COMP0080 Graphical Models - Assignment

University College London - Department of Computer Science

Team: YES
Student IDs: 19081939, 21110083, 21174897, 21079102, 20084999

Contents

1 Part 1

1.1 Question 1

Consider the two earthquake/explosion problem, exercise 1.22 from "Bayesian Reasoning and Machine Learning" by Barber.

Given N sensors equally spaced on the circumference of a circle and two explosion locations, s_1 and s_2, occurring somewhere within the circle.

The measurement of each sensor, v_i, is

$$v_i = \frac{1}{d_{1,i}^2 + 0.1} + \frac{1}{d_{2,i}^2 + 0.1} + \sigma\epsilon_i \tag{1}$$

where $d_{1,i}^2$ is the squared euclidean distance of explosion to the ith sensor location and the first explosion: $d_{1,i}^2 = (s_{1,x} - v_{i,x})^2 + (s_{1,y} - v_{i,y})^2$, where $(s_{1,x}, s_{1,y})$ are the coordinates of explosion s_1 and $(v_{i,x}, v_{i,y})$ are the coordinates of sensor v_i. Similarly, $d_{2,i}^2$, is the squared euclidean distance of explosion to the ith sensor location and the second explosion. The ϵ_i is a normally distributed random variable with mean 0 and standard deviation 1. The noise on each sensor is assumed to be independent.

The sensor value can be viewed as a clean signal coming from two explosions is, $c_{2,i} = \frac{1}{d_{1,i}^2+0.1} + \frac{1}{d_{2,i}^2+0.1}$, plus some noise, $\sigma\epsilon_i$. Thus, the sensor measurement is distributed:

$$v_i \sim \mathbb{N}(c_{2,i}, \sigma^2) \tag{2}$$

The potential explosion locations occurring within the circle will be assumed to occur at points along a spiral. In our implementation the radius of the circle was $R = 1$ and there were $S = 2500$ points along the spiral, or potential explosion locations. The jth potential explosion location had radius $r_j = \frac{j}{S}$ with $j \in \{1, \ldots, S\}$. The angle of each explosion location was $\theta_j = 2m\pi\frac{j}{S}$, again, with $j \in \{1, \ldots, S\}$ and in our case we used $m = 25$. So, the coordinate location in Cartesian space of a potential explosion location is then:

$$s_{j,x} = r_j \cos(\theta_j) \tag{3}$$
$$s_{j,y} = r_j \sin(\theta_j) \tag{4}$$

The locations along a spiral is visualised in figure 1.

We were given $n = 30$ sensor measurement values, as found in the "EarthquakeExercise-Data.txt". The standard deviation on each sensor measurement was assumed to be $\sigma = 0.2$. We also assumed the prior locations of the explosions were uniform and independent along the spiral.

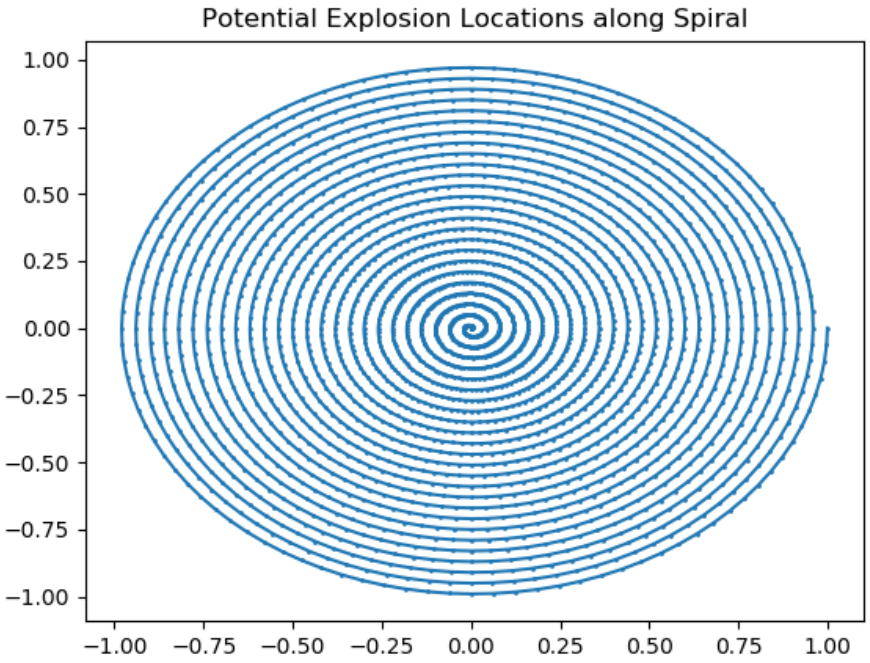


Figure 1: Potential explosion locations along spiral

1)

Calculate the posterior $p(s_1|v_{1:n})$ and draw an image that visualises the posterior.

The relationship of the explosion locations with the sensors can be represented in a belief network, as seen in figure 2.

Given the belief network representation we can write the joint probability as

$$p(s_1, s_2, v_{1:n}) = p(s_1)p(s_2)\prod_{i=1}^{n} p(v_i|s_1, s_2) \tag{5}$$

Where $p(s_1) = p(s_2) = \frac{1}{S}$ and $p(v_i|s_1, s_2) = \frac{1}{\sigma\sqrt{2\pi}} \exp(\frac{-(v_i - c_i)^2}{2\sigma^2})$

From Bayes rule and (5) we get

$$p(s_1, s_2|v_{1:n}) = \frac{p(s_1, s_2, v_{1:n})}{p(v_{1:n})} \tag{6}$$

and we can get the marginal, $p(v_{1:n})$ from (7)

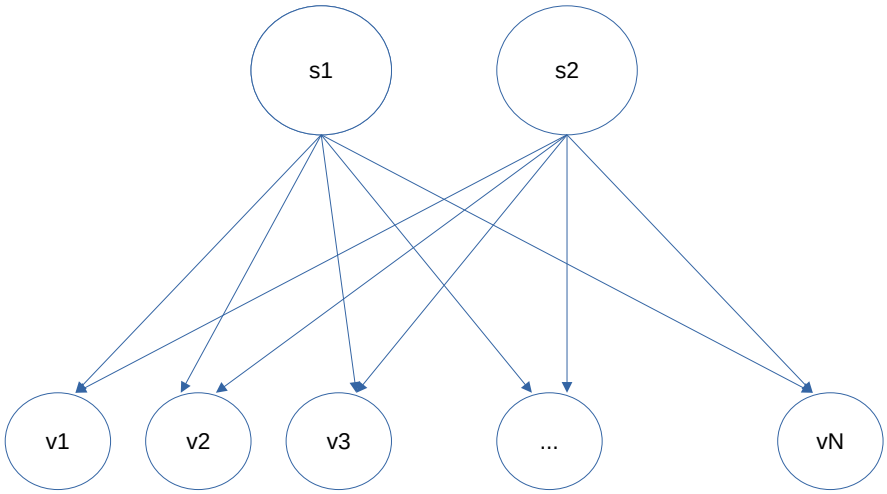


Figure 2: Belief network representation of explosions and sensors

$$p(v_{1:n}) = \sum_{s_1,s_2} p(s_1, s_2, v_{1:n}) \tag{7}$$

The posterior probability, $p(s_1|v_{1:n})$, can be calculated by marginalising (6)

$$p(s_1|v_{1:n}) = \sum_{s_2} p(s_1, s_2|v_{1:n}) \tag{8}$$

To avoid numerical issues in calculating the joint probability $p(s_1, s_2, v_{1:n})$ we use the log probability

$$\log p(s_1, s_2, v_{1:n}) = \log p(s_1) + \log p(s_2) + \sum_{i=1}^{n} \log p(v_i|s_1, s_2) \tag{9}$$

Equation 6 can be calculated using

$$p(s_1, s_2|v_{1:n}) = \frac{e^{\log p(s_1,s_2,v_{1:n})+\beta}}{\sum_{s_1,s_2} e^{\log p(s_1,s_2,v_{1:n})+\beta}} \tag{10}$$

Where β is used to avoid any underflow issues. We used $\beta = -\max(\log p(s_1, s_2, v_{1:n}))$

Equation 10 gives the probability of two explosions occurring, one at location s_1 and the other at s_2 given the sensor measurements, $v_{1:n}$. We can then marginalise over s_2, according to (8), to get the posterior probability $p(s_1|v_{1:n})$, which is represented in figure 3

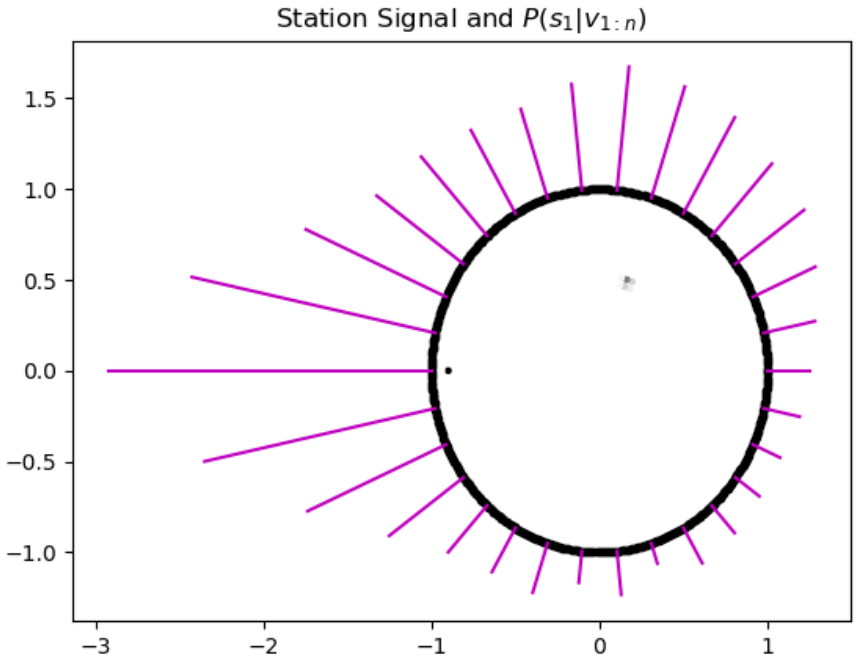


Figure 3: Sensor Signal and Posterior Probability of Explosion Location(s)

The magenta lines in figure 3 are proportional to the signal strength of each sensor located on the edge of the circle. On the interior of the circle are grey or black dots representing the posterior probability of an explosion occurring at those locations, with darker dots indicating high probability.

2)

Writing $\mathcal{H}_2$ for the hypothesis that there are two explosions and $\mathcal{H}_1$ for the hypothesis that there is only one explosion, report the value of $\log p(v_{1:n}|\mathcal{H}_2) - \log p(v_{1:n}|\mathcal{H}_1)$

The probability of getting the sensor measurements given there was only one explosion is

$$p(v_{1:n}|\mathcal{H}_1) = \sum_{s_1} p(s_1, v_{1:n}) \tag{11}$$

and similarly for the two explosion hypothesis

$$p(v_{1:n}|\mathcal{H}_2) = \sum_{s_1}\sum_{s_2} p(s_1, s_2, v_{1:n}) \tag{12}$$

For the one explosion hypothesis the joint distribution can be expressed as

$$p(s_1, v_{1:n}) = p(s_1) \prod_{i=1}^{n} p(v_i|s_1) \tag{13}$$

For the one explosion case the clean signal is $c_{1,i} = \frac{1}{d_{1,i}^2+0.1}$ and the ith sensor value would be distributed $v_i \sim \mathbb{N}(c_{1,i}, \sigma^2)$.
For the two explosion hypothesis we, again, have

$$p(s_1, s_2, v_{1:n}) = p(s_1)p(s_2) \prod_{i=1}^{n} p(v_i|s_1, s_2) \tag{14}$$

We can re-write (11) as

$$p(v_{1:n}|\mathcal{H}_1) = \sum_{s_1} p(s_1, v_{1:n}) = \sum_{s_1} e^{\log p(s_1, v_{1:n})} \tag{15}$$

To avoid any underflow issues arising due to the largest $\log p(s_1, v_{1:n}) << 0$

$$p(v_{1:n}|\mathcal{H}_1) = \sum_{s_1} e^{\log p(s_1, v_{1:n})} \tag{16}$$

$$= \frac{e^{\beta_1}}{e^{\beta_1}} \sum_{s_1} e^{\log p(s_1, v_{1:n})} \tag{17}$$

$$= \frac{1}{e^{\beta_1}} \sum_{s_1} e^{\log p(s_1, v_{1:n}) + \beta_1} \tag{18}$$

So then

$$\log p(v_{1:n}|\mathcal{H}_1) = \log(\sum_{s_1} e^{\log p(s_1, v_{1:n}) + \beta_1}) - \beta_1 \tag{19}$$

where we used $\beta_1 = -\max(\log p(s_1, v_{1:n}))$
A similarly for the two explosion hypothesis

$$\log p(v_{1:n}|\mathcal{H}_2) = \log(\sum_{s_1, s_2} e^{\log p(s_1, s_2, v_{1:n}) + \beta_2}) - \beta_2 \tag{20}$$

where we used $\beta_2 = -\max(\log p(s_1, s_2, v_{1:n}))$
From (19) and (20) and using the setup as before along with the same sensor measurement values, $v_{1:n}$, we found

$$\log p(v_{1:n}|\mathcal{H}_2) - \log p(v_{1:n}|\mathcal{H}_1) = 746.42 \tag{21}$$

3)

Assuming we have no prior preference, that is $p(\mathcal{H}_1) = p(\mathcal{H}_2) = const.$*, explain why* $\log p(v_{1:n}|\mathcal{H}_2) - \log p(v_{1:n}|\mathcal{H}_1)$ *relates to the probability that there are two explosions compared to only one.*

The difference of probabilities can be expressed as

$$\log p(v_{1:n}|\mathcal{H}_2) - \log p(v_{1:n}|\mathcal{H}_1) = \log(\frac{p(v_{1:n}|\mathcal{H}_2)}{p(v_{1:n}|\mathcal{H}_1)}) \tag{22}$$

This log of the ratio of probabilities can be directly interpreted as being proportional to have probable one hypothesis relative to another. In our case we found the difference of log probabilities to be quite large, so one could say the probability of $\mathcal{H}_2$ being true is relatively much greater than the probability of $\mathcal{H}_1$ being true.

Additionally the differences in log probabilities of nested hypothesis, or models, is used in the log likelihood ratio test, also referred to as the Wilk's Test [1]:

$$LR = 2(loglike(\mathcal{H}_2) - loglike(\mathcal{H}_1)) \tag{23}$$

where in our case $loglike(\mathcal{H}_2) = \log(p(v_{1:n}|\mathcal{H}_2)$ and $loglike(\mathcal{H}_1) = \log(p(v_{1:n}|\mathcal{H}_1)$.

The likelihood ratio test is asymptotically χ^2 distributed with degrees of freedom equal to the difference in the number of free parameters in each hypothesis. In our case the degrees of freedom would be 1.

Given high LR test statistic value of 1492.82 we would reject $\mathcal{H}_2$ in favour of $\mathcal{H}_2$, even at a very low p-value, say $p = 0.001$

4)

If we assumed that there were k *explosions, explain the computational complexity of calculating* $\log p(v_{1:n}|\mathcal{H}_k)$

Calculating $p(v_{1:n}|\mathcal{H}_k)$ would be $O(N^k)$ where N is the number of potential explosion points to be considered. This is the case because the probability of all k combinations of the N potential explosion locations would need to be calculated, leading to N^k calculations. This can be further seen from (11) and (12) where adding more potential explosion locations would require more summations in the marginalisation.

1.2 Question 2

(a)

Given that all friends need to arrive in time to not miss the train, to answer this questions we want to find the joint probability of all friends to be less than T_0 minutes delayed with a probability of at least 90%, where T_0 represents the number of minutes before 21:05 that we ask our friends to meet. From the provided PDF distribution, we constructed a table summarising all PDFs and the resulting CDFs used throughout question 2 as shown in 1.

	PDF Punct.	CDF Punct.	PDF N. Punct.	CDF N. Punc.	CDF Marginalised
0	0.70	0.70	0.50	0.50	0.63
$(0, 5)$	0.10	0.80	0.20	0.70	0.77
$[5, 10)$	0.10	0.90	0.10	0.80	0.87
$[10, 15)$	0.07	0.97	0.10	0.90	0.95
$[15, 20)$	0.02	0.99	0.05	0.95	0.98
> 20	0.01	1.00	0.05	1.00	1.00

Table 1: Overview of used PDF and CDFs

The problem can therefore be described as follows.

$$P(D_1, D_2, ..., D_N < T_0(N)) \geq 90\% \tag{24}$$

Given that the delay of each friend is distributed identically and independent, we can factorise the joint pdf and drop the subscript.

$$P(D < T_0(N))^N \geq 90\% \tag{25}$$

Table 2 shows the joint PDF for different numbers of N. The answer to part a) is to choose a number just above the upper bound of the bracket in the distribution that will achieve a CDF of above 90%. The reason for this is that we do not know the distribution within each time interval and thus conservatively assume that all probability mass is at the top end of the interval. For $N = 3$ the answer is $T(3) = 15$, for $N = 5$ the answer is $T(5) = 20$ and for $N = 10$ the answer is $T(10) = 20$. We choose a number just above the open interval bound of the bracket, since that we are not given the distribution within each bracket. For example, we only know that the probability of a friend being delayed $D \in [15, 20)$ is 2%, however we do not know the distribution of the delay within this bracket. To ensure that the joint CDF is above 90% we therefore choose the value just outside of the respective bracket.

(b)

Now an additional random variable has been added. We therefore need to model the joint PDF of all friends and all prior distributions of the state of the random variable Z but

N	0	(0,5)	[5,10)	[10,15)	[15,20)	> 20
1	0.7	0.8	0.9	0.97	0.99	1
2	0.49	0.64	0.81	0.9409	0.9801	1
3	0.343	0.512	0.729	**0.912673**	0.970299	1
4	0.2401	0.4096	0.6561	0.885293	0.960596	1
5	0.16807	0.32768	0.59049	0.858734	**0.95099**	1
6	0.117649	0.262144	0.531441	0.832972	0.94148	1
7	0.0823543	0.209715	0.478297	0.807983	0.932065	1
8	0.057648	0.167772	0.430467	0.783743	0.922745	1
9	0.0403536	0.134218	0.38742	0.760231	0.913517	1
10	0.0282475	0.107374	0.348678	0.737424	**0.904382**	1

Table 2: Probability of not missing the train for chosen time intervals and number of friends

marginalise out Z. This is equivalent to weighting the PDF of the delay for each state (punctual and not punctual) by the respective prior probability of each state. The CDF for this marginalised distribution is shown in Table 1. The question asks for the probability of missing the train which is equal to 1 minus the probability of not missing the train. The probability of missing the train is therefore given by:

$$1 - \sum_{Z} [P(D_1, D_2, ..., D_N < T_0(N) | Z_1, Z_2, ..., Z_N) \times P(Z_1, Z_2, ..., Z_N)] \tag{26}$$

Again, we can factorise the joint PDF given that the distribution is independent for each friend. This also means that the state Z_j is independent of Z_k, as well as Z_j is independent of D_k for any $j \neq k$.

$$= 1 - \{\sum_{Z} [P(D < T_0(N) | Z) \times P(Z)]\}^N \tag{27}$$

Alternatively, one can also derive the joint PDF as done in 32 by summing the product of likelihood and prior over all possible number of non-punctual friends.

The results of each probability to not miss the train given a respective $T_0(N)$ and the number of friends are shown in table 3. The probability of missing the train is calculated as 1 minus the probability of not missing the train. For $T(3) = 15$ the probability of missing the train is 15.16%. For $T(5) = 20$ the probability of missing the train is 11.13%. For $T(10) = 20$ the probability of missing the train is 21.03%.

N	0	(0,5)	[5,10)	[10,15)	[15,20)	> 20
1	0.633333	0.766667	0.866667	0.946667	0.976667	1
2	0.401111	0.587778	0.751111	0.896178	0.953878	1
3	0.254037	0.45063	0.650963	**0.848382**	0.931621	1
4	0.16089	0.345483	0.564168	0.803135	0.909883	1
5	0.101897	0.26487	0.488946	0.760301	**0.888652**	1
6	0.0645348	0.203067	0.423753	0.719751	0.867917	1
7	0.0408721	0.155685	0.367252	0.681365	0.847666	1
8	0.0258856	0.119358	0.318285	0.645025	0.827887	1
9	0.0163942	0.091508	0.275847	0.610624	0.808569	1
10	0.010383	0.0701562	0.239068	0.578057	**0.789703**	1

Table 3: (Marginal) Probability of not missing the train for chosen time intervals and number of friends

(Bonus)

For this question we want to find the posterior distribution on the number of friends that are not punctual (NPC) given the observation that we missed the train for $N = 5$ and $T(5) = 20$. That is, we would like to find

$$P(NPC = i|Miss) = \frac{P(Miss|NPC = i) \times P(NPC = i)}{P(Miss)}, for\ i = 0, 1, 2, ..., 5 \tag{28}$$

Given that each friend is independently determined to be punctual or not punctual, the prior of the count variable NPC that represents the number of friends that are not punctual is given by a binomial distribution with $p = \frac{1}{3}$ as given in b).

$$P(NPC = i) = B(k = i, N = 5, p = \frac{1}{3}) = \binom{5}{i} \times \left(\frac{1}{3}\right)^i \times \left(\frac{2}{3}\right)^{5-i} \tag{29}$$

The likelihood or conditional probability of missing the train is given by 1 minus the conditional joint PDF of not missing the train given the number of non punctual friends. Again we can simplify the joint PDF by making use of the fact that the distributions of delays and the priors are identical and independent.

$$P(Miss|NPC = i) = P(D_1, D_2, ..., D_5 > T_0(5)|NPC = i) \tag{30}$$

Which can be simplified to

$$P(Miss|NPC = i) = 1 - P(D \leq T_0(5)|Z = 0)^i \times P(D \leq T_0(5)|Z = 1)^{5-i} \tag{31}$$

The conditional CDF of each friend being delayed depending on being punctual or not punctual are as given in b) and summarised in table 4.

Finally, the evidence or the probability of missing the train can be taken from b), or can be computed by summing the numerator of (28) over all possible number of friends that are not punctual NPC $= \{0, 1, 2, 3, 4, 5\}$.

$$P(Miss) = \sum_{i=0}^{5} [P(D_1, D_2, ..., D_5 > T_0(5) | NPC = i) \times P(NPC = i)] \tag{32}$$

The posterior distribution of the number of friends that are not punctual is calculated using (28) and computed in table 4

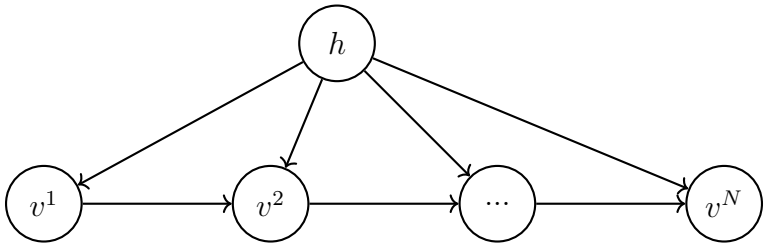
	NPC = 0	NPC = 1	NPC = 2	NPC = 3	NPC = 4	NPC = 5
$P(NPC = i \mid Miss)$	0.0580	0.2585	0.3675	0.2361	0.0716	0.0084
$P(NPC = i)$	0.1317	0.3292	0.3292	0.1646	0.0412	0.0041

Table 4: Posterior distribution of number of Not Punctual friends

1.3 Question 3

(a)

We denote our set of $N = 500$ sequences as $V = \{v^n_{1:T}, n = 1...N\}$, where all our sequences are of the same length $T = 100$. We assume the sequences are generated from a three component, $H = 3$, mixture of Markov chains, where the discrete hidden variable $\text{dom}(h) = \{1, 2, 3\}$ indexes the Markov chain $\prod_t p(v_t \mid v_{t-1}, h)$. The graphical model for this problem is given below.



The parameters for this model, for all h in $\text{dom}(h)$ are:

- $p(h)$: the probability of of the source station
- $p(v_1 \mid h)$: the probability of the first observed state v_1 of a sequence given a station h
- $p(v_t \mid v_{t-1}, h)$: the probability of an observed state at time t, v_t, given the previous observed state v_{t-1} and the source h

The aim of the EM algorithm for this problem will be to find the maximum likelihood estimate for these parameters (such that we are maximizing the probability of all the data, given our parameters) in order to assign clusters according to $p(h \mid v^n_{1:T})$, the posterior probability of a particular cluster given a sequence.

For the EM algorithm, we first initialize our parameter values to some initial guess. We then iterate through the following steps until convergence:

(1) E-step: calculate a posterior probability over the hidden variables h for each sequence v^1, v^2, v^3, given by

$$p^{old}(h \mid v^n_{1:T}) \propto p(h)p(v^n_{1:T} \mid h) = p(h) \prod_{t=1}^{T} p(v^n_t \mid v^n_{t-1}, h) \tag{33}$$

In practice we will work with logs to avoid numerical underflow issues:

$$\log p^{old}(h \mid v^n_{1:T}) = \log p(h) + \sum_{t=1}^{T} \log p(v^n_t \mid v^n_{t-1}, h) \tag{34}$$

(2) M-step: update the model parameters such that the energy is maximised. Given the portion of the free energy equation that the parameter $p(h)$ contributes to, and by viewing that maximising free energy is equivalent to minimising KL divergence, the optimal choice for updating $p(h)$ is given by

$$p^{new}(h) \propto \sum_{n=1}^{N} p^{old}(h \mid v_{1:T}^n) \tag{35}$$

Similarly the update for $p(v_t \mid v_{t-1}, h)$ is given by

$$p^{new}(v_t = i \mid v_{t-1} = j, h = k) \propto \sum_{n=1}^{N} p^{old}(h = k \mid v_{1:T}^n) \sum_{t=2}^{T} \mathbb{I}[v_t^n = i]\mathbb{I}[v_{t-1}^n = j] \tag{36}$$

And the update for $p(v_1 \mid h)$ is given by

$$p^{new}(v_1 = i \mid h = k) \propto \sum_{n=1}^{N} p^{old}(h = k \mid v_{1:T}^n)\mathbb{I}[v_1^n = i] \tag{37}$$

(b)

All values are rounded to 3 decimal places.
Log Likelihood of the data for the learned parameters: -45254.510
Learned parameters:

	$p(h)$
$h = 1$	0.508
$h = 2$	0.192
$h = 3$	0.300

Table 5: Learned parameter $p(h)$

	$p(v_1 \mid h = 1)$	$p(v_1 \mid h = 2)$	$p(v_1 \mid h = 3)$
$v_1 = 0$	0.461	0.427	0.192
$v_1 = 1$	0.331	0.573	0.419
$v_1 = 2$	0.208	0.000	0.389

Table 6: Learned parameter $p(v_1 \mid h)$

	$p(v_t \mid v_{t-1}=0, h=1)$	$p(v_t \mid v_{t-1}=1, h=1)$	$p(v_t \mid v_{t-1}=2, h=1)$
$v_1=0$	0.070	0.137	0.511
$v_1=1$	0.495	0.226	0.437
$v_1=2$	0.435	0.367	0.052

Table 7: Learned parameter $p(v_t \mid v_{t-1}, h=1)$

	$p(v_t \mid v_{t-1}=0, h=2)$	$p(v_t \mid v_{t-1}=1, h=2)$	$p(v_t \mid v_{t-1}=2, h=2)$
$v_1=0$	0.389	0.241	0.545
$v_1=1$	0.148	0.515	0.018
$v_1=2$	0.463	0.244	0.437

Table 8: Learned parameter $p(v_t \mid v_{t-1}, h=2)$

	$p(v_t \mid v_{t-1}=0, h=3)$	$p(v_t \mid v_{t-1}=1, h=3)$	$p(v_t \mid v_{t-1}=2, h=3)$
$v_1=0$	0.060	0.130	0.418
$v_1=1$	0.140	0.324	0.428
$v_1=2$	0.800	0.546	0.154

Table 9: Learned parameter $p(v_t \mid v_{t-1}, h=3)$

	Station 1	Station 2	Station 3
Sequence 1	0.000	1.000	0.000
Sequence 2	1.000	0.000	0.000
Sequence 3	0.000	0.000	1.000
Sequence 4	0.000	0.000	1.000
Sequence 5	0.000	0.000	1.000
Sequence 6	1.000	0.000	0.000
Sequence 7	0.000	1.000	0.000
Sequence 8	0.000	0.000	1.000
Sequence 9	0.000	0.000	1.000
Sequence 10	0.000	0.000	1.000

Table 10: Posterior distribution of which stations the first 10 sequences come from

(c)

EM can converge to a local maximum or saddle point of the likelihood of the data given the model, and not necessarily the global maximum. Therefore we ran the algorithm several times and chose the solution with the highest likelihood.

To initialize our parameters, we randomly drew samples from a uniform distribution over the interval $[0, 1)$ and normalised the samples so that they would reflect a probability distribution over all possible values of the parameter:

- For the parameter $p(h)$ we randomly drew three numbers (since $H = 3$) from a uniform distribution over $[0, 1)$, then normalized those three values by dividing them over the sum of all the values.
- For $p(v_1 \mid h)$, we randomly drew $3*3 = 9$ values from the uniform distribution to reflect the probability of the first value given station, where we have 3 observable states s and 3 possible source stations h. We normalize these values so that for each h we have a probability distribution describing the probability that each state s is the first observed state, v_1.
- Finally, for $p(v_t \mid v_{t-1}, h)$, we randomly drew $3 * 3 * 3 = 27$ values from the uniform distribution to reflect the probability of an observation at time t, given the previous observation and the source. Again we have 3 observable current states and 3 possible sources, this time with 3 observable previous states. Here we normalize across h and v_{t-1} so that for each h and v_{t-1} we have a probability distribution for each observable value of v_t.

When normalising probabilities, sometimes all normalised probabilities can be 0, leading to division by 0. Here we incremented the unnormalised probabilities by a small value to avoid division by 0. Additionally, we can run into numerical underflow issues during the E-step when computing the product of many terms. To avoid this, we work with logs so that we instead compute the sum of terms, as outlined in (b).

(d)

We ran an experiment by initializing our parameters uniformly such that $p(h) = 1/3$ for all h, $p(v_1 \mid h) = 1/3$ for all h and v_1, and $p(v_t \mid v_{t-1}, h) = 1/3$ for all h, v_t, and v_{t-1}. We observed that the value of $p(h)$ remained the same (1/3) after the EM algorithm converged, and that $p(v_1 \mid h)$ and $p(v_t \mid v_{t-1}, h)$ values were updated, but remained uniform across h.

$p(h)$ remains the same because, since we initialized all three values at 1/3, each time we update the *lph_old* variable in our algorithm all three output values will be the same. Converting an array of three equal values into a probability distribution will give three outputs with a value of 1/3, keeping $p(h)$ constant across iterations. If all h's have the same probability, the other parameters will also have the same probability across h.

See tables below for outputs (all values rounded to 3 decimal points):

	$p(h)$
$h = 1$	0.333
$h = 2$	0.333
$h = 3$	0.333

Table 11: Learned parameter $p(h)$ with uniform initialization

	$p(v_1 \mid h=1)$	$p(v_1 \mid h=2)$	$p(v_1 \mid h=3)$
$v_1=0$	0.374	0.374	0.347
$v_1=1$	0.404	0.404	0.404
$v_1=2$	0.222	0.222	0.222

Table 12: Learned parameter $p(v_1 \mid h)$ with uniform initialization

	$p(v_t \mid v_{t-1}=0, h=1)$	$p(v_t \mid v_{t-1}=1, h=1)$	$p(v_t \mid v_{t-1}=2, h=1)$
$v_1=0$	0.160	0.145	0.487
$v_1=1$	0.306	0.283	0.350
$v_1=2$	0.534	0.572	0.163

Table 13: Learned parameter $p(v_t \mid v_{t-1}, h=1)$ with uniform initialization

	$p(v_t \mid v_{t-1}=0, h=2)$	$p(v_t \mid v_{t-1}=1, h=2)$	$p(v_t \mid v_{t-1}=2, h=2)$
$v_1=0$	0.160	0.145	0.487
$v_1=1$	0.306	0.283	0.350
$v_1=2$	0.534	0.572	0.163

Table 14: Learned parameter $p(v_t \mid v_{t-1}, h=2)$ with uniform initialization

	$p(v_t \mid v_{t-1}=0, h=3)$	$p(v_t \mid v_{t-1}=1, h=3)$	$p(v_t \mid v_{t-1}=2, h=3)$
$v_1=0$	0.160	0.145	0.487
$v_1=1$	0.306	0.283	0.350
$v_1=2$	0.534	0.572	0.163

Table 15: Learned parameter $p(v_t \mid v_{t-1}, h=3)$ with uniform initialization

Log likelihood of the data for these parameters = -49473.547

We conclude that no, it would not be a good idea to initialize the parameters uniformly.

Additionally, it's important to try out different parameter initializations to ensure the EM solution is not a local minimum or saddle point, but initializing the parameters uniformly each time would not allow us to do that and could leave us stuck in a local minimum or saddle point.

2 Part 2

2.1 Question 1

Encoding Algorithm

Let H be the $(N-K) \times N$ parity check matrix, where N is the codeword length and K is the length of the message blocks. To build a systematically encoding matrix G, we can apply Gaussian elimination to H over modulo 2 to find its reduced row echelon form, $\hat{H}$. Given that the number of rows of the matrix will always be less than the number of columns, $\hat{H}$ will always have non-pivot columns. Indeed, the left-most columns will form the pivot columns while the right-most columns will form the non-pivot columns. If the row echelon form doesn't have that identity block, we can simply swap columns in the fashion of renaming variables.
Thus, in reduced row echelon form, we can think of $\hat{H}$ as

$$\hat{H} = [I_{N-K} P] \tag{38}$$

where I_{N-K} is the identity matrix containing the pivot columns while P is the matrix of right-most non-pivot columns. We can then swap the these two constituent matrices around to form

$$\hat{H} = [P I_{N-K}]. \tag{39}$$

Using this matrix of non-pivot columns, P, we can then form the generator matrix for the encoding as follows

$$G = [I_K P]^T \tag{40}$$

where $\hat{H}Gt = 0$ for each t where t is a given message block to be transmitted.

Results

By employing this encoding algorithm, we can then build a systematically encoding matrix G for the given parity check matrix,

$$H = \begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 \end{pmatrix} \tag{41}$$

We find that the $\hat{H}$ in the form of $\hat{H} = [P I_{N-K}]$ is then,

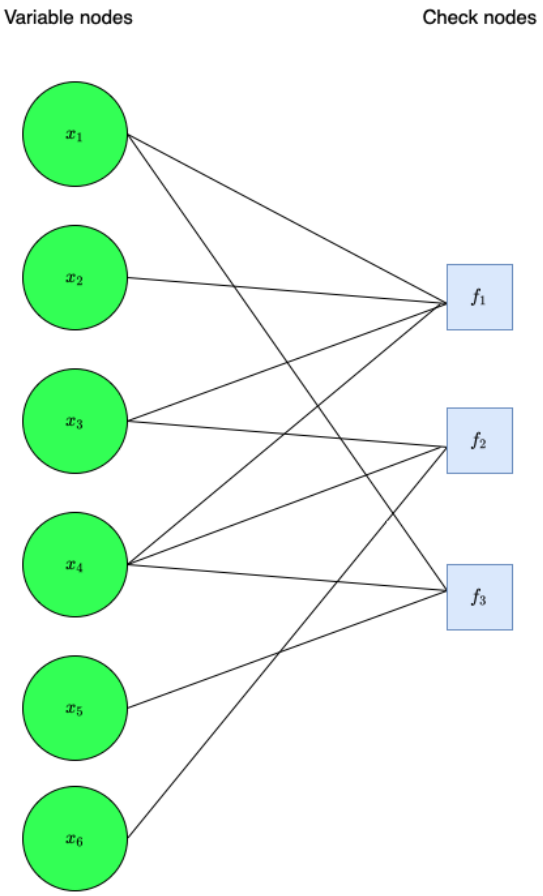
$$\hat{H} = \begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} \tag{42}$$

While the generator matrix, G, is

$$G = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix} \tag{43}$$

Together, $\hat{H}Gt = 0$ as expected.

2.2 Question 2



$$p(x) = f_1(x_1, x_2, x_3, x_4) f_2(x_3, x_4, x_6) f_3(x_1, x_4, x_5)$$

Figure 4: Factor graph and distribution

From Figure 4,
$p(x) \propto f(x) = f_1(x_1, x_2, x_3, x_4) f_2(x_3, x_4, x_6) f_3(x_1, x_4, x_5)$,
where $f_i(x_{N(i)}) = I[\sum_{N(i)} x_i = 0]$
To decode the message, the posterior $p(x|y) \propto p(y|x)p(x)$ is maximised.
Given,

$$P(y|x) = \prod_{n=1}^{N} p(y_n|x_n) = \prod_{n=1}^{N} p^{x_n - y_n}(1-p)^{x_n + y_n + 1} \tag{44}$$

Therefore,

$$P(x|y) \propto \prod_{n=1}^{N} p^{x_n - y_n}(1-p)^{x_n + y_n + 1} I[Hx = 0] \tag{45}$$

2.3 Question 3

To decode the coded message, we use the Loopy Belief Propagation for Binary Symmetric Channel (BSC) in logarithmic domain, as described by Shokrollahi [3]. By decoding in logarithmic domain, we are able to reduce the computational complexity. Below is a description of the algorithm used. For the application of this algorithm, see the Python code we wrote in (Appendix B, Questions 3 and 4).

Binary Symmetric Channel (BSC) Model

Let x be a transmitted vector and y the received word. For the BSC model, each bit is independently flipped with probability p. The likelihood $P(y|x)$ is the probability that y is received, given that x was transmitted. For the BSC model, this likelihood is given as,

$$P(y|x) = \prod_{n=1}^{N} p(y_n|x_n) = \prod_{n=1}^{N} p^{x_n - y_n}(1-p)^{x_n + y_n + 1} \tag{46}$$

We then try to estimate the marginals $P(x_n|y)$ and obtain the decoding $x^* = argmax P(x_n|y)$.

Loopy Belief Propagation Algorithm in Logarithmic Domain

To decode the codeword with the BSC model, we use the Loopy Belief Propagation algorithm in logarithmic domain. The following description of the algorithm follows closely to that given by Shokrollahi [3], with similar notation used.

For this algorithm, we have an $(N-K) \times N$ parity check matrix, H, which can be thought of as similar to the bipartite graph seen in Question 2. That is, there are N left nodes which form our message nodes and $N-K$ right nodes which form our check nodes. If there is a connection between the ith check node and the jth message node in the graph, then the element (i, j) of H will be 1. If not, then element (i, j) will be 0.

With the message passing algorithm Loopy Belief Propagation, we are attempting to decode the received word y by iteratively passing the messages (probabilities) from the message nodes to the check nodes and back again, updating our messages as we go along. The message passed from a given message node v to a given check node c is the probability that v has a certain value given the observed value of that message node. Meanwhile the message passed from the check node c to the message node v is the probability that v has a certain value given all of the messages passed to c in the previous round from message nodes other than v. For each round of iteration, after we have updated the messages, we sum the check node messages for the message node to check node pass to form the decoded vector, d. If the message has been correctly decoded, then $Hd^T = 0$.

Algorithm Steps

For the first step of the algorithm, we initialise the log-likelihood of all message nodes, over the corresponding bits of the received word, y. If the observed bit of y is 0, the log-likelihood is given as

$$l(y_n = 0|x_n) = ln\left(\frac{1-p}{p}\right) \tag{47}$$

where p is the probability of each bit being flipped. If the observed bit of y is 1, the log-likelihood is given as

$$l(y_n = 1|x_n) = ln\left(\frac{p}{1-p}\right). \tag{48}$$

After constructing the vector of log-likelihoods, $\hat{y}$, we then carry out a row element-wise multiplication of the parity check matrix H and the vector $\hat{y}$. This is to form the matrix of log-likelihoods of the message nodes over the bits of the received word, M_v. That is,

$$M_v = H \circ \hat{y}. \tag{49}$$

With the initial log-likelihoods of the message nodes constructed, we now reach the iterative stage of the algorithm. In these steps, we are passing the messages from the message nodes to the check nodes and back again, updating the messages each time. For a given iteration round l, a given message node v and a given check node c, define the message passed from c to v at iteration round l, as $m_{cv}^{(l)}$. Similarly, we define the message passed from v to c as $m_{vc}^{(l)}$. In the first step of the iterative rounds, we calculate $m_{cv}^{(l)}$ as follows,

$$m_{cv}^{(l)} = ln\left(\frac{1+\prod_{v'\in V_c\setminus\{v\}}\tanh(m_{v'c}^{(l)}/2)}{1-\prod_{v'\in V_c\setminus\{v\}}\tanh(m_{v'c}^{(l)}/2)}\right) \tag{50}$$

where V_c is the set of all message nodes connected to check node c. In the first round, we

define $m_{vc}^{(l)}$ as m_v, where m_v is an element of M_v. We calculate this message pass for each message and check node, to form the matrix $M_{cv}^{(l)}$. We next update each $m_{vc}^{(l)}$ as follows,

$$m_{vc}^{(l)} = m_v + \sum_{c' \in C_v \setminus \{c\}} m_{c'v}^{(l-1)} \tag{51}$$

where C_v is the set of all check nodes connected to message node v. We do this for each message and check node to form the matrix $M_{vc}^{(l)}$. We next sum all messages along the check nodes of $M_{vc}^{(l)}$. If the message is less than or equal to 0, we set the decoded bit as 1. If it is positive we set the bit to 0. With the decoded vector d, we then calculate Hd^T and if it equals 0, we return the decoded vector. If it does not equal 0, we repeat the previous steps, updating $M_{cv}^{(l)}$ and $M_{vc}^{(l)}$ until either we reach the maximum number of iterations or $Hd^T = 0$.

Results

Running this algorithm for the given parity check matrix H_1, received word y_1, probability of bit being flipped $p = 0.1$ and maximum number of iterations of 20, the algorithm was able to successfully converge in 8 iterations.

2.4 Question 4

By translating the first 248 bits of the decoded message from Question 3, we found that the message translated to be "Happy Holidays! Dmitry&David :)".

3 Part 3

In this question we perform different inference methods for the Ising model, defined on a 10 x 10 lattice. In particular, we will find the joint probability distribution of nodes $x_{1,10}$ and $x_{10,10}$ using 1) exact inference, 2) variational inference and 3) sampling for three different parameter choices of the model, $\beta = 4$, $\beta = 1$ and $\beta = 0.01$.

3.1 Question 1

In this question we will perform exact inference on the Ising model using message passing on a factor graph. Before being able to perform message passing we first convert the graph into a singly connected graph. This is achieved by clustering all nodes in a column of the lattice into one cluster node. We define X_i as the cluster node that comprises all nodes $x_{j,i}$ for $j = 1, 2, ..., 10$. One cluster node consists of 10 binary nodes and thus has $2^{10} = 1024$ states, which represent all possible combination of states from the binary nodes that make up the cluster node. The graph of the transformed model thus consists of 10 singly connected cluster nodes X_i for $i = 1, 2, ..., 10$. The factor graph representation of this distribution is shown in figure 5.

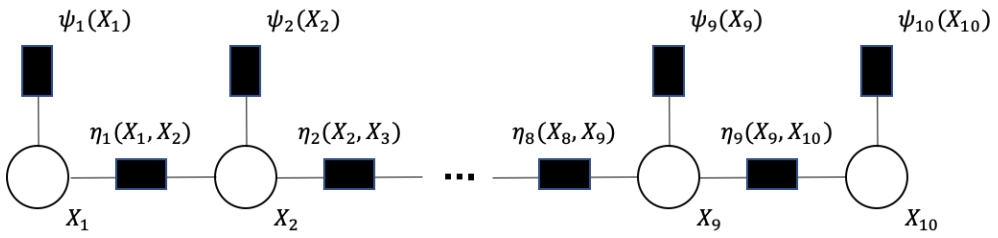


Figure 5: Factor Graph of Clustered Nodes

The factor ψ_i represents the potential for each state coming from the connections of the member nodes within the cluster node X_i. This could be thought of as the vertical connections between the nodes in the lattice. It is calculated as follows:

$$\psi_i(X_i) = \prod_{j=1}^{N-1} \phi(x_j^i, x_j^i), \tag{52}$$

where x_j^i is the j-th member node of cluster node i.

The factor η_i represents the potential for each state in cluster node X_i coming from the connections to the member nodes from the cluster node X_{i-1} left to it. This could be thought of as the horizontal connections between the nodes in the lattice. This transition probability from each state of X_{i-1} to each state of X_i forms a $2^N \times 2^N$ matrix. It is calculated as follows:

$$\eta(X_{i-1}, X_i) = \prod_{j=1}^{N-1} \phi(x_j^i, x_j^{i-1}), \tag{53}$$

In order to calculate the joint probability distribution of nodes $x_{1,10}$ and $x_{10,10}$, we will first calculate $P(X_{10})$, the marginal distribution of cluster node X_{10} by passing all the messages in the graph to this node.

We will pass messages from left to right starting with the message from factor ψ_1 to node X_1, which is simply the factor ψ_1 itself. Once the message from the ψ_{i-1} factor is passed, the message from node X_{i-1} to the factor η_{i-1} is simply the product of all the messages received by node X_{i-1}. When passing the message from factor η_{i-1} to node X_i, we marginalise over node X_{i-1}. Finally, to complete all the messages to node X_i we multiply the message from factor η_i by the message from factor ψ_i. The full description of the message passing processes is as follows:

The initial messages from factor ψ_1 to node X_1 and from node X_1 to factor η_1 are defined as:

$$\mu_{\psi_1 \to X_1} = \psi_1(X_1) \tag{54}$$
$$\mu_{X_1 \to \eta_1} = \mu_{\psi_1 \to X_1} \tag{55}$$

For all the subsequent nodes $i = 2, ..., 10$ the message passing process is defined as:

$$\mu_{\eta_{i-1} \to X_i} = \sum_{X_{i-1}} [\eta_{i-1}(X_{i-1}, X_i) \times \mu_{X_{i-1} \to \eta_{i-1}}] \tag{56}$$
$$\mu_{X_{i-1} \to \eta_{i-1}} = \mu_{\eta_{i-2} \to X_{i-1}} \times \mu_{\psi_{i-1} \to X_{i-1}} \tag{57}$$
$$\mu_{\psi_{i-1} \to X_{i-1}} = \psi_i(X_{i-1}) \tag{58}$$
$$\mu_{\eta_0 \to X_1} = 1 \tag{59}$$

Once all messages have been calculate we can derive the non-normalised marginal probability $P^*(X_{10})$ of cluster node X_{10} as

$$P^*(X_{10}) = \mu_{\eta_9 \to X_{10}} \times \mu_{\psi_{10} \to X_{10}} \tag{60}$$

Once the non-normalised marginal probability of cluster node X_{10} has been calculated, we can now calculate the joint probability distribution of nodes $x_{1,10}$ and $x_{10,10}$ by summing the potential of the relevant states in cluster node X_{10} and subsequently normalising. For example, we sum all states in cluster node X_{10} that have $x_{1,10} = 1$ and $x_{10,10} = 1$ to know the marginal potential of that state. We do so for all 4 possible configurations of the two binary nodes $x_{1,10}$ and $x_{10,10}$ and divide by the sum of the four potentials to derive the marginal

probability distribution.

Importantly, for very large or very small β we could run into numerical problems when calculating the messages, as given that products of factors are calculated, numbers can become either very small or very large. Therefore, we convert above outlined process to log-messages.

This is done by re-defining the messages from nodes to factors as follows:

$$\lambda_{X_{i-1}\to\eta_{i-1}} = \log(\mu_{X_{i-1}\to\eta_{i-1}}) \tag{61}$$

Messages from factors to nodes are re-defined as:

$$\lambda_{\eta_{i-1}\to X_i} = \lambda^*_{X_{i-1}\to\eta_{i-1}} + \log(\sum_{X_{i-1}}[\eta_{i-1}(X_{i-1}, X_i) \times \exp(\lambda_{X_{i-1}\to\eta_{i-1}} - \lambda^*_{X_{i-1}\to\eta_{i-1}})]), \tag{62}$$

where $\lambda^*_{X_{i-1}\to\eta_{i-1}} = max_s(\lambda_{X_{i-1=s}\to\eta_{i-1}})$ represents the highest log-message for a given state s and is subtracted to increase numerical stability.

Finally, log-messages are no longer multiplied but added to derive the non-normalised log probability.

$$\log(P^*(X_{10})) = \lambda_{\eta_9\to X_{10}} + \lambda_{\psi_{10}\to X_{10}} \tag{63}$$

This non-normalised log probability is then transformed back to non-normalised probabilities by similarly subtracting the maximum non-normalised log probability to avoid numerical problems:

$$\hat{P}(X_{10}) = \exp(\log(P^*(X_{10})) - \max[\log(P^*(X_{10} = s))]), \tag{64}$$

where s represents all possible states of node X_{10}.

The results of the exact inference are shown in Table 16, rounded to four decimal places. In the table, the rows represent the states for $x_{1,10}$ and the columns represent the states of $x_{10,10}$. For small $\beta \to 0$, the probability of each state converges to be equal. Conversely, for large $\beta \to \infty$ the probability for a situation in which both nodes have different states converges to zero, with equal probability that both nodes are either in state 1 or -1. Lastly, for moderate β, the probability of both nodes being in the same state is slightly higher compared to both being in separate states. This is driven by the potential function which assigns higher potential to situations in which both nodes have the same state. The higher the β the more the model favours nodes to be in the same state.

$\beta = 0.01$	-1	1
-1	0.2500	0.2500
1	0.2500	0.2500

$\beta = 1$	-1	1
-1	0.2804	0.2196
1	0.2196	0.2804

$\beta = 4$	-1	1
-1	0.4997	0.003
1	0.003	0.4997

Table 16: Exact Inference for $P(x_{1,10}, x_{10,10})$ for different β parameters

3.2 Question 2

Given an Ising model of a $n \times n$ grid with potential function for two neighbouring nodes, (x_i, x_j):

$$\phi(x_i, x_j) = \exp[\beta \mathbb{1}[x_i = x_j]] \tag{65}$$

The probability of given state, $\boldsymbol{x} \in \{-1, 1\}^N$, is then

$$P(\boldsymbol{x}) = \frac{1}{Z} \prod_{i>j} \phi(x_i, x_j) \tag{66}$$

where Z is a normalising constant.

Following from the "Information Theory, Inference, and Learning Algorithms" by David MacKay [2] we can use *mean field theory*, to find an approximation to $P(\boldsymbol{x})$ with a simpler one, namely $Q(\boldsymbol{x}; \boldsymbol{\theta})$ where $\boldsymbol{\theta}$ are adjustable parameters. Mean field theory is a special case of a general variational free energy approach of Feynman and Bogoliubov [2].

Consider the relative entropy, otherwise known as the Kullback-Leibler divergence, of distributions Q and P

$$D_{KL}(Q||P) = \sum_{\boldsymbol{x}} Q(\boldsymbol{x}; \boldsymbol{\theta}) \log \frac{Q(\boldsymbol{x}; \boldsymbol{\theta})}{P(\boldsymbol{x})} \tag{67}$$

Note: from Gibbs' inequality we have $D_{KL}(Q||P) \geq 0$ and $D_{KL}(Q||P) = 0$ if and only if $Q = P$

Equation 66 can be expressed as

$$P(\boldsymbol{x}) = \frac{1}{Z} \exp[-\beta E(\boldsymbol{x})] \tag{68}$$

where the energy function in, $E(\boldsymbol{x})$, is

$$E(\boldsymbol{x}) = -\frac{1}{2} \sum_{i} \sum_{j \in ne(i)} \mathbb{1}[x_i = x_j] \tag{69}$$

since

$$\begin{aligned} P(\boldsymbol{x}) &= \frac{1}{Z}\prod_{i>j}\phi(x_i, x_j) \\ &= \frac{1}{Z}\prod_{i>j}\exp[\beta\mathbb{1}[x_i = x_j]] \\ &= \frac{1}{Z}\exp[\beta\sum_{i>j}\mathbb{1}[x_i = x_j]] \\ &= \frac{1}{Z}\exp\big[\frac{\beta}{2}\sum_{i}\sum_{j\in ne(i)}\mathbb{1}[x_i = x_j]\big] \\ &= \frac{1}{Z}\exp[-\beta E(\boldsymbol{x})] \end{aligned} \tag{70}$$

where, for clarity, we've used

$$\sum_{i>j}\mathbb{1}[x_i = x_j] = \frac{1}{2}\sum_{i}\sum_{j\in ne(i)}\mathbb{1}[x_i = x_j] \tag{71}$$

where $ne(i)$ denotes the set of neighbours of i and the $\frac{1}{2}$ was introduced to avoid double counting.

Minimisation Problem

Combining (67) and (68) and expanding we get

$$\begin{aligned} D_{KL}(Q||P) &= \sum_{\boldsymbol{x}} Q(\boldsymbol{x};\boldsymbol{\theta})\log\frac{Q(\boldsymbol{x};\boldsymbol{\theta})}{P(\boldsymbol{x})} \\ &= \sum_{\boldsymbol{x}} Q(\boldsymbol{x};\boldsymbol{\theta})\log Q(\boldsymbol{x};\boldsymbol{\theta}) + \beta\sum_{\boldsymbol{x}} Q(\boldsymbol{x};\boldsymbol{\theta})E(\boldsymbol{x}) - \log(Z) \end{aligned} \tag{72}$$

Here we can see if we want to find a $\boldsymbol{\theta}$ that minimises $D_{KL}(Q||P)$, giving a $Q(\boldsymbol{x};\boldsymbol{\theta})$ that is close, as measured by relative entropy, to $P(\boldsymbol{x})$ we need to find:

$$\hat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} D_{KL}(Q||P) \tag{73}$$

$$= \arg\min_{\boldsymbol{\theta}} \beta\sum_{\boldsymbol{x}} Q(\boldsymbol{x};\boldsymbol{\theta})E(\boldsymbol{x}) + \sum_{\boldsymbol{x}} Q(\boldsymbol{x};\boldsymbol{\theta})\log Q(\boldsymbol{x};\boldsymbol{\theta}) \tag{74}$$

the $\log(Z)$ was dropped as it does not dependent on $\boldsymbol{\theta}$.

Now, lets name some terms in (73). Let the expected energy under distribution $Q(\boldsymbol{x};\boldsymbol{\theta})$, be

$$\langle E(\boldsymbol{x})\rangle_Q = \sum_{\boldsymbol{x}} Q(\boldsymbol{x};\boldsymbol{\theta})E(\boldsymbol{x}) \tag{75}$$

and notice

$$S_Q = -\sum_{\boldsymbol{x}} Q(\boldsymbol{x};\boldsymbol{\theta}) \log Q(\boldsymbol{x};\boldsymbol{\theta}) \tag{76}$$

is the entropy of distribution Q.
The *variational free energy* can be expressed as

$$\beta\tilde{F}(\boldsymbol{\theta}) = \beta\langle E(\boldsymbol{x})\rangle_Q - S_Q \tag{77}$$

Approximating Distribution

Let a separable approximating distribution, with adjustable parameters $\boldsymbol{\theta} = \boldsymbol{a}$, be:

$$\begin{aligned} Q(\boldsymbol{x};\boldsymbol{a}) &= \frac{1}{Z_Q}\exp[\sum_i a_i x_i] \\ &= \frac{1}{Z_Q}\prod_i \exp[a_i x_i] \end{aligned} \tag{78}$$

Again, given $\boldsymbol{x} \in \{-1,1\}^N$, and $Q(\boldsymbol{x};\boldsymbol{a})$ is a probability distribution we have

$$\begin{aligned}
\sum_{\boldsymbol{x}} Q(\boldsymbol{x};\boldsymbol{a}) &= 1 \\
&= \sum_{\boldsymbol{x}} \frac{1}{Z_Q} \prod_{i=1}^{N} \exp[a_i x_i] \\
&= \frac{1}{Z_Q} \sum_{x_1,\ldots,x_N} \prod_{i=1}^{N} \exp[a_i x_i] \\
&= \frac{1}{Z_Q} \sum_{x_1} \cdots \sum_{x_N} \prod_{i=1}^{N} \exp[a_i x_i] \\
&= \frac{1}{Z_Q} \sum_{x_1} \exp[a_1 x_1] \cdots \sum_{x_N} \exp[a_N x_N] \\
&= \frac{1}{Z_Q} \prod_{i=1}^{N} \sum_{x_i} \exp[a_i x_i] \\
&= \frac{1}{Z_Q} \prod_{i=1}^{N} (\exp[a_i] + \exp[-a_i])
\end{aligned} \tag{79}$$

so then $Z_Q = \prod_{i=1}^{N} (\exp[a_i] + \exp[-a_i])$ and (78) can be expressed as

$$\begin{aligned}
Q(\boldsymbol{x};\boldsymbol{a}) &= \prod_{i=1}^{N} \frac{\exp[a_i x_i]}{\exp[a_i] + \exp[-a_i]} \\
&= \prod_{i=1}^{N} q(x_i; a_i)
\end{aligned} \tag{80}$$

Entropy of Q

Let $q_k = q(x_k = 1; a_k)$ and, so then, $1 - q_k = q(x_k = -1; a_k)$. Recall, the binary entropy is then

$$\begin{aligned}
H_2(q_k) &= \sum_{x_k} q(x_k; a_k) \log \frac{1}{q(x_k; a_k)} \\
&= q_k \log \frac{1}{q_k} + (1 - q_k) \log \frac{1}{1 - q_k}
\end{aligned} \tag{81}$$

Given (76), (80) and (81) can make the following derivation

$$
\begin{aligned}
S_Q &= -\sum_{\boldsymbol{x}} Q(\boldsymbol{x};\boldsymbol{a}) \log Q(\boldsymbol{x};\boldsymbol{a}) \\
&= -\sum_{\boldsymbol{x}} \big(\prod_{i=1}^{N} q(x_i;a_i)\big) \sum_{k=1}^{N} \log q(x_k;a_k) \\
&= -\sum_{\boldsymbol{x}} \sum_{k=1}^{N} \big(\prod_{i=1}^{N} q(x_i;a_i)\big) \log q(x_k;a_k) \\
&= -\sum_{k=1}^{N} \sum_{\boldsymbol{x}} \big(\prod_{\substack{i=1 \\ i\neq k}}^{N} q(x_i;a_i)\big) q(x_k;a_k) \log q(x_k;a_k) \\
&= -\sum_{k=1}^{N} \sum_{\boldsymbol{x}\setminus x_k} \big(\prod_{\substack{i=1 \\ i\neq k}}^{N} q(x_i;a_i)\big) \sum_{x_k} q(x_k;a_k) \log q(x_k;a_k) \\
&= -\sum_{k=1}^{N} \sum_{\boldsymbol{x}\setminus x_k} \big(\prod_{\substack{i=1 \\ i\neq k}}^{N} q(x_i;a_i)\big)(-H_2(q_k)) \\
&= \sum_{k=1}^{N} \langle H_2(q_k)\rangle_{Q_{-k}} \\
&= \sum_{k=1}^{N} H_2(q_k)
\end{aligned}
\tag{82}
$$

In (82) the $\langle H_2(q_k)\rangle_{Q_{-k}}$ is the expectation of $H_2(q_k)$ over distribution Q_{-k} which excludes the x_k element of $\boldsymbol{x}$, making the expectation $H_2(q_k)$ equivalent to the expectation of a constant, i.e. equal to itself.

Expected Energy under Q

Combining (69) and (75) we have

$$
\langle E(\boldsymbol{x})\rangle_Q = \sum_{\boldsymbol{x}} Q(\boldsymbol{x};\boldsymbol{a})\big(-\frac{1}{2}\sum_{i}\sum_{j\in ne(i)} \mathbb{1}[x_i = x_j]\big) \tag{83}
$$

and note

$$\begin{aligned}\sum_{\boldsymbol{x}} Q(\boldsymbol{x};\boldsymbol{a})\mathbb{1}[x_i=1] &= q_i \\ \sum_{\boldsymbol{x}} Q(\boldsymbol{x};\boldsymbol{a})\mathbb{1}[x_i=-1] &= 1-q_i\end{aligned} \tag{84}$$

and also

$$\mathbb{1}[x_i=x_j] = \mathbb{1}[x_i=1]\mathbb{1}[x_j=1] + \mathbb{1}[x_i=-1]\mathbb{1}[x_j=-1] \tag{85}$$

so then

$$\begin{aligned}\langle \mathbb{1}[x_i=x_j]\rangle_Q &= \sum_{\boldsymbol{x}} Q(\boldsymbol{x};\boldsymbol{a})\mathbb{1}[x_i=x_j] \\ &= \sum_{\boldsymbol{x}} Q(\boldsymbol{x};\boldsymbol{a})\big(\mathbb{1}[x_i=1]\mathbb{1}[x_j=1] + \mathbb{1}[x_i=-1]\mathbb{1}[x_j=-1]\big) \\ &= q_iq_j + (1-q_i)(1-q_j) \\ &= 2q_iq_j - q_i - q_j + 1\end{aligned} \tag{86}$$

since under Q each x_i and x_j is independent. Equation 83 is then

$$\begin{aligned}\langle E(\boldsymbol{x})\rangle_Q &= \sum_{\boldsymbol{x}} Q(\boldsymbol{x};\boldsymbol{a})\big(-\frac{1}{2}\sum_i \sum_{j\in ne(i)} \mathbb{1}[x_i=x_j]\big) \\ &= -\frac{1}{2}\sum_i \sum_{j\in ne(i)} \sum_{\boldsymbol{x}} Q(\boldsymbol{x};\boldsymbol{a})\mathbb{1}[x_i=x_j] \\ &= -\frac{1}{2}\sum_i \sum_{j\in ne(i)} \langle \mathbb{1}[x_i=x_j]\rangle_Q \\ &= -\frac{1}{2}\sum_i \sum_{j\in ne(i)} 2q_iq_j - q_i - q_j + 1\end{aligned} \tag{87}$$

Minimising Variational Free Energy

Combining (77), (82) and (87) we have

$$\begin{aligned}\beta\tilde{F}(\boldsymbol{a}) &= \beta\langle E(\boldsymbol{x})\rangle_Q - S_Q \\ &= \beta\big[-\frac{1}{2}\sum_i \sum_{j\in ne(i)} 2q_iq_j - q_i - q_j + 1\big] - \sum_{k=1}^{N} H_2(q_k)\end{aligned} \tag{88}$$

We can apply coordinate descent to minimise the variational free energy with respect to parameters $\boldsymbol{a}$.

Given

$$q_m = \frac{\exp[a_m]}{\exp[a_m] + \exp[-a_m]} = \frac{1}{1 + \exp[-2a_m]} \tag{89}$$

the (partial) derivative of the binary entropy with respect to a_m is

$$\begin{aligned} \frac{\partial H_2(q_m)}{\partial a_m} &= \frac{\partial H_2(q_m)}{\partial q_m} \frac{\partial q_m}{\partial a_m} \\ &= \log \frac{1 - q_m}{q_m} \frac{\partial q_m}{\partial a_m} \\ &= -2a_m \frac{\partial q_m}{\partial a_m} \end{aligned} \tag{90}$$

and the (partial) derivative of the expected energy with respect to a_m is

$$\begin{aligned} \frac{\partial \langle E(\boldsymbol{x}) \rangle_Q}{\partial a_m} &= \frac{\partial}{\partial a_m} \Big(-\frac{1}{2} \sum_i \sum_{j \in ne(i)} 2q_i q_j - q_i - q_j + 1 \Big) \\ &= -\sum_{j \in ne(m)} \frac{\partial}{\partial a_m} \big[2q_m q_j - q_m - q_j + 1 \big] \\ &= -\frac{\partial q_m}{\partial a_m} \sum_{j \in ne(m)} (2q_j - 1) \end{aligned} \tag{91}$$

In the second line in (91) the $\frac{1}{2}$ was dropped because the double counting is no longer occurring when we select only the neighbours of m. Finally, the partial derivative of the variational free energy with respect to a_m is then

$$\begin{aligned} \frac{\partial \beta \tilde{F}(\boldsymbol{a})}{\partial a_m} &= \beta \frac{\partial \langle E(\boldsymbol{x}) \rangle_Q}{\partial a_m} - \frac{\partial \sum_{i=1}^N H_2(q_i)}{\partial a_m} \\ &= \beta \Big(-\frac{\partial q_m}{\partial a_m} \sum_{j \in ne(m)} (2q_j - 1) \Big) - \frac{\partial H_2(q_m)}{\partial a_m} \\ &= \frac{\partial q_m}{\partial a_m} \Big(2a_m - \beta \sum_{j \in ne(m)} (2q_j - 1) \Big) \end{aligned} \tag{92}$$

Coordinate descent can now be applied by successively minimising the variational free energy with respect to each coordinate until convergence. This is done by holding all parameters in

$\boldsymbol{a}$ fixed except a given a_m and then finding the value of a_m that will minimise $\beta\tilde{F}(\boldsymbol{a})$, where $\tilde{F}(\boldsymbol{a})$ is a convex function of a_m, that is:

$$\begin{aligned}\frac{\partial\beta\tilde{F}(\boldsymbol{a})}{\partial a_m} &= 0 \\ 0 &= \frac{\partial q_m}{\partial a_m}\Big(2a_m - \beta\sum_{j\in ne(m)}(2q_j-1)\Big) \\ 2a_m &= \beta\sum_{j\in ne(m)}(2q_j-1) \\ a_m &= \frac{\beta}{2}\sum_{j\in ne(m)}(2q_j-1)\end{aligned} \tag{93}$$

This gives the update to apply until a minimum is found.

Results

Mean field approximation using coordinate descent on the 10×10 Ising model grid was applied for $\beta \in \{0.01, 1.0, 4.0\}$. For each β the approximation was generated 1000 times with the conditional probabilities $P(x_{1,10}, x_{10,10})$ determined from each resulting Q, and then averages were taken for the 1000 conditional probabilities. The Q distribution was initialised each time with uniform random values in $[0, 1]$. Coordinate descent to applied to update a_m was applied to nodes in a random order for each step. The results can be seen in Table 17.

$\beta = 0.01$	-1	1
-1	0.25	0.25
1	0.25	0.25

$\beta = 1$	-1	1
-1	0.334	0.159
1	0.166	0.341

$\beta = 4$	-1	1
-1	0.345	0.17
1	0.158	0.327

Table 17: Mean Field Approximation for $P(x_{1,10}, x_{10,10})$ for different β parameters

The asymmetries of the values in Table 17 can be attributed to the simulated nature of how they were generated. It can be seen the values for $\beta = 1.0$ and $\beta = 4.0$ are quite similar. It was noted that final variational free energy of many of the Q did not achieve the global minimum. As seen in Figure 6 for both $\beta = 1.0$ and $\beta = 4.0$, the process appeared to get stuck on several local minima.

Figure 7 show examples of Q's that were found for $\beta = 1.0$ for both global minimum and local minima variational free energy.

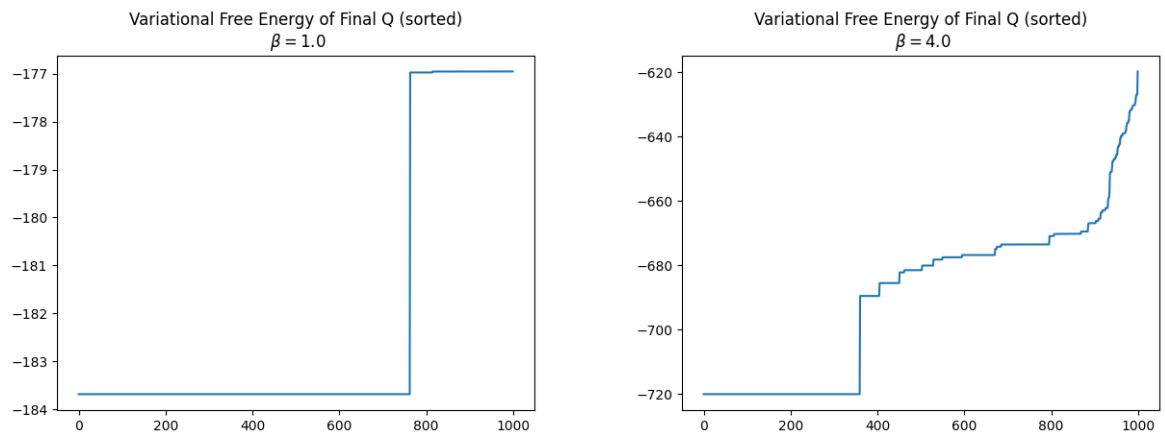


Figure 6: Variational Free Energy of Q found using Mean Field Approximations

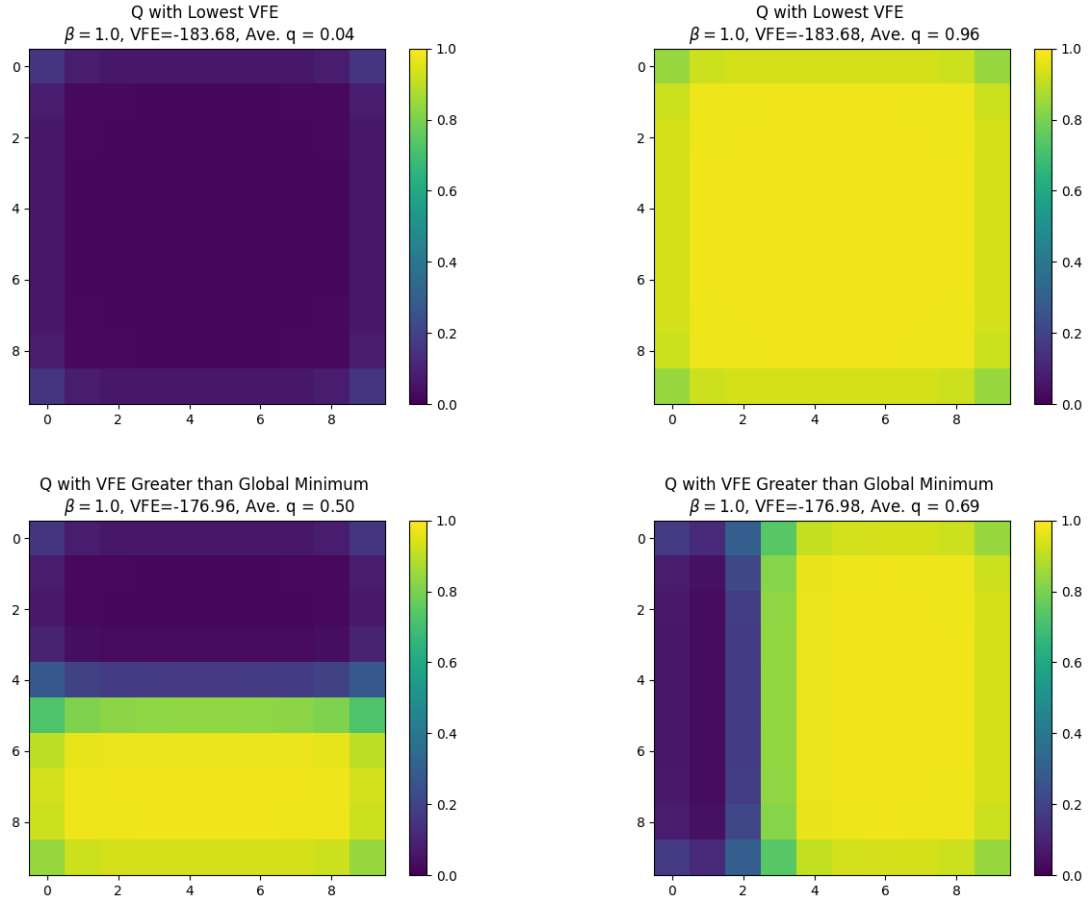


Figure 7: Example of Q's using Mean Field Approx. with $\beta = 1.0$ Top: at Global Min., Bottom: Local Min.

Similarly, Figure 7 show examples of Q's that were found for $\beta = 4.0$ for both global minimum and local minima variational free energy.

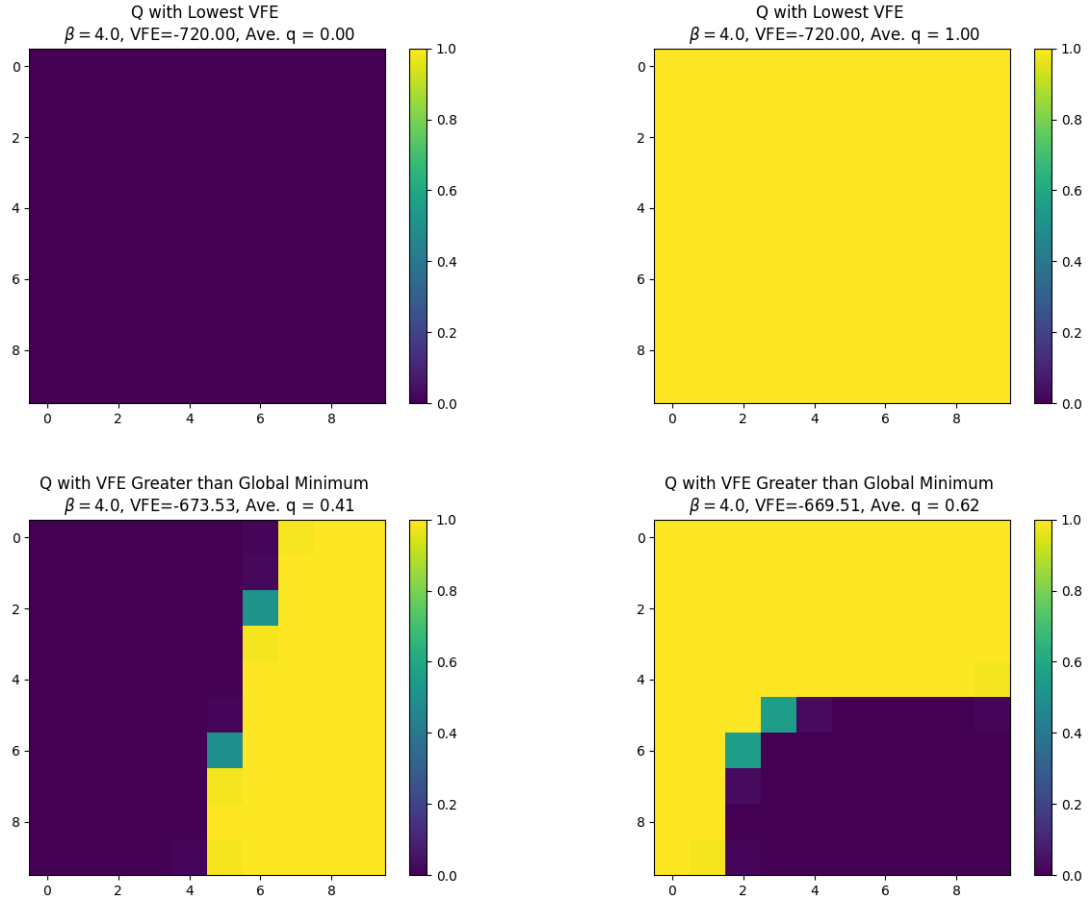


Figure 8: Example of Q's using Mean Field Approx. with $\beta = 4.0$ Top: at Global Min., Bottom: Local Min.

If the Q's that found a local minima were excluded in the conditional probability calculations we get the results in Table 18.

$\beta = 0.01$	-1	1
-1	0.25	0.25
1	0.25	0.25

$\beta = 1$	-1	1
-1	0.355	0.131
1	0.131	0.383

$\beta = 4$	-1	1
-1	0.51	0.0003
1	0.0003	0.489

Table 18: Mean Field Approximation for $P(x_{1,10}, x_{10,10})$ for different β parameters using only Q's that gave a global minimum variational free energy

When excluding the Q's that did not reach the variational free energy global minimum, the

$\beta = 4.0$ is very close to the exact inference case seen earlier. Although the $\beta = 1.0$ still differs from the exact inference case, this can be expected due to the approximating nature of mean fields. Due to stochastic nature of finding mean field approximations, because of the random initialisation and updating process, one would expect a slightly different results for Tables 17 and 18 if the process was run again.

3.3 Question 3

In this question we will perform approximate inference on the Ising model using Gibbs sampling, a Markov Chain Monte Carlo (MCMC) method. This is achieved by updating the state of each node x_i for $i = 1, ..., 100$ in the lattice sequentially by sampling a state from its conditional distribution $P(x_i|x_j), j \in Ne(i)$, where $Ne(i)$ returns the set of neighbours of node i. Sampling from the conditional distribution of a node in the Ising model is simple, as the state of one node only depends on its direct neighbours and therefore at most 4 other nodes. The transition probability is thus defined as $T(x \to x') = \pi_i p(x'_i|x_{\setminus i})$, where π_i is the probability of updating node i. Given that we update each node sequentially, after updating node i all the nodes $\setminus i$ remain the same. We therefore have that $x'_{\setminus i} = x_{\setminus i}$ and it is easy to show that the detailed balance condition holds, meaning that the probability distribution that is sampled from the Markov Chain is invariant. For small enough β parameter the distribution is ergodic, however we further discuss potential problems with respect to ergodicity for larger β values below.

The steps in the Gibbs sampling process to create one Monte Carlo consisting of the sampled data points are as follows.

1) Initialise the lattice by randomly assigning each node x_i an initial state of -1 or 1. The initial state makes up the first sample in the Markov Chain.

2) Pick a random order in which all nodes are updated sequentially. This orders is reshuffled after each node has been updated once, i.e. after one new sample has been added to the chain.

3) For the node currently being updated, calculate the conditional distribution $p(x_i|x_j), \forall j \in Ne(i)$. The conditional distribution is derived by first calculating the potential of each state $x_i = 1$ and $x_i = -1$ to get a normalising factor. The normalised potential is the conditional distribution, for state $x_i = 1$ the conditional distribution is calculated as follows:

$$p(x_i = 1|x_j) = \frac{\prod_{j \in Ne(i)} \phi(x_i = 1, x_j)}{\prod_{j \in Ne(i)} \phi(x_i = 1, x_j) + \prod_{j \in Ne(i)} \phi(x_i = -1, x_j)} \tag{94}$$

4) For the node currently being updated, using the conditional probability distribution, sample a state $s \in \{-1, 1\}$ by uniformly sampling a number $c \in [0, 1]$. If $c < p(x_i = 1|x_j)$ set $x_i = 1$, otherwise set $x_i = -1$.

5) Proceed with updating all the other remaining nodes sequentially, following steps 3) and 4).

6) Once all nodes have been updated, the grid consisting of the states of each node x_i, for $i = 1, ..., 100$, is stored as a new sample. Add the sample to the Markov Chain.

7) Repeat the process starting from step 2), i.e. without re-initialising the grid but creating a new random order for updating the nodes, to produce a new sample. This process is repeated for a given number of times.

In order to perform inference on the Ising model via sampling, we follow the procedure outlined above to create a sample set or Markov Chain of size $10,000$ for each β. For each sample set, we infer the joint probability distribution $P(x_{1,10}, x_{10,10})$ by counting the number of times each of the 4 possible outcomes has occurred divided by the sample size. Importantly, we disregard the first $1,000$ samples from the Markov Chain as these initial samples are still fairly dependent on the random initialisation. We comment on the required length for the distribution to start converging below. This is commonly referred to as burn-in. The results are shown in table 19. We note that the results are fairly close to the results from exact inference for $\beta \in \{0.01, 1\}$. However, for the case $\beta = 4$, the joint probability distribution puts almost all mass on the outcome $x_{1,10} = 1, x_{10,10} = 1$ instead of having equal probability between the outcome where both nodes are in state 1 and the outcome where both nodes are in state -1 that is given by the exact inference approache. The reason is that for large β parameters, the joint probability distribution no longer is ergodic. In other words, the state space includes two states with high probability separated by states of very low probability and as such are impossible (or very unlikely) to be reached from one another. As a consequence, the Gibbs sampling either converges to the outcome $x_{1,10} = -1, x_{10,10} = -1$ or the outcome $x_{1,10} = 1, x_{10,10} = 1$. This will result in a sampled distribution that puts all probability mass in either of these states. The case for $\beta = 4$ thus represents an example where conventional Gibbs sampling might fail to represent the true probability distribution.

$\beta = 0.01$	-1	1
-1	0.24	0.24
1	0.26	0.26

$\beta = 1$	-1	1
-1	0.23	0.22
1	0.22	0.33

$\beta = 4$	-1	1
-1	0.00	0.00
1	0.00	1.00

Table 19: Gibbs Sampling for $P(x_{10,1}, x_{10,10})$ for different β parameters

Given the pathological properties of the probability distribution for $\beta = 4$, we extend the Gibbs sampling procedure by repeating the outlined steps above 100 times, i.e. creating 100 different Markov Chains. Importantly, each Markov Chain has a different random initialisation. For each chain we then infer the probability distribution as done before. However, we now create the final inferred probability distribution by averaging across the distributions inferred by each Markov Chain. The results for this approach are shown in table 20. For $\beta = 0.01$ and $\beta = 1$ the results remain similar, albeit becoming closer to the exact

inference. Therefore, sampling across chains increases the precision of the sampled distribution. Importantly however for $\beta = 4$, we now see the expected probability distribution that puts approximately 50% of mass on the state $x_{1,10} = -1, x_{10,10} = -1$ and the state $x_{1,10} = 1, x_{10,10} = 1$. Put differently, half of the Markov Chains that were generated for $\beta = 4$ converge to the outcome where both nodes have the value 1 and the other half to the outcome where both nodes have the value -1. Lastly, we note that for $\beta = 4$ the average probability distribution is not exactly split into 50% mass for each of the two outcomes but also puts some small weight on the other two possible outcomes, in which both nodes have a different state. This is caused by the fact that convergence to the most likely outcome in which both nodes have an equal state can be slow depending on the state of the full grid. As a result the sample set will include a larger than expected share of samples where both nodes have different states. We further comment on this case below.

$\beta = 0.01$	-1	1
-1	0.25	0.25
1	0.25	0.25

$\beta = 1$	-1	1
-1	0.28	0.22
1	0.22	0.28

$\beta = 4$	-1	1
-1	0.51	0.00
1	0.00	0.49

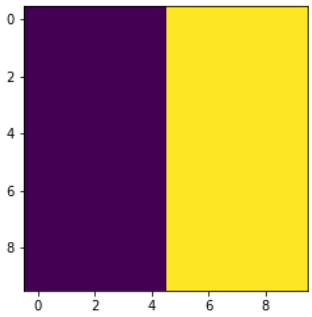
Table 20: Average of 100 sampled distributions of $P(x_{10,1}, x_{10,10})$ for different β parameters

For $\beta = 0.01$ and $\beta = 1$ a chain length of $1,000$ is sufficient for Gibbs sampling to achieve results fairly close to the exact inference case. However, for $\beta = 4$ a chain length of $1,000$, i.e. updating each node $1,000$ times, might not be sufficiently large for the system to have converged to the most probable state. In particular, we note that for certain compositions convergence can slow down a lot, which will result in a Markov Chain that has a too large number of samples for which the system has not converged yet.

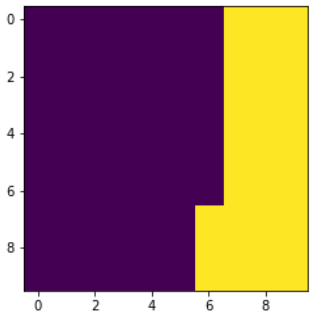
In order to further analyse the convergence behaviour, we ran an experiment in which we initialise the board to be exactly split in half. All the nodes in the first 5 columns are initialised as -1 and all nodes in the last 5 columns are initialised as 1. We then analyse the state of the model after 100, $1,000$ and $10,000$ epochs of updating. We note that starting from this slow convergence environment, the model typically only converges after $10,000$ epochs of updating, whereas after $1,000$ the model often has not converged fully yet. This means that even when disregarding the first $1,000$ samples of the Markov Chain, the sample set will include many samples in which the system has not converged yet.

To provide more information on the cause of the slow convergence, the case in which the model has developed a line that splits the node states along a straight line creates an environment in which the system converges slower. Whilst all nodes that are surrounded by nodes of identical state are remaining in the same state with a very high probability, the nodes at the splitting line are more likely to switch. As a consequence such a global state for the model means convergence is achieved much slower compared to a situation in which most nodes

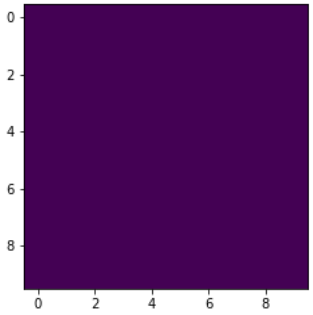
have the same state, and only a few nodes have another state. Particularly, in cases where nodes are surrounded by nodes with a different state, they are very likely to switch. One possible analogy could be that nodes that are surrounded by nodes of a different state are turned easier, whereas a situation in which there is one "front line" which splits the board in two states means that it will take a while until either side "takes over" the full board. An example run of the experiment with initial state, the lattice after $1,000$ updates of each nodes and the lattice after $10,000$ is shown in Figure 9.



(a) Initial State



(b) After 1,000 Updates



(c) After 10,000 Updates

Figure 9: Experiment results for $\beta = 4$, one example iteration

References

[1] S. S. Wilks. "The Large-Sample Distribution of the Likelihood Ratio for Testing Composite Hypotheses". In: *The Annals of Mathematical Statistics* 9.1 (1938), pp. 60–62. DOI: 10.1214/aoms/1177732360. URL: https://doi.org/10.1214/aoms/1177732360.

[2] David J. C. MacKay. *Information Theory, Inference, and Learning Algorithms*. Copyright Cambridge University Press, 2003, pp. 422–428.

[3] Amin Shokrollahi. "LDPC Codes: an Introduction". In: Jan. 2004, pp. 85–110. ISBN: 978-3-0348-9602-3. DOI: 10.1007/978-3-0348-7865-4_5.

A Part 1: Code

Question 1

```
# Exercise 1.22 in BRML - two earthquakes / explosions

import pandas as pd
import numpy as np
import numba as nb

import matplotlib.pyplot as plt
# from scipy.stats import norm

import os


@nb.guvectorize([(nb.float64[:], nb.float64[:], nb.float64[:],
                  nb.float64[:], nb.float64[:])],
                "(), (), (N), (N) -> (N)", target="parallel")
def clean_signal_1exp(vx, vy, sx, sy, out):
    """
    Get the clean signal for ONE explosion
    for station located at (vx, vy)
    with explosion located at sx and sy

    Parameters
    ----------
    vx: float
        x coordinate of station
    vy: float
        y coordinate of station
    sx, sy: ndarray
        shape (N,) of floats for x, y position for explosion position

    Returns
    -------
    out: ndarray
        shape (N,) of floats giving the clean signal at sensor

    Note
    ----
    function is NumPy universal function, so can use broadcasting
```

```
    """

    for i in range(len(sx)):
        # ith center location
        sxi, syi = sx[i], sy[i]
        # (squared) distance
        d2i = (vx[0] - sxi)**2 + (vy[0] - syi)**2
        # clean signal is then
        out[i] = 1 / (d2i + 0.1)


@nb.guvectorize([(nb.float64[:], nb.float64[:], nb.float64[:],
                  nb.float64[:], nb.float64[:, :])],
                "(), (), (N), (N) -> (N, N)", target="parallel")
def clean_signal_2exp(vx, vy, sx, sy, out):
    """
    Get the clean signal for TWO explosion
    for station located at (vx, vy)
    with explosion located at sx and sy

    Parameters
    ----------
    vx: float
        x coordinate of station
    vy: float
        y coordinate of station
    sx, sy: ndarray
        shape (N,) of floats for x, y position for explosion position

    Returns
    -------
    out: ndarray
        shape (N,) of floats giving the clean signal at sensor

    Note
    ----
    function is NumPy universal function, so can use broadcasting
    """

    for i in range(len(sx)):
        # ith center location
        sxi, syi = sx[i], sy[i]
```

```
        # (squared) distance to ith center
        d2i = (vx[0] - sxi)**2 + (vy[0] - syi)**2

        for j in range(len(sy)):

            # TODO: could have an if i > j condition here to cut calc in half
            # jth center location
            sxj, syj = sx[j], sy[j]
            # (squared) distance to ith center
            d2j = (vx[0] - sxj)**2 + (vy[0] - syj)**2

            # clean signal is then
            out[i, j] = ( 1 / (d2i + 0.1)) + ( 1 / (d2j + 0.1) )


def sensor_location(N, R=1):
    """
    Sensor location on edge of circle
    - will specify the locations of N evenly spaced locations
    Parameters
    ----------
    N: int
        number of sensors / locations on circle
    R: float, default 1
        radius of circle

    Returns
    -------
    tuple: (x,y)
        cartesian coordinates of the N sensors locations
        each has shape (N,)

    """
    theta_sensor = 2 * np.pi * R * (np.arange(1, N + 1) / N)
    v_x = np.cos(theta_sensor)
    v_y = np.sin(theta_sensor)

    return v_x, v_y


def potential_explosion_location(S, R=1, rate=25):
    """
```

```
    Potential Explosion locations within circle
    - values places along a spiral

    Parameters
    ----------
    S: int
        number of points to place within circle
    R: float, default 1
        radius of circle
    rate: int or float, default 25
        defines the number of complete rotation of the spiral

    Returns
    -------
    tuple: (x,y)
        cartesian coordinates of the S potential explosion locations
        along a spiral within circle
        x and y have shape (N,)

    """
    # NOTE: in below starting index at 1, rather than 0 to align with
    # earthquakeExerciseSetup.jl

    # (potential) quake centers
    # radius
    r_i = R * (np.arange(1, S + 1) / S)
    # angle
    theta_i = 2 * rate * np.pi * (np.arange(1, S + 1) / S)
    # x,y positions
    s_x = r_i * np.cos(theta_i)
    s_y = r_i * np.sin(theta_i)

    return s_x, s_y


def plot_circle_with_sensor_reading(v, v_x, v_y, sf=0.2, R=1):
    """
    plot a circle with sensor reading coming out radially

    Parameters
    ----------
    v: float np.array
```

```
        sensor values, shape (N,)
    v_x: float np.array
        x coordinate location of sensors, shape (N,)
    v_y: float np.array
        x coordinate location of sensors, shape (N,)
    sf: float, default 0.2
        scaling factor for sensor values
    R: float, default 1
        radius of circle

    Returns
    -------
    None

    """
    # plot circle
    for theta in np.arange(0, 2*np.pi, 0.01):
        plt.plot(np.cos(theta), np.sin(theta), ".", color=tuple([0,0,0]))

    # plot signal strength around the edge
    # from sensor location radially outward - proportional to size
    for sensor in range(len(v)):

        theta_sensor = 2 * np.pi * R * (sensor+1) / len(v)
        # clean signal strength - scaled by sf
        x_sensor = (R+sf*v[sensor,0])*np.cos(theta_sensor)
        y_sensor = (R+sf*v[sensor,0])*np.sin(theta_sensor)
        plt.plot([v_x[sensor], x_sensor],
                 [v_y[sensor], y_sensor], "-m")


@nb.guvectorize([(nb.float64[:], nb.float64[:], nb.float64[:], nb.float64[:])],
                "(), (), () -> ()", target="parallel")
def lognormpdf(x, mu, sig, out):
    """
    log of normal density function
    - made as a low memory alternative to scipy.stats.norm.logpdf
    - is a numpy universal function, so can handle broadcasting

    Parameters
    ----------
    x: float or np.array
```

```
        realisation of a random variable, assumed to be normally distributed
    mu: float
        normal random variable mean
    sig: float
        normal random variable standard deviation


    Returns
    -------
    float - log of normal probability density: log(p(x;mu, sig))

    Note:
    -----
    although 'out' is defined as parameter, the signature specifies it
    is the output (outputs are after ->), so should not be provided as input

    """
    out[0] = -0.5 * (x[0] - mu[0]) * (x[0] - mu[0]) / \
             (sig[0] * sig[0]) - np.log(sig[0]*np.sqrt(2 * np.pi))


def get_tex_path(*args,  create_dir_if_not_exist=True):
    """
    get path to 'tex' directory, contained in same directory as file
    if create_dir_if_not_exist=True will make directories """

    try:
        if __file__ == "":
            file_dir = os.getcwd()
        else:
            file_dir = os.path.dirname(__file__)
    except Exception as e:
        print("issue getting directory of file from __file__")
        file_dir = os.getcwd()

    out = os.path.join(file_dir, 'tex', *args)
    if create_dir_if_not_exist:
        os.makedirs(os.path.dirname(out), exist_ok=True)

    return out
```

```
if __name__ == "__main__":


    # -------
    # parameters
    # -------

    # number of points in spiral (points within circle)
    S = 2500
    # rate - number of rotations / loops spiral makes
    rate = 25
    # number of sensors / stations
    N = 30
    # standard deviation of sensor noise
    sd = 0.2

    plot_dir = get_tex_path("plots")

    # -------
    # locations
    # -------

    # Sensor locations
    v_x, v_y = sensor_location(N)

    # (potential) explosion locations
    s_x, s_y = potential_explosion_location(S, R=1, rate=rate)

    # plot spiral
    plt.close()
    plt.plot(s_x, s_y)
    plt.scatter(s_x, s_y, s=1)
    plt.title("Potential Explosion Locations along Spiral")
    plt.savefig(get_tex_path("plots", "explosion_locations.png"))
    # plt.show()

    # -----
    # clean signal from potential locations
    # -----

    # clean signal from two explosions
```

```
cs2 = clean_signal_2exp(v_x, v_y, s_x, s_y)

# clean signal from one explosion
cs1 = clean_signal_1exp(v_x, v_y, s_x, s_y)

# -----
# read in the station measurements values
# -----

# NOTE: expects EarthquakeExerciseData.txt to be in same
# directory as this file
try:
    v = pd.read_csv("EarthquakeExerciseData.txt", header=None).values
except FileNotFoundError as e:
    v = pd.read_csv("part1/EarthquakeExerciseData.txt", header=None).values

# ----
# 1) calculate p(s_1 | v_{1:n})
# ----

# given a BN to describe the relationship between explosion location(s)
# and sensor value, we have (for the two explosion) case
# p(s_1, s_2, v_{1:n}) = p(s_1) * p(s_2) \prod_{i=1}^{n} p(v_i | s_1, s_2)
# with p(s_1) = p(s_2) = 1 / S
# where the sensor value v_i ~ N( c_i, \sigma)
# where c_i is the clean signal is a function of s_1, s_2
# (in the two explosion case)

# log ( p(s_1, s_2, v_{1:n}) ) = log(p(s_1)) + log(p(s_2)) +
#   \sum_{i=1}^{n} log(p(v_i | s_1, s_2))

# calculate log(p(v_i | s_1, s_2))
# - the log prob of getting measurement v_i (at ith station) given explosions
# occurring at s_1, s_2

# NOTE: broadcasting v so will match dimension of the clean signal
# this can be memory intensive, but it's not permanent...
# lp2 = norm.logpdf(v[:, None], loc=cs2, scale=sd)

# less memory intensive, validated in utils
lp2 = lognormpdf(v[:, None], cs2, sd)
```

```
# validate against scipy.stats.norm:
# lp2_chk = norm.logpdf(v[:, None], loc=cs2, scale=sd)
# assert np.abs(lp2-lp2_chk).max() < 1e-10

# summing across each sensor (the first dimension) and adding 2 * log(1/S)
# we get  log ( p(s_1, s_2, v_{1:n}) )  for each s_1, s_2 pair
lp_s1_s2_v1n = np.log(1/S) + np.log(1/S) + lp2.sum(axis=0)

# using marginalisation we have
# p(s_1 | v_{1:n})  = \sum_{s_2} p(s_1, s_2 | v_{1:n})
# and from Bayes Rule
# p(s_1, s_2 | v_{1:n}) * p(v_{1:n}) =  p(s_1, s_2, v_{1:n})
# p(s_1, s_2 | v_{1:n}) =  p(s_1, s_2, v_{1:n}) /  p(v_{1:n})
# and
# p(v_{1:n}) = \sum_{s_1, s_2} p(s_1, s_2, v_{1:n})

#  p(s_1, s_2, v_{1:n}) = exp( log(  p(s_1, s_2, v_{1:n}) )
# adding (subtracting negative) term to avoid underflow
p = np.exp(lp_s1_s2_v1n - np.max(lp_s1_s2_v1n))
# p = np.exp(lp_s1_s2_v1n)
# effective scaling terms cancels when normalising
# - i.e. divide by \sum_{s_1, s_2} p(s_1, s_2, v_{1:n})
p /= p.sum()

# summing over the s_2 dimension (doesn't matter which one as
# p(s_1, s_2, v_{1:n}) is symmetric)
# to get:
# p(s_1 | v_{1:n})  = \sum_{s_2} p(s_1, s_2 | v_{1:n})
p_s1_v1n = np.sum(p, axis=1)

# plot the circle with station strength
plt.close()
plot_circle_with_sensor_reading(v, v_x, v_y)
# scatter plot of each point with the color (alpha) being proportional to the
# probability of an explosion happening at that point
plt.scatter(s_x, s_y, s=5, c=p_s1_v1n, cmap='gray_r')
plt.title("Station Signal and $P(s_1|v_{1:n})$")
plt.savefig(get_tex_path("plots", "explosion_posterior_prob.png"))
# plt.show()

# plt.plot(p_s1_v1n)
# plt.show()
```

```
# ----
# 2) calc. log(p(v_{1:n}| H_2) ) - log(p(v_{1:n}| H_1) )
# ----

# log( p(v_i | s_1) ) - for each point s_1
# lp_vi_h1 = norm.logpdf(v, loc=cs1, scale=sd)
lp_vi_h1 = lognormpdf(v, cs1, sd)

# lp_vi_h2 = norm.logpdf(v[:, None], loc=cs2, scale=sd)
lp_vi_h2 = lognormpdf(v[:, None], cs2, sd)

# log( p(s_1, v_{1:n})) = log( p(s_1)
lp_s1_v1n = np.log(1/S) + lp_vi_h1.sum(axis=0)
lp_s1_s2_v1n = 2 * np.log(1/S) + lp_vi_h2.sum(axis=0)

# convert from logs(p) to probs
# -num1 and num2 to deal with underflow
num1 = - lp_s1_v1n.max()
num2 = - lp_s1_s2_v1n.max()

# num1 = num2 = 0

p_s1_v1n = np.exp(lp_s1_v1n + num1)
p_s1_s2_v1n = np.exp(lp_s1_s2_v1n + num2)

# sum over all (scaled) probabilities and take log
lpvh1 = np.log(np.sum(p_s1_v1n)) - num1
lpvh2 = np.log(np.sum(p_s1_s2_v1n)) - num2
# lpvh1 = np.log(np.sum(p_s1_v1n))
# lpvh2 = np.log(np.sum(p_s1_s2_v1n))

print("log(p(v_{1:n}| H_2) ) - log(p(v_{1:n}| H_1) ): ")
print(lpvh2 - lpvh1)

# using probabilities directly
# p_h1 = norm.pdf(v, loc=cs1, scale=sd)
# #  p(v_i | s_1, s_2)
# p_h2 = norm.pdf(v[:, None], loc=cs2, scale=sd)
#
# # sum over the sensors
# p_h1 = np.prod(p_h1, axis=0) * (1/S)
```

```
    # p_h2 = np.prod(p_h2, axis=0) * (1/S) * (1/S)
    #
    # np.log(np.sum(p_h2)) - np.log(np.sum(p_h1))
```

Question 2

```
#Import libraries
import numpy as np #Key library for all computations
import matplotlib.pyplot as plt #Key library for all plots
import pandas as pd #Used for some plots
import seaborn as sns #Used for some plots
from time import gmtime, strftime #Used to check speed of implementation

#Number of friends from 1, 10
Ns = np.arange(1,11)
#Calculate the probability cutoff we need to achieve per person
Ts = 0.9**(1/Ns)

#Plot the cut-off needed per person
plt.scatter(Ns,Ts)
plt.xticks(ticks = Ns)
plt.show()

#Shows cutoff that we need to have for the Max-delay that each friend is not breaching
summary = pd.DataFrame(Ts, index = Ns,columns = ["Cutoffs"])
print(summary)

#Calculate the cumulative distribution
times = ["0","(0,5)","[5,10)","[10,15)","[15,20)",">20"]
probs = np.array([0.7,0.1,0.1,0.07,0.02,0.01])
#Cumulative Probability for one person
cumprobs = probs.cumsum()

#Show PDF and CDF next to each other
probs_df = pd.DataFrame(probs, index = times,columns = ["PDF - Punctual"])
probs_df["CDF - Punctual"] = probs_df["PDF - Punctual"].cumsum()
print(probs_df)

#Show fore each number of friends and each max-delay the probability that all will sta
#P(D_1,D_2,...,D_N<T) = P(D<T)^N
#We need to pick the maximum delay that gives a probability of > 90%
```

```
timescum = times
cumprobs_df = pd.DataFrame(index = Ns,columns = timescum)
for n in Ns:
  cumprobs_df.iloc[n-1,:] = cumprobs**n
print(cumprobs_df)

#Our t-Star is being 20 minutes early
t_star = "[15,20)"

#Part b
#New PDF for non-punctual
probs_notpunc = np.array([0.5,0.2,0.1,0.1,0.05,0.05])
#Prior of being punctual
prior = 2/3
#CDF for non punctual
cumprobs_notpunc = probs_notpunc.cumsum()
#New CDF where punctual is marginalised out
w_cumprobs = prior * cumprobs + (1 - prior) * cumprobs_notpunc
w_cumprobs
#Add to dataframe

probs_df["PDF - Non Punctual"] = probs_notpunc
probs_df["CDF - Non Punctual"] = cumprobs_notpunc
probs_df["CDF - Marginalised"] = w_cumprobs
print(probs_df)

#Calculate the probability of being with the max delay for each number of friends and
w_cumprobs_df = pd.DataFrame(index = Ns,columns = timescum)
for n in Ns:
  w_cumprobs_df.iloc[n-1,:] = w_cumprobs**n
print(w_cumprobs_df)

#Calculate the probability of missing the train given the time advance t_star we picke
#Just read form the table above and subtract this amount from 1
1 - w_cumprobs_df


#Part c
from scipy.stats import binom #Import to calculate binomial prior
#Number of friends
N = 5
#Chosen T* for N = 5
```

```
t_star #from above

#Prior probability of #Friends punctional
prior_NPCs = np.array([binom.pmf(x, N, 1/3, loc=0) for x in range(0,N+1)])
print(prior_NPCs)

#Calculate likelihood = Pr (Miss | #Punctual Friends) for each # Friends (0 to 5)
like_Misss_NPC = np.array([1 - probs_df.loc[t_star,"CDF - Punctual"] **(N-x) * probs_df.
print(like_Misss_NPC)

#Numerator = Joint PDF of Missing and Each punctuality count = P(Missing x PC), for ea
num_NPC = like_Misss_NPC * prior_NPCs
#Denominator -> Marginalise out PCs to get P(Miss)
denom_NPC=num_NPC.sum()
print(denom_NPC)

#Posterior of each non-punctuality count
post_NPCs = num_NPC / denom_NPC
print(post_NPCs)

#Conversion to DatFrame for Latex
columns = ["NPC = 0", "NPC = 1", "NPC = 2", "NPC = 3", "NPC = 4", "NPC = 5"]

post_NPCs_df = pd.DataFrame(post_NPCs.reshape(1,-1), columns = columns, index = ["P(NPC
post_NPCs_df.loc["P(NPC)",:] = prior_NPCs
print(post_NPCs_df)
```

Question 3

```
# adapted from mixMarkov.m from the brml toolkit

import numpy as np
import csv

# compute a probability distribution matrix
def condp(data):
    data += 2.2251e-308 # in case sum of unnormalized vaues is 0
    return (data / sum(data))

# compute exp(logp) as conditional probability distribution
def condexp(logp):
```

```
    pmax = np.max(logp, 0)
    P = logp.shape[0]
    return(condp(np.exp(logp - np.ones((P,1)) * pmax)))


sequences = []
with open('meteo1.csv') as file:
    reader = csv.reader(file, delimiter=' ')
    for row in reader:
        sequences.append(row)
sequences = np.array(sequences, dtype=int)

H = 3 # number of mixtures
V = 3 # number of states
max_it = 15 # max iterations

# EM algorithm

# initialize parameters randomly
ph = condp(np.random.uniform(size=(H,1))) # learned p(h)
pv1gh = condp(np.random.uniform(size=(V,H))) # p(v(1)|h))
pvgvh = condp(np.random.uniform(size=(V, V, H))) # p(v(t)|v(t-1), h))

llik = np.zeros(max_it) # stores the likelihood at each em loop

#em loop
for loop in range(max_it):
    #E-step
    ph_stat = np.zeros((H, 1))
    pv1gh_stat = np.zeros((V, H))
    pvgvh_stat = np.zeros((V, V, H))
    loglik = 0

    ph_old = []
    for n in range(len(sequences)):
        T = len(sequences[n])
        lph_old = np.log(ph) + np.vstack(np.log(pv1gh[sequences[n][0], :]))

        for t in range(1,T): # start on the second observation
            lph_old += np.vstack(np.log(pvgvh[sequences[n][t], sequences[n][t-1], :]))

        ph_old.append(condexp(lph_old))
```

```
            loglik += np.log(np.sum(np.exp(lph_old)))

            # get stats for M step
            ph_stat += ph_old[n]
            pv1gh_stat[sequences[n][0], :] += np.squeeze(ph_old[n])

            for t in range(1,T): # start on second obs
                pvgvh_stat[sequences[n][t], sequences[n][t-1], :] = np.squeeze(np.vstack(pvg

        # update llike for this run of loop
        llik[loop] = loglik

        #M-step
        ph = condp(ph_stat)
        pv1gh = condp(pv1gh_stat)
        pvgvh = condp(pvgvh_stat)

phgv = np.array(ph_old)

print('Learned parameters:\n')
print('p(h) = ')
print(ph)
print('\np(v(1)|h) = ')
print(pv1gh)
print('\np(v(t)|v(t-1), h) = ')
h1 = pvgvh[:,:,0]
h2 = pvgvh[:,:,1]
h3 = pvgvh[:,:,2]
print(h1)
print(h2)
print(h3)
print('\nLog likelihood of the data for these parameters = {:.3f}'.format(loglik))

print('Posterior distribution of the first 10 sequences belonging to each station\n')
for n in range(10):
    print('Sequence {}:'.format(n+1))
    print(phgv[n])
    print('\n')
```

B Part 2: Code

Question 1

```
# Graphical Models Assignment - Task 2 Question 1 - Encoder

import numpy as np

def gaus_elim_mod2(mx):
    """Function to perform Gaussian elimination with modulo 2 on the matrix
    mx and return the matrix in reduced row echelon form.

    Parameters
        - mx: The matrix that we are applying Gaussian elimination modulo 2
              over

    Return
        - mx: The matrix in RREF modulo 2"""

    # Apply Gaussian elimination to each row
    for i in range(mx.shape[0]):

        # In the column in question, find the row position of the nonzeros
        col_ones=np.where(mx[:,i] == 1)[0]

        # Apply row or column swaps if the pivot element is zero
        # If the row and column entry is zero when it should be a 1 for the
        # pivot, then swap the pivot row for the nonzero row below it
        if i not in col_ones:

            # Swap rows if there is a 1 in the pivot column after the pivot
            # element
            if len(np.where(col_ones > i)[0]) > 0:
                swapRow=min(col_ones[np.where(col_ones > i)])
                mx[[i,swapRow]]=mx[[swapRow,i]]

            # Swap columns if there are no 1s in the pivot column after the
            # pivot element
            if len(np.where(col_ones > i)[0]) == 0:
                swapCol=min(np.where(mx[i,i:] == 1)[0])+i
                mx[:,[i,swapCol]]=mx[:,[swapCol,i]]
```

```
            col_ones=np.where(mx[:,i] == 1)[0]

        # Remove pivot value from the series of ones and add pivot row to
        # each row with a 1 in the column in question
        col_ones=np.delete(col_ones,np.where(col_ones==i)[0])
        if len(col_ones) > 0:
            mx[col_ones,:]+=mx[i,:]
            mx[mx == 2]=0

    return mx


def encoder(H):
    """Function to encode a parity check matrix, H. This function will find
    the RREF version of H in modulo 2 before splitting the matrix between
    the left most columns containing the pivots and the right most columns
    containing the non pivots. The right most columns will be defined as P.
    Let M be the number of rows of H, N be the codeword length (number of
    columns) and K the block length (N-M). We then return the reordered
    version of H in RREF as [P,I_M] and the generator matrix G=[I_K,P]^T.

    Parameters
        - H: The parity check matrix

    Return
        - H_hat_ordered: The parity check matrix in the form [P,I_M]
        - G: The generator matrix in the form [I_K,P]^T"""

    # Perform Gaussian elimination modulo 2 to return the RREF
    H_hat=gaus_elim_mod2(mx=H)

    # Find the N, K values
    M,N=H_hat.shape
    K=N-M

    # Find P, H=[I_{N-K},P] and G=[P,I_K]^T
    P=H[:,M:]
    H_hat_ordered=np.concatenate([P,np.identity(M)], axis=1)
    G=np.concatenate([np.identity(K), P], axis=0)

    return H_hat_ordered, G
```

```
if __name__ == "__main__":

    H = np.array([[1,1,1,1,0,0],
                  [0,0,1,1,0,1],
                  [1,0,0,1,1,0]])

    # Q1
    H_hat, G=encoder(H=H.copy())

    print('H_hat')
    print(H_hat)

    print('G')
    print(G)
```

Questions 3 and 4

```
# Graphical Models Assignment - Task 2 Questions 3 and 4 - Decoder

import numpy as np


def message_node_2_check_node(Mv, Mvc, Mcv):
    """Update the marginal probabilities of the messages from the message
    nodes to the check nodes. This is calculated as described by
    Shokrallahi. That is, we calculate
        Mvc=Mv+sum(Mc'v) where c' in Cv\{c}

    Parameters
        - Mv: The vector containing the lls of the message nodes over the
              received word
        - Mvc: The original message nodes to check nodes matrix that we
               are updating
        - Mcv: The check nodes to message nodes matrix

    return
        - Mvc: The updated message nodes to check nodes matrix"""

    # Update the Mvc matrix by column
```

```
    for i in range(Mvc.shape[1]):

        # For each column, find the connections
        connections=np.where(Mvc[:,i] != 0)[0]

        # Calculate the sum of the Mcv for the connections, not including
        # self
        Mcv_sum=np.repeat(np.sum(Mcv[connections,i]),len(connections))
        Mcv_sum=Mcv_sum-Mcv[connections,i]

        # Update given column of Mvc
        Mvc[connections,i]=Mv[connections,i]+Mcv_sum

    return Mvc



def check_node_2_message_node(Mvc):
    """Pass the message from the check nodes to the message nodes as
    described by Shokrallahi. That is, we calculate
        Mcv = ln((1+prod(tanh(Mv'c/2)))/(1-prod(tanh(Mv'c/2))))
        where v' in Vc\{v}.

    Parameters
        - Mvc: The message passed from the message nodes to the check
               nodes

    return
        - Mcv: The message passed from the check nodes to the message
               nodes"""

    # Create the check nodes to variable nodes matrix by row
    Mcv=np.zeros(Mvc.shape)
    for i in range(len(Mvc)):

        # For each row, find the connections (non zero elements)
        connections=np.where(Mvc[i,:] != 0)[0]

        # Calculate the tanh product of all connections, except self
        tanh_msg=np.tanh(Mvc[i,connections]/2)
        tanh_prod=np.repeat(np.product(tanh_msg),len(tanh_msg))
        tanh_prod=tanh_prod/tanh_msg
```

```
        # Calculate the check nodes to message nodes for each connection
        # and assign to the Mcv matrix
        Mcv_l=np.log((1+tanh_prod)/(1-tanh_prod))
        Mcv[i,connections]=Mcv_l

    return Mcv



def initialise(H, y, p):
    """Calculate the initial log-likelihoods of the message nodes over the
    observed bits of the received word.

    Parameters
        - H: The mxn risk parity matrix
        - y: The coded message to be decoded
        - p: The noise ratio

    return
        - Mv: The initial log-likelihoods of the message nodes over the
              observed bits of the received word."""

    # Calculate the log-likelihoods
    # The lls of the message nodes over observed bits equal to 0 is
    # ln((1-p)/p)
    # The lls of the message nodes over observed bits equal to 1 is
    # ln(p/(1-p))
    log_prob=np.log((1-p)/p)
    prob_y=np.array([log_prob if int(i)==0 else log_prob*(-1) for i in y])
    Mv=np.multiply(H,prob_y)

    return Mv



def decoder(H, y, p, num_iter=20):
    """Function to decode a message using loopy belief propagation for the
    binary symmetric channel. For ease of computation, the probabilities
    are calculated using log-likelihood as described in 'Amin Shokrallahi.
    LDPC codes: An introduction. 2002'. This algorithm works as follows
        1. Calculate the log-likelihoods for the message nodes over the
```

```
        received word
    2. Pass the message from the check nodes to the message nodes
    3. Pass the message from the message nodes to the check nodes
    4. Decode the message
    5. Find the product of the parity check matrix and the decoded
       message.
    6. If the message was correctly decoded (and so Hx=0 where H is
       the parity check matrix and x is the decoded message) then
       return the decoded message.
    7. If the message was not decoded, repeat steps 2-6 until either
       the message is correctly decoded or the maximum number of
       iterations are reached.

Parameters
    - H: The mxn risk parity matrix
    - y: The coded message to be decoded
    - p: The noise ratio
    - num_iter: A single, positive integer giving the maximum number
                of iterations the algorithm is to be run over. The
                default is 20.

Return
    - decoded_vector: The decoded message
    - decode_success: A single integer that is 0 if the message was
                      successfully decoded and -1 otherwise.
    - i: Number of iterations the algorithm was run over"""

# Initialise the lls of the message nodes over the bits of the
# recieved word. Use these lls as well as the initial Muc
Mv=initialise(H=H, y=y, p=p)
Mvc=Mv.copy()

# Run the loopy belief propagation decoding algorithm over multiple
# iterations
# Stop when either the code has been successfully decoded or the maximum
# number of iterations have been reached
decode_success=-1
i=0
while i < num_iter and decode_success == -1:

    i=i+1
    # Compute/update the messages from the check nodes to the message
```

```
        # nodes
        Mcv=check_node_2_message_node(Mvc=Mvc)

        # Update the messages from the message nodes to the check nodes
        Mvc=message_node_2_check_node(Mv=Mv, Mvc=Mvc, Mcv=Mcv)

        # Find the decoded message
        # Sum the messages along the check nodes for the messages from
        # the message nodes to check nodes to form the messages for each
        # bit.
        # If the message is greater than 0, set the bit to 0. If the
        # message is less than or equal to 0, set the bit to 1.
        Mvc_col_sum=Mvc.sum(axis=0)
        decoded_vector=np.zeros(shape=(Mvc_col_sum.shape))
        decoded_vector[np.where(Mvc_col_sum <= 0)[0]]=1

        # Calculate Hx where H is the parity check matrix and x is the
        # decoded message
        pred_conv=np.matmul(H,decoded_vector)
        pred_conv=pred_conv%2

        # If the message has been sucessfully decoded, then Hx=0
        if sum(pred_conv) == 0:
            decode_success=0

        print('iteration: '+str(i)+' Decoded Success: '+str(decode_success))

    return decoded_vector, decode_success, i




def translate_code(decoded_vector, first_bits, ascii_seq):
    """Given a decoded message vector, convert the numerical string into
    English. We first convert the decoded vector into a character string
    of integers. Next take the relevant, first bits of the code. Next
    separate the code by the ASCII symbols sequence. Finally, join up the
    separate sequences and convert for each character.

    Parameters
```

```
        - decoded_vector: The decoded vector
        - first_bits: The bits of the decoded vector relevant to the
                      message
        - ascii_seq: The ASCII sequence

    return
        - final_msg: The final message, converted into English"""

    # Convert the decoded vector into separate character strings of
    # integers
    decoded_vector=decoded_vector.astype('int64')
    decoded_vector=decoded_vector.astype(str)

    # Keep the relevant bits and split into sequences
    orig_msg=decoded_vector[:first_bits]
    orig_msg_split=np.array_split(orig_msg,ascii_seq)

    # Join the individual sequences and convert into English, before
    # joining up the individual characters
    orig_msg_split=[''.join(i) for i in orig_msg_split]
    orig_msg_split=[chr(int(i,2)) for i in orig_msg_split]
    final_msg=''.join(orig_msg_split)

    return final_msg



if __name__ == "__main__":

    # Q3
    H1=np.loadtxt('./H1.txt')
    y1=np.loadtxt('./y1.txt')
    decoded_vector, decode_success, iters = decoder(H=H1, y=y1, p=0.1)

    # Q4
    decoded_message=translate_code(decoded_vector=decoded_vector,
                                   first_bits=248, ascii_seq=31)
    print(decoded_message)
```

C Part 3: Code

Question 1

```
import numpy as np
import itertools
import matplotlib.pyplot as plt
import pandas as pd

#Set Parameter
N = 10
betas = [0.01,1,4]

#Unary States Matrix
state_mat = np.array([np.array(i) for i in itertools.product([0, 1], repeat = N)])

#Unary Indicator Matrix
state_mat_ind = np.zeros((2**N,N-1))
for i in range(state_mat_ind.shape[1]):
  state_mat_ind[:,i] = 1*(state_mat[:,i]==state_mat[:,i+1])

#Message Passing Matrix
phi_mat = np.zeros((2**N,N))

#Matrix to store the results
results_mat = np.zeros((len(betas),4))

#Create factor from top (now without exponential and summing)
log_factor_init = state_mat_ind.sum(axis=1)

#Transition Matrix
trans_mat_init = np.zeros((2**N,2**N))
for row in range(state_mat.shape[0]):
  for col in range(state_mat.shape[0]):
    trans_mat_init[row,col] = np.exp(sum(state_mat[row]==state_mat[col]))

for idx,beta in enumerate(betas):
  #Include beta in the factor and transition matrix
  log_factor = log_factor_init * beta
  trans_mat = trans_mat_init ** beta

  #Message passing loop:
```

```
#Start with initial factor previous
log_message = log_factor.copy()
phi_mat[:,0] = log_message
for i in range(1,N):
  log_message_star = log_message.max()
  log_message = log_message_star + np.log(trans_mat@np.exp(log_message-log_message_sta
  phi_mat[:,i] = log_message

#transform the log message back into exponential but subtract to max to avoid overfl
log_message_star = log_message.max()
log_message_transformed = (np.exp(log_message-log_message_star))


#Select states that represent the relevant case and sum up their potential

#State x_1,10 = 1 & x_10,10 = 1
idx_11 = (state_mat[:,0]==1) & (state_mat[:,-1]==1)
log_message_transformed_11 = log_message_transformed[idx_11].sum()

#State x_1,10 = 1 & x_10,10 = 0
idx_10 = (state_mat[:,0]==1) & (state_mat[:,-1]==0)
log_message_transformed_10 = log_message_transformed[idx_10].sum()

#State x_1,10 = 0 & x_10,10 = 1
idx_01 = (state_mat[:,0]==0) & (state_mat[:,-1]==1)
log_message_transformed_01 = log_message_transformed[idx_01].sum()

#State x_1,10 = 0 & x_10,10 = 0
idx_00 = (state_mat[:,0]==0) & (state_mat[:,-1]==0)
log_message_transformed_00 = log_message_transformed[idx_00].sum()

#store sum of relevant potentials in a list
log_message_transformed_list = np.array([log_message_transformed_11,log_message_transf

#normalise the 4 probabilities
z = log_message_transformed_list.sum()

#calculate all probabilities
log_message_transformed_list_norm = log_message_transformed_list / z
log_message_transformed_list_norm.round(4)

#Store results
```

```
  results_mat[idx,:] = log_message_transformed_list_norm.round(2)

#Convert results to a PD Dataframe
cols = ["11","10","01","00"]
res_df = pd.DataFrame(data=results_mat,index=betas,columns=cols)
res_df
```

Question 2

```
# mean field approximation of the Ising model using coordinate ascent
import os

import numpy as np
import numba as nb
import pandas as pd

import matplotlib.pyplot as plt

# from gm.part3.validation import get_joint_prob_of_two_nodes
# from gm.part3.message_passing import message_passing


def sum_neighbors(i, j, q):
    """
    Sum the neighbours: 2 * q_k - 1

    Parameters
    ----------
    i: int
        'row' index of node, used to determine neighbours on grid
    j: int
        'col' index of node, used to determine neighbours on grid
    q: float np.array
        probability array, each entry should be in [0,1], gives the prob.
        of node being in +1 state
        shape (n,n)

    Returns
    -------
    float: sum of (2 * q - 1) from neighboring node
```

```
    """

    nrow = q.shape[0]
    ncol = q.shape[1]
    out = 0

    if (i - 1) >= 0:
        out += (2 * q[(i-1), j] - 1)
    if (i + 1) <= (nrow - 1):
        out += (2 * q[(i + 1), j] - 1)
    if (j - 1) >= 0:
        out += (2 * q[i, (j-1)] - 1)
    if (j + 1) <= (ncol - 1):
        out += (2 * q[i, (j+1)] - 1)

    return out


@nb.jit(nopython=True)
def binary_entropy(q):
    """
    Binary Entropy

    Parameters
    ----------
    q: float
        probability of getting positive state (for binomial random variable)

    Returns
    -------
    float: binary entropy:  q * log(1/q) + (1-q) * log(1/(1-q))
    """
    # q \in [0,1]
    # to avoid division by zero
    if q == 0.0:
        return 0
    # to avoid division by zero
    elif q == 1.0:
        return 0
    else:
        return q * np.log(1 / q) + (1-q) * np.log(1 / (1 - q))
```

```
@nb.jit(nopython=True)
def neighbour_expected_energy(i, j, q):
    """
    Expected energy of neighbour of node in nxn grid

    Parameters
    ----------
    i: int
        'row' index of node, used to determine neighbours on grid
    j: int
        'col' index of node, used to determine neighbours on grid
    q: float np.array
        probability array, each entry should be in [0,1], gives the prob. of node
        being in +1 state.
        shape (n,n)

    Returns
    -------
    float: sum of expected energy for a single node and it's neighbours

    """
    # given a node in the grid (i,g) and a q array
    # sum the expected energy for given node
    out = 0
    qi = q[i, j]
    nrow, ncol = q.shape[0], q.shape[1]

    if (i - 1) >= 0:
        qj = q[(i-1), j]
        # out += qi*qj + (1-qi) * (1-qj)
        out += (2 * qi * qj - qi - qj + 1)
    if (i + 1) <= (nrow - 1):
        qj = q[(i + 1), j]
        # out += qi*qj + (1-qi) * (1-qj)
        out += (2 * qi * qj - qi - qj + 1)
    if (j - 1) >= 0:
        qj = q[i, (j-1)]
        # out += qi*qj + (1-qi) * (1-qj)
        out += (2 * qi * qj - qi - qj + 1)
    if (j + 1) <= (ncol - 1):
        qj = q[i, (j+1)]
```

```
        # out += qi*qj + (1-qi) * (1-qj)
        out += (2 * qi * qj - qi - qj + 1)

    return out


@nb.jit(nopython=True)
def expected_energy(q):
    """
    Expected energy for nxn grid
    <E(x)>_Q

    Parameters
    ----------
    q: float np.array
        probability array, each entry should be in [0,1], gives the prob. of
        node being in +1 state
        shape (n,n)

    Returns
    -------
    float: sum of expected energy, E(x), of grid under probability distribution Q

    """
    nrow, ncol = q.shape[0], q.shape[1]

    tot = 0
    for i in range(nrow):
        for j in range(ncol):
            tot += neighbour_expected_energy(i, j, q)

    return -0.5 * tot


@nb.jit(nopython=True)
def sq_entropy(q):
    """
    Entrop of the grid: sum of binary entropy for each node
    S_Q = \sum_{n} H(q_n)

    Parameters
    ----------
```

```
    q: float np.array
        probability array, each entry should be in [0,1], gives the prob. of
        node being in +1 state
        shape (n,n)


    Returns
    -------
    float: entropy of grid using distribution Q

    """
    nrow, ncol = q.shape[0], q.shape[1]
    tot = 0
    for i in range(nrow):
        for j in range(ncol):
            tot += binary_entropy(q[i, j])

    return tot


@nb.jit(nopython=True)
def vfree_energy(q, b):
    """
    variational free energy (vfe) of grid
    vfe is upper bound of estimate of true free energy

    Parameters
    ----------
    q: float np.array
        probability array, each entry should be in [0,1], gives the prob. of node
        being in +1 state
        shape (n,n)
    b: float
        beta parameter, measure of coolness (bigger means cooler), should be > 0

    Returns
    -------

    """
    # beta \tilde{F}(\theta) = \beta <E(x)>_Q - S_Q
    return b * expected_energy(q) - sq_entropy(q)
```

```
def mean_field(N, n, b, node1=None, node2=None, max_iter=1000, tol=1e-10,
               verbose=False, track_state=False):
    """
    find Mean Field Approximation by coordinate descent (to minimising
    variational free energy) to estimate joint probabilities of
    two nodes on an n x n grid

    Parameters
    ----------
    N: int
        number of mean field approximations to make
    n: int
        specifies (n x n) grid size
    b: float
        beta (coolness) parameter, must be >= 0
    node1: tuple or None, default None
        first node location in grid
        Each value in tuple must be in {0, ..., n-1}
        if None defaults (0, n-1)
    node2: tuple or None, default None
        second node location in grid
        Each value in tuple must be in {0, ..., n-1}
        if None defaults (n-1, n-1)
    max_iter: int, default 1000
        maximum number of iterations of coordinate descent when
        finding mean field approx.
    tol: float, default 1e-10
        stop search for approximating distribution when the largest absolute
        change in approximating distribution, q, from one iteration to
        another is less that or equal to tol
    verbose: bool, default False
        print steps along the way
    track_state: bool, default False
        if True will provide more detailed output, see Returns

    Returns
    -------
    if track_state is False
        array with shape (2,2) of the joint probability of node1 and node2
        being in states: [ [P(-1,-1), P(-1,1)], [P(1,-1), P(1,1)] ]
        joint probability taken as average joint probabilities generated
```

```
        from each of the N mean field approximations, q, made
    otherwise track_state is True
        four objects:
        prob: np.array, shape (2,2) same as track_state = False
        prob_states: np.array, shape (N,2,2), joint probability of from
            each of the N q's generated
        q_states: np.array, shape (N, n, n), each mean field
            approximation generated
        vfe_state: np.array, shape(N,), variational free energy of each
            mean field approximation generated

    """
    if node1 is None:
        node1 = (0, n-1)
    if node2 is None:
        node2 = (n-1, n-1)

    # array to store the join prob
    prob = np.zeros((2, 2))

    # track the final states, and the variation free energy?
    # - will come at a memory cost
    if track_state:
        prob_states = np.zeros((N, 2, 2))
        q_states = np.zeros((N, n, n))
        vfe_state = np.zeros(N)

    # approximate p(x) with
    # q(x) = prod_{n=1}^{N} (q_n(x; a_n))
    # where q_n(x) = e^(a_n * x_n) / (e^{a_n} + e^{-a_n})

    # generate many final states
    for _ in range(N):

        # start with random q
        q = np.random.uniform(0, 1, (n, n))

        # get the corresponding a values (not needed
        a = (-1/2) * np.log(1/q - 1)

        # coordinate ascent with update
        # a_m = (b/2) sum_{j \in ne(m)} (2*q_j - 1)
```

```
# apply coordinate ascent
for iter in range(max_iter):
    vfe = vfree_energy(q, b)
    q_prev = q.copy()
    update_order = np.random.choice(np.arange(n*n), n*n, replace=False)
    for u in update_order:
        # get the i,j locations
        i = u//n
        j = u - (i * n)
        # print(i, j)

        # update 'a_m'
        a[i, j] = (b / 2) * sum_neighbors(i, j, q)

        # update q - probability of being in spin state 1
        # q = np.exp(a) / (np.exp(a) + np.exp(-a))
        q[i, j] = np.exp(a[i, j]) / (np.exp(a[i, j]) + np.exp(-a[i, j]))

    if verbose > 1:
        print(f"change in v. free energy: {vfree_energy(q, b) - vfe}")

    max_diff = np.abs(q - q_prev).max()
    if max_diff <= tol:
        if verbose:
            print(f"converged after: {iter}")
        break

# recall q is the prob of a given node being 1
# can use this to build the conditional probability
# prob node1 is 1
n1_1 = q[node1[0], node1[1]]
n1_0 = (1 - n1_1)
# prob node2 is 1
n2_1 = q[node2[0], node2[1]]
# n2_1 = q[node1[0], node1[1]]
n2_0 = (1 - n2_1)

if verbose:
    print(f"n1_1: {n1_1:.2f}, n2_1: {n2_1:.2f}")

# in the model of q used each x is independent so
```

```
        # when getting the join prob p(x_1, x_2) = p(x_1) p(x_2)
        # adding to previous value, will then average later
        # p_tmp = np.zeros((2,2))

        prob[0, 0] += n1_0 * n2_0
        prob[0, 1] += n1_0 * n2_1
        prob[1, 0] += n1_1 * n2_0
        prob[1, 1] += n1_1 * n2_1

        # track the states?
        if track_state:
            q_states[_, ...] = q
            # ... fairly duplicated code here
            prob_states[_, 0, 0] += n1_0 * n2_0
            prob_states[_, 0, 1] += n1_0 * n2_1
            prob_states[_, 1, 0] += n1_1 * n2_0
            prob_states[_, 1, 1] += n1_1 * n2_1
            vfe_state[_] = vfree_energy(q, b)

    # convert the cumulative joint probability to an average
    prob /= N

    if not track_state:
        return prob
    else:
        return prob, prob_states, q_states, vfe_state


def plot_selected_q_states(select_states,
                           plot_prefix,
                           q_states,
                           b,
                           plot_title=""):
    """
    Plot selected states of q_states array
    - generates two plots, one with lower average q values, other
        with higher average q values
    - plots are written to file using get_tex_path('plots', *args),
        i.e. write plots to <path_to_this_file>/tex/plots
    - selected states is used to take a subset of q_states, these are
        further split into low q and high q
    - of each of those a random entry is selected for plotting
```

```
    Parameters
    ----------
    select_states: bool np.array
        used for selecting states from q_state to plot, shape (N,)
    plot_prefix: str
        prefix, used as prefix of plot file name (after 'vfe_')
    q_states: float np.array
        mean field approximations, shape (N,n,n)
        can be generated from mean_field(..., track_states=True)
    b: float
        beta value used to generate the 'q_states'
    plot_title: str, default ""
        used to specify main plot title / information about plot
        a subtitle with b (beta), variational free energy (VFE) and average q

    Returns
    -------
    None

    """
    # of the lowest vfe - get two examples one with q's very high, one with low
    q_select = q_states[select_states]
    # take the average q value across the grid (last two dimensions)
    ave_grid_q = q_select.mean(axis=(1, 2))

    # NOTE: no longer getting the highest or lowest
    # just getting less than average or more than average
    # grid with (lowest) q values - select random if more than 1
    lowest_q = np.where(ave_grid_q <= ave_grid_q.mean())[0]
    lowest_q = np.random.choice(lowest_q)
    # grid with highest q values - select random if more than 1
    highest_q = np.where(ave_grid_q >= ave_grid_q.mean())[0]
    highest_q = np.random.choice(highest_q)

    for k, v in {"Low Average Q": lowest_q, "High Average Q": highest_q}.items():
        plt.close()
        # select the grid
        tmp_Q = q_select[v, ...]
        # get the average q
        ave_q = tmp_Q.mean()
        # variational free energy
```

```
        tmp_vfe = vfree_energy(tmp_Q, b)
        # heatmap = plt.imshow(q_states[max_vfe, ...], vmin=0, vmax=1.0)
        heatmap = plt.imshow(tmp_Q, vmin=0, vmax=1.0)
        plt_title = f"{plot_title} \n $\\beta=${b}, " \
                    f"VFE={tmp_vfe:.2f}, " \
                    f"Ave. q = {ave_q:.2f}"
        plt.title(plt_title)
        plt.colorbar(heatmap)
        plot_file = get_tex_path("plots", f"vfe_{plot_prefix}_{k}_n{n}_b{b}.png")
        print("-" * 10)
        print(f"writing q state plot to:\n{plot_file}")

        plt.savefig(plot_file)


def plot_results(vfe_state, q_states, b):
    """
    Given the Variational Free Energy and the Q states
    - plot the (sorted)
    - plot two cases Q in the lowest vfe
    - if there are cases with Q not at (near) global min, plot those

    - plots are written to file using get_tex_path('plots', *args),
        i.e. write plots to <path_to_this_file>/tex/plots


    Parameters
    ----------
    vfe_state: float np.array
        variational free energy of each mean field approximation, shape (N,)
        can be generated from mean_field(..., track_states=True)
    q_states: float np.array
        mean field approximations, shape (N,n,n)
        can be generated from mean_field(..., track_states=True)

    Returns
    -------
    None

    Notes
    -----
    see plot_selected_q_states for more details
```

```
    """

    # --
    # variational free energy of each Q (approximated distribution)
    # --

    # - for high b (b=4.0) vfe takes on two states

    plt.close()
    plt.plot(np.sort(vfe_state))
    plt.title(f"Variational Free Energy of Final Q (sorted)\n $\\beta=${b}")
    plot_file = get_tex_path("plots", f"V_free_energy_n{n}_b{b}.png")
    print("-" * 10)
    print(f"writing final variational free energy states to:\n{plot_file}")
    plt.savefig(plot_file)
    # plt.show()

    # identify the states with lowest vfe
    # NOTE: vfe assumed to be negative
    # taking all values slightly above min and below as being in lowest start
    # this is not robust!
    lowest_vfe = vfe_state <= (vfe_state.min() * 0.99)

    print("plotting examples of Q's with lowest VFE")
    plot_selected_q_states(select_states=lowest_vfe,
                           plot_prefix="MinState",
                           q_states=q_states,
                           b=b,
                           plot_title="Q with Lowest VFE")

    # if not all states had global min vfe - plot some of those too
    if not lowest_vfe.all():
        print("not all final states had the same lowest variational "
              "free energy, plotting example")

        plot_selected_q_states(select_states=~lowest_vfe,
                               plot_prefix="NonMinState",
                               q_states=q_states,
                               b=b,
                               plot_title="Q with VFE Greater than Global Minimum")
```

```
def get_tex_path(*args,  create_dir_if_not_exist=True):
    """
    get path to 'tex' directory, contained in same directory as file
    if create_dir_if_not_exist=True will make directories """

    try:
        if __file__ == "":
            file_dir = os.getcwd()
        else:
            file_dir = os.path.dirname(__file__)
    except Exception as e:
        print("issue getting directory of file from __file__")
        file_dir = os.getcwd()

    out = os.path.join(file_dir, 'tex', *args)
    if create_dir_if_not_exist:
        os.makedirs(os.path.dirname(out), exist_ok=True)

    return out


print("-" * 200)
print("Mean Field Approximation of (Modified) Ising Model")
print("-" * 20)

# from gm.part3 import get_tex_path

# ---
# parameters
# ---

# grid size
n = 10
# number of mean fields to produce
N = 1000
# max number of iterations
max_iter = 10000
# 'coolness' parameter
betas = [0.01, 1.0, 4.0]
# betas = [1.0]
```

```
# node location
node1 = (0, n-1)
node2 = (n-1, n-1)

print(f"grid size: {n} x {n}")
print(f"getting conditional prob for nodes: {node1} , {node2}")
print(f"number of approximate distributions to produce: {N}")

for b in betas:
    print("*" * 50)
    print(f"running for beta: {b}")

    # ---
    # apply mean field - tracking the states and metrics
    # ---

    print("getting mean field approx using coordinate descent")
    prob, prob_states, q_states, vfe_state = mean_field(N=N,
                                                        n=n,
                                                        b=b,
                                                        node1=node1,
                                                        node2=node2,
                                                        max_iter=max_iter,
                                                        verbose=False,
                                                        track_state=True)

    # ---------
    # analysis
    # ---------

    # ---
    # generate plots
    # ---

    plot_results(vfe_state, q_states, b)

    # ---
    # print tables and write to file
    # ---

    print("-" * 10)
    print("selecting all Q's")
```

```
    pdf = pd.DataFrame(prob, index=[-1, 1], columns=[-1, 1])
    print("conditional prob using all Q's "
          "(mean field approximations) found ")
    print(pdf)
    table_file = get_tex_path("tables",
                              f"conditional_prob_global_ave_n{n}_b{b}.tex")
    with open(table_file, "w") as f:
        f.write(pdf.to_latex(float_format="%.3g", index=True))

    dif = vfe_state.max() - vfe_state.min()
    print(f"difference between min and max "
          f"variational free energy was: {dif:.3g}")
    if dif > 1e-6:
        # select only those in min state
        # - this is not very robust!
        in_min_state = vfe_state <= (vfe_state.min() + 0.1 * dif)

        print(f"number of points in (effectively) min vfe state: "
              f"{in_min_state.sum()} out of {N}")
        min_state_prob = prob_states[in_min_state, ...].mean(axis=0)

        print("selecting just from the states with min VFE")
        pdf = pd.DataFrame(min_state_prob, index=[-1, 1], columns=[-1, 1])
        print("conditional prob using mean field "
              "(selecting those with global min vfe)")
        print(pdf)

        table_file = get_tex_path("tables",
                                  f"conditional_prob_min_only_n{n}_b{b}.tex")
        with open(table_file, "w") as f:
            f.write(pdf.to_latex(float_format="%.3g", index=True))
```

Question 3

```
import numpy as np
import itertools
import matplotlib.pyplot as plt
import pandas as pd

#Function to identify neighbours
#Def here
```

```
def neighbour_indices(curr_node,N=10):
  i,j = curr_node
  neighbours_indices_list = []
  #check down
  if i+1 <=N-1:
    neighbours_indices_list.append((i+1,j))
  #check up
  if i-1 >=0:
    neighbours_indices_list.append((i-1,j))
  #check right
  if j+1 <=N-1:
    neighbours_indices_list.append((i,j+1))
  #check left
  if j-1 >=0:
    neighbours_indices_list.append((i,j-1))
  return neighbours_indices_list

#Function that returns the probability of the current node being in in state 0 or stat
def cond_prob(curr_grid,neighbours_indices_list,beta):
  non_norm_prob_list = np.zeros((2))
  non_norm_prob_list[0] = 1
  non_norm_prob_list[1] = 1
  for neighbour in neighbours_indices_list:
    non_norm_prob_list[0] *= np.exp(beta * (0 == curr_grid[neighbour]))
    non_norm_prob_list[1] *= np.exp(beta * (1 == curr_grid[neighbour]))

  Z = non_norm_prob_list.sum()
  norm_prob_list = non_norm_prob_list / Z

  return norm_prob_list

  #Version using the "True" definition of Gibbs sampling
#-> Create one long chain of samples and then use that to create a probability distrib


#Loop implementation to run all params

N = 10
betas = [0.01,1,4]

num_chains = 100
num_iters = 10000
```

```
burn_period = 1000
print_gird = False


#List of States to draw from
state_mat = np.array([np.array(i) for i in itertools.product([0, 1], repeat = N)])

#list of touple indices
idx_grid_list = []
for col in range(N):
  for row in range(N):
    idx_grid_list.append((row,col))

#Matrix to store the results
results_mat_gibbs = np.zeros((len(betas),4))

#Matrix to store the result for each chain
results_cube_gibbs = np.zeros((len(betas),4,num_chains))

#Gibs Sampling Loop 1
for curr_chain in range(num_chains):
  #if curr_chain%100 == 0:
  print("Chain: {}".format(curr_chain))

  #Loop through all beta params
  for idx, beta in enumerate(betas):
    print("Running for beta: {}".format(beta))
    sample_states = np.zeros((num_iters,N**2))

    #Randomly initialise grid
    curr_grid = np.random.randint(0, 2, (N,N))
    #Plot initial grid
    if print_gird:
      print("Random Grid")
      plt.imshow(curr_grid, vmin = 0, vmax = 1)
      plt.show()

    #Repeat update of all nodes num_iters times
    for curr_iter in range(num_iters):

      #Shuffle Order of update
      np.random.shuffle(idx_grid_list)
```

```
    #Go through each variable sequentially - One Gibbs Sweep
    for curr_node in idx_grid_list:

      #Identify the neighbours of the chosen point
      neighbours_indices_list = neighbour_indices(curr_node,N)

      #Calculate the potential for each state and normalise to get conditional distr
      cond_dist = cond_prob(curr_grid,neighbours_indices_list,beta=beta)

      #Draw random sample for the chosen variable and update the grid
      curr_grid[curr_node] = 1 if np.random.rand()<cond_dist[1] else 0

    #Following Gibbs Sweep, add sample to the markov chain
    sample_states[curr_iter,:] = curr_grid.reshape(-1)

  #Plot after updating each variable num_iters time -> Last sample in the Markov Cha
  if print_gird:
    print("Last Sample in Chain (after {} Gibbs Sweeps".format(num_iters))
    plt.imshow(curr_grid, vmin = 0, vmax = 1)
    plt.show()

  #Calculate probability from saples

  #Select states that represent the relevant case and sum up their potential

  #State x_1,10 = 1 & x_10,10 = 1
  idx_11 = (sample_states[burn_period:,9]==1) & (sample_states[burn_period:,-1]==1)
  sample_states_11 = idx_11.sum()

  #State x_1,10 = 1 & x_10,10 = 0
  idx_10 = (sample_states[burn_period:,9]==1) & (sample_states[burn_period:,-1]==0)
  sample_states_10 = idx_10.sum()

  #State x_1,10 = 0 & x_10,10 = 1
  idx_01 = (sample_states[burn_period:,9]==0) & (sample_states[burn_period:,-1]==1)
  sample_states_01 = idx_01.sum()

  #State x_1,10 = 0 & x_10,10 = 0
  idx_00 = (sample_states[burn_period:,9]==0) & (sample_states[burn_period:,-1]==0)
  sample_states_00 = idx_00.sum()
```

```
    #store sum of relevant potentials in a list
    sample_states_list = np.array([sample_states_11,sample_states_10,sample_states_01,sa

    #normalise the 4 probabilities
    z = sample_states_list.sum()

    #calculate all probabilities
    sample_states_list_norm = sample_states_list / z

    #Store results
    results_mat_gibbs[idx,:] = sample_states_list_norm

  #Add results of each chain to a cube to store several chains for each beta
  results_cube_gibbs[:,:,curr_chain] = results_mat_gibbs

#Convert results to a PD  for last used chains
cols = ["11","10","01","00"]
res_df_gibbs_last = pd.DataFrame(data=results_mat_gibbs,index=betas,columns=cols)

#Convert results to a PD  for average of all chains
results_mat_gibbs_avg = results_cube_gibbs.mean(axis=2)
cols = ["11","10","01","00"]
res_df_gibbs_avg = pd.DataFrame(data=results_mat_gibbs_avg,index=betas,columns=cols)

#Convert results to a PD  for last used chains
results_mat_gibbs_std = results_cube_gibbs.std(axis=2)
cols = ["11","10","01","00"]
res_df_gibbs_std = pd.DataFrame(data=results_mat_gibbs_std,index=betas,columns=cols)

print("Prediction from last chain")
print(pd.DataFrame(data=res_df_gibbs_last.round(4),index=betas,columns=cols))

print("Average prediction of 100 chains")
print(pd.DataFrame(data=res_df_gibbs_avg.round(4),index=betas,columns=cols))

print("Std. of the predictions of 100 chains")
print(pd.DataFrame(data=res_df_gibbs_std.round(4),index=betas,columns=cols))


#Experiment - Start with a 50/50 grid and observe convergence

N = 10
```

```
betas = [4]

num_samples = 1
num_iters = 10000
print_gird = True

#List of States to draw from
state_mat = np.array([np.array(i) for i in itertools.product([0, 1], repeat = N)])

#list of touple indices
idx_grid_list = []
for col in range(N):
  for row in range(N):
    idx_grid_list.append((row,col))

#Matrix to store the results
results_mat_gibbs = np.zeros((len(betas),4))

#Loop through all beta params
for idx, beta in enumerate(betas):
  print("Running for beta: {}".format(beta))
  sample_states = np.zeros((num_samples,N**2))

  #Gibs Sampling Loop 1
  for curr_sample in range(num_samples):
    if curr_sample%100 == 0:
      print("Sample: {}".format(curr_sample))

    #Randomly initialise grid
    curr_grid = np.random.randint(0, 2, (N,N))
    curr_grid[:,:5] = 0
    curr_grid[:,5:] = 1


    #Plot initial grid
    if print_gird:
      print("Split Grid")
      #print("Random Grid")
      plt.imshow(curr_grid, vmin = 0, vmax = 1)
      plt.savefig("experiment_init.png")
      plt.show()
```

```
    #Repeat update of all nodes num_iters times
    for curr_iter in range(num_iters):

      #Shuffle Order of update
      np.random.shuffle(idx_grid_list)

      #Go through each variable sequentially
      for curr_node in idx_grid_list:

        #Identify the neighbours of the chosen point
        neighbours_indices_list = neighbour_indices(curr_node,N)

        #Calculate the potential for each state and normalise to get conditional distr
        cond_dist = cond_prob(curr_grid,neighbours_indices_list,beta=beta)

        #Draw random sample for the chosen variable and update the grid
        curr_grid[curr_node] = 1 if np.random.rand()<cond_dist[1] else 0

        #Plot after updating each variable num_iters time
      if print_gird:
        if curr_iter==999 or curr_iter==9999:
          print("Grid after updating each node {} times".format(curr_iter))
          plt.imshow(curr_grid, vmin = 0, vmax = 1)
          plt.savefig("experiment_beta4_{}.png".format(curr_iter+1))
          plt.show()

    sample_states[curr_sample,:] = curr_grid.reshape(-1)

  #Calculate probability from saples

  #Select states that represent the relevant case and sum up their potential

  #State x_1,10 = 1 & x_10,10 = 1
  idx_11 = (sample_states[:,9]==1) & (sample_states[:,-1]==1)
  sample_states_11 = idx_11.sum()

  #State x_1,10 = 1 & x_10,10 = 0
  idx_10 = (sample_states[:,9]==1) & (sample_states[:,-1]==0)
  sample_states_10 = idx_10.sum()

  #State x_1,10 = 0 & x_10,10 = 1
  idx_01 = (sample_states[:,9]==0) & (sample_states[:,-1]==1)
```

```
  sample_states_01 = idx_01.sum()

  #State x_1,10 = 0 & x_10,10 = 0
  idx_00 = (sample_states[:,9]==0) & (sample_states[:,-1]==0)
  sample_states_00 = idx_00.sum()

  #store sum of relevant potentials in a list
  sample_states_list = np.array([sample_states_11,sample_states_10,sample_states_01,samp

  #normalise the 4 probabilities
  z = sample_states_list.sum()

  #calculate all probabilities
  sample_states_list_norm = sample_states_list / z

  #Store results
  results_mat_gibbs[idx,:] = sample_states_list_norm

#Convert results to a PD Dataframe
cols = ["11","10","01","00"]
res_df_gibbs = pd.DataFrame(data=results_mat_gibbs,index=betas,columns=cols)
res_df_gibbs
```