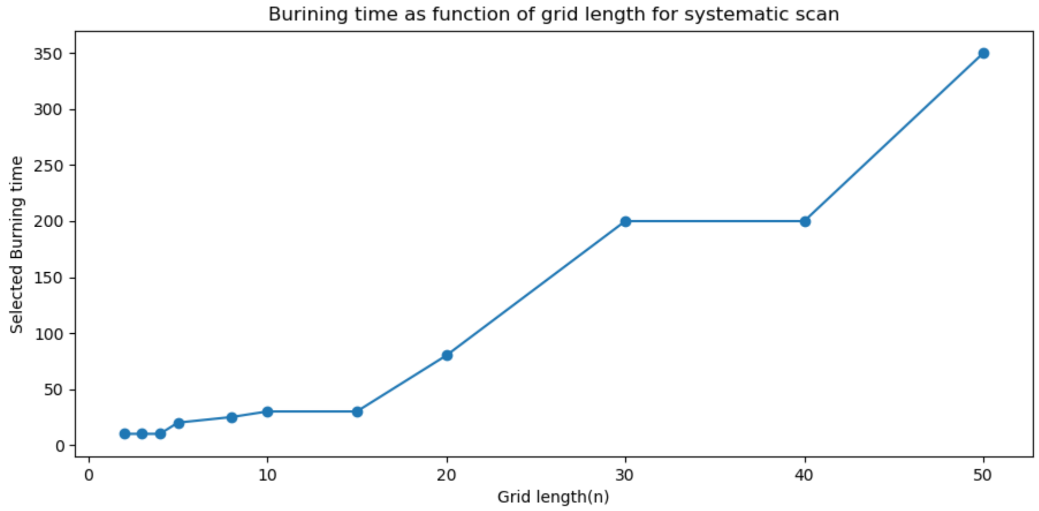


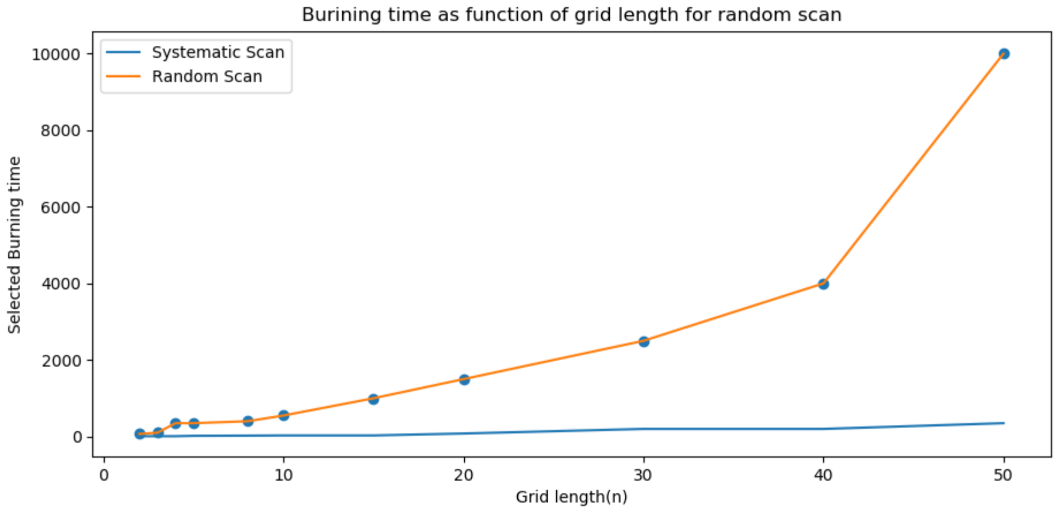


The red and grey lines are average smoothed curves considering average of 100 consecutive samples

Though histograms look similar, the trace plot for systematic scan has larger variance since it can make bigger *jumps* at each time step whereas random scan changes value of atmost one node chosen randomly at each iteration.

We choose burning time by examining when the patterns in traceplot stabilize and more importantly the histogram of last (atmost)5000 samples.





When we choose a node X_i, the proposal distribution for $x_i^{(t+1)}$ is $P(X|X_{-i} = x^{(}_{-i}t))$. Whatever the value proposed we accept it ant $x_i^{(t)}$ to that value. Thus acceptance rate is 1.

4. (10 points) [BLOCK NOW, COLLAPSE LATER] Under some conditions, it is useful to partition the random variables into "blocks", and consider these blocks as "super" random variables to do Gibbs sampling on. The same logic for Gibbs sampling applies here as well to show that this also has the stationary distribution equal to P. Sometimes implementing a block Gibbs sampling step takes strictly more computation than standard Gibbs sampling, but it helps by aiding a faster convergence to the stationary distribution.

   (a) (3 points) For Model (a) given below, what is the computational complexity (give an upper bound) of one complete cycle of Gibbs sampling, and of one complete cycle of block Gibbs sampling with $\frac{n}{k}$ blocks of k variables each? Assume "systematic scan" Gibbs sampling (see previous question) is done in each epoch.

Model (a): Let Φ be a set of factors over $X = (X_1, \dots, X_n)$. Let
$P(X) = \frac{1}{Z} \prod_{\phi \in \Phi} \phi(D_\phi)$,
where D_ϕ is the scope of factor ϕ. Let each random variable X_i take values in $\{0, 1, \dots, c-1\}$. Let each variable X_i occur in at most b factors. Let cardinality of D_ϕ be upper bounded by a for any factor $\phi \in \Phi$.

Solution: Gibbs Sampling: To compute $P(X_i = x | X_{-i} = x') \propto \tilde{P}(X_i = x, X_{-i} = x')$. We thus calculate $\tilde{P}(X_i = x, X_{-i} = x')$ for all values of X_i and get

$$P(X_i = x | X_{-i} = x') = \frac{\tilde{P}(X_i = x, X_{-i} = x')}{\sum_{x* \in Val(X_i)} \tilde{P}(X_i = x*, X_{-i} = x')}$$

To calculate $\tilde{P}(X_i = x, X_{-i} = x')$ it takes $|\Phi| - 1$ multiplications. We do c such multiplications and $c-1$ additions to get denominator. So for each variable we have time complexity of $O(c|\Phi|)$. For one round the total complexity is $O(cn|\Phi|)$.

Since we are changing only one variable x_i which is present is atmost b factors we can ignore other factors as they cancel out. Then to calculate $P(X_i = x, X_{-i} = x')$ for all c values takes $c(b-1)$ multiplications and $c-1$ additions for denominator. Thus the one round the total complexity is $O(cnb)$.

For T rounds we have running time of $O(Tcnb)$.

Block Gibbs Sampling: Let the set of variables in block j be represented as Y_j and $Y_{-j} = X \setminus Y_j$. Then,

$$P(Y_j = y_j | Y_{-j}) = \frac{\tilde{P}(Y_j = y_j, Y_{-j})}{\sum_{y* \in Val(Y_j)} \tilde{P}(Y_j = y_j, Y_{-j})}$$

For computing $\tilde{P}(Y_j = y_j, Y_{-j})$ it takes $|\Phi| - 1$ multiplications. For all c^k combinations of $Val(Y_j)$ it takes $c^k(|\Phi| - 1)$ multiplication. To calculate denominator we have $c^k - 1$ additions.

To further optimize we can ignore factors which have none of the k variables. Thus there are utmost $\min(bk, |\Phi|)$ factors to consider.

We do this for $\frac{n}{k}$ times in each round. Thus for T rounds we do $O(T\frac{n}{k}c^k \min(|\Phi|, bk))$ computations.

(b) (7 points) Try both Gibbs and block Gibbs sampling approaches for Model (b) given below (group the strongly coupled random variables X_1, X_2 into a block and X_3, X_4 into another block). Provide code, and show how long your code took to reach stationary distribution with or without blocking? Provide intuition on why sampling from Model (b) with or without blocking resulted in different convergence rates.

Model (b): Consider four random variables with distribution
$P(X_1, X_2, X_3, X_4) = \frac{1}{Z}\psi(X_1, X_2)\psi(X_3, X_4)\phi(X_2, X_3)$,
where $\psi = \begin{bmatrix} 100 & 1 \\ 1 & 100 \end{bmatrix}$ and $\phi = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$.

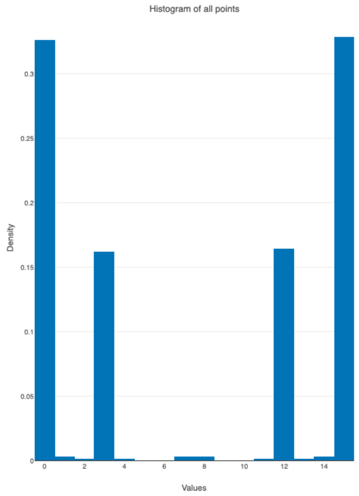
(Note: Block Gibbs sampling is different from collapsed Gibbs sampling, where certain variables are integrated out. We may see collapsed Gibbs sampling in a later assignment/tutorial.)

Solution: For the sake of brevity, we visualize the states as converting the binary number $(x_4x_3x_2x_1)$ as decimal representation.

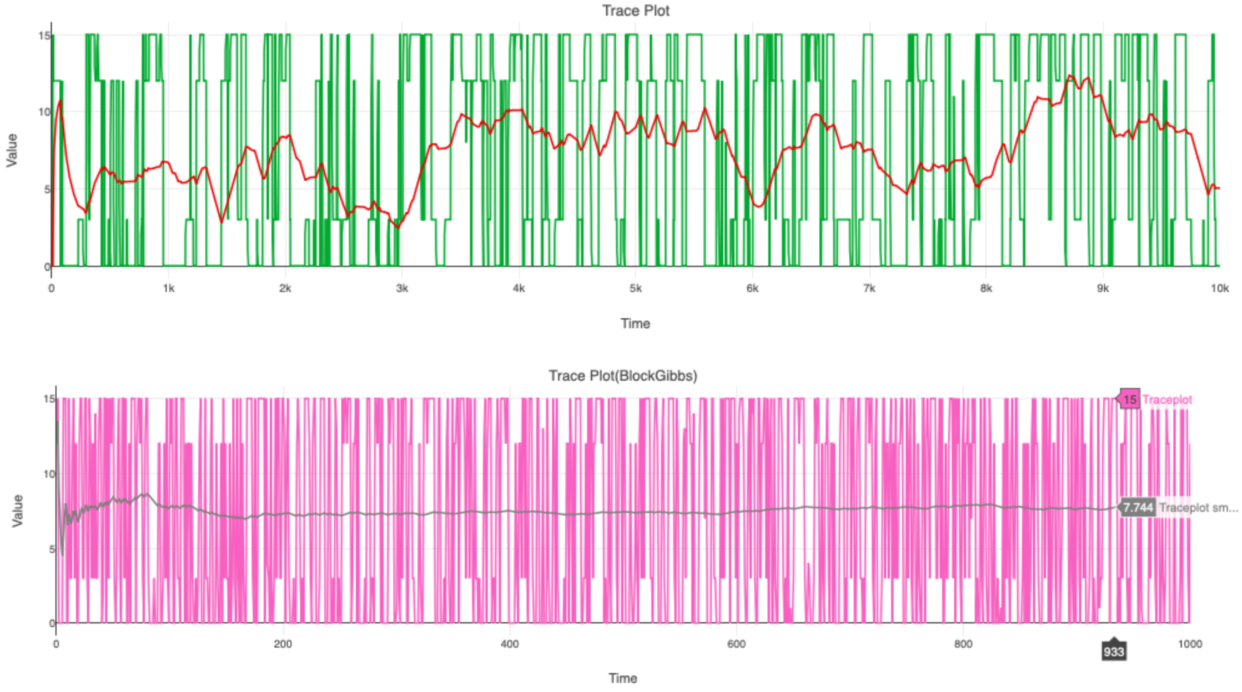
```python
import numpy as np
from numba import jit, njit, prange
import time
import plotly.graph_objs as go
from plotly import tools
from plotly.offline import download_plotlyjs, init_notebook_mode, plot, iplot

@njit
def get_unnormalized(x=[0,0,0,0]):
    ans = 1.0
    ans *= 100.0 if x[0] == x[1] else 1.0
    ans *= 100.0 if x[2] == x[3] else 1.0
    ans *= 2.0 if x[1] == x[2] else 1.0
    return ans

@njit
def gibbs_sample(T=100000):
    current_state = np.array([0,0,0,0])
    states = []
    states.append(current_state)

    for _ in range(T):
        for i in range(4):
            current_state[i] = 0
            p0 = get_unnormalized(current_state)
            current_state[i] = 1
            p1 = get_unnormalized(current_state)
            current_state[i] = 0 if np.random.rand() < (p0/(p0+p1)) else 1
        states.append(current_state.copy())

    return states

@njit
def block_gibbs(T=100000):
    current_state = np.array([0,0,0,0])
    states = []
    states.append(current_state)
    p = np.array([0.,0.,0.,0.])

    for _ in range(T):
        for i in range(2):
            for j in range(2):
                for k in range(2):
                    current_state[i*2], current_state[i*2+1] = j,k
                    p[2*j+k] = get_unnormalized(current_state)
            p = p/np.sum(p)
            p = np.cumsum(p)
            r = np.random.rand()
            if r < p[0]: current_state[i*2], current_state[i*2+1] = 0, 0
            elif r < p[1]: current_state[i*2], current_state[i*2+1] = 0, 1
            elif r < p[2]: current_state[i*2], current_state[i*2+1] = 1, 0
            else: current_state[i*2], current_state[i*2+1] = 1, 1

        states.append(current_state.copy())

    return states
```

```
'''
Concert binary to decimal
'''
def dec(x):
    ans = 0
    for i in range(4):
        ans+= x[i]*(2**i)
    return ans

def convert_states(s):
    return [dec(x) for x in s]

'''
A smoothing function for plots
'''
@njit
def get_smooth(states, lag):
    st=[]
    for i in prange(lag):
        st.append(np.mean(states[:i+1]))
    for i in prange(lag, len(states)):
        st.append(np.mean(states[i-lag:i]))
    return st


if __name__ == "__main__":
    T1=150000
    T2 = 40000
    start = time.time()
    states1 = gibbs_sample(T1)
    stop = time.time()
    print('GS Took', stop-start,'seconds')
    start = time.time()
    states2 = block_gibbs(T2)
    stop = time.time()
    print('BGS Took', stop-start,'seconds')
    s1 = convert_states(states1)
    s2 = convert_states(states2)
    print("Converted")

    lag=500
    last1 = 40000
    last2 = 10000

    st1 = get_smooth(np.array(s1, dtype=float), lag)
    st2 = get_smooth(np.array(s2, dtype=float), lag)

    print("Done")


    hist_full1 = go.Histogram(
        x=s1,
        histnorm='probability density',
        name='histogram of '+str(T1)+' runs',
        xaxis='x1',
        yaxis='y1'
    )

    hist_last1 = go.Histogram(
        x=s1[-last1:],
        histnorm='probability density',
        name='histogram of last '+str(last1)+' samples',
        xaxis='x2',
        yaxis='y2'
    )

    trace_plot1 = go.Scatter(
        x=np.arange(len(s1)),
        y=s1,
        mode='lines',
        name='Traceplot',
        xaxis='x3',
        yaxis='y3'
    )

    trace_plot_mean1 = go.Scatter(
        x=np.arange(len(st1)),
        y=st1,
        mode='lines',
        name='Traceplot smoothened (average of '+str(lag)+' samples)',
        xaxis='x3',
        yaxis='y3'
    )

    hist_full2 = go.Histogram(
        x=s2,
        histnorm='probability density',
        name='histogram of '+str(T2)+' runs',
        xaxis='x1',
        yaxis='y1'
    )

    hist_last2 = go.Histogram(
        x=s2[-last2:],
        histnorm='probability density',
        name='histogram of last '+str(last2)+' samples',
        xaxis='x2',
        yaxis='y2'
    )

    trace_plot2 = go.Scatter(
        x=np.arange(len(s2)),
        y=s2,
        mode='lines',
        name='Traceplot',
        xaxis='x3',
        yaxis='y3'
    )

    trace_plot_mean2 = go.Scatter(
        x=np.arange(len(st2)),
        y=st2,
        mode='lines',
        name='Traceplot smoothened (average of '+str(lag)+' samples)',
        xaxis='x3',
        yaxis='y3'
    )

    fig = tools.make_subplots(rows=6, cols=2, subplot_titles=('Histogram of all points', 'Histogram of last '+str(last1)+' points','Trace Plot',
                                                                   'Histogram of all points (BlockGibbs)', 'Histogram of last '+str(last2)+' points(BlockGibbs)','Trace Plot(BlockGibbs)'),
                            specs=[
                                    [{'rowspan': 2}, {'rowspan':2}],
                                    [None, None],
                                    [{'colspan':2}, None],
                                    [{'rowspan': 2}, {'rowspan':2}],
                                    [None, None],
                                    [{'colspan':2}, None]],
                            print_grid=True)

    #fig = tools.make_subplots(rows=2, cols=1, subplot_titles=('Histogram of all points', 'Histogram of last 1000 points'))
    fig.append_trace(hist_full1, 1, 1)
    fig.append_trace(hist_last1, 1, 2)
    fig.append_trace(trace_plot1, 3, 1)
    fig.append_trace(trace_plot_mean1, 3, 1)
    fig.append_trace(hist_full2, 4, 1)
    fig.append_trace(hist_last2, 4, 2)
    fig.append_trace(trace_plot2, 6, 1)
    fig.append_trace(trace_plot_mean2, 6, 1)


    fig['layout']['xaxis1'].update(title='Values')
    fig['layout']['xaxis2'].update(title='Values')
    fig['layout']['xaxis3'].update(title='Time')


    fig['layout']['yaxis1'].update(title='Density')
    fig['layout']['yaxis2'].update(title='Density')
    fig['layout']['yaxis3'].update(title='Value')

    fig['layout']['xaxis4'].update(title='Values')
    fig['layout']['xaxis5'].update(title='Values')
    fig['layout']['xaxis6'].update(title='Time')


    fig['layout']['yaxis4'].update(title='Density')
    fig['layout']['yaxis5'].update(title='Density')
    fig['layout']['yaxis6'].update(title='Value')

    fig['layout'].update(title='Gibbs Sampling', height=3000)

    plot(fig, filename='plots/Q3_Summary.html')


```

On convergence both histogram looks like:



Histogram on convergence



Trace plots for plain and blocked gibbs sampling

The burning time for blocked gibbs sampling is around 250 rounds and for plain gibbs sampling it is around 3500 rounds. To run plain gibbs sampling for 4000 rounds takes 5 ms and to run blocked gibbs sample for 750 rounds (500 samples) takes 1.5 ms.

Note that if we run blocked gibbs sampling for same rounds(4000) it takes 9.2 ms, almost twice the time taken by plain gibbs sampling.

Blocked gibbs sampler essentially samples multiple variables in a single step. This intuitively exploits higher order couplings between two subsets of variables. For instance, we can see that X_1, X_2 and X_3, X_4 pairs having same values is higher. But ϕ tell that X_2, X_3 having same values is more likely. When we have state $(0, 1, 1, 1)$. In X_1, X_2 is picked in blocked gibbs sampling round, they both are more likely going to end up with values 1. On other hand if X_2 gets picked in plain gibbs sampling it much more likely to go to 0 than 1.