# Probabilistic Modelling and Reasoning: Assignment Part 2/2

*This is Part 2/2 of the assignment with Questions 3 and 4 out of four, containing the remaining 50/100 marks.*

**Deadline: Friday March 29 2019 at 16:00**

---

**Marks:** The assignment is out of 100 marks and forms 20% of your final grade for the course.

**Group work:** You may work alone or in pairs. If the work is done in pairs, both students have to work together on all questions in *both* Part 1 and Part 2.

**Submission instructions:** There are different deadlines for the two parts of the assignment. When completed, submit this part manually to the ITO office by the above deadline. Handwritten submissions are acceptable if the handwriting is *neat and legible*. If we cannot read something we cannot give you marks for it. Do not put your name but only your student ID on the copy.

If the work is done in pairs, for each part only a single report needs to be submitted. Note both student IDs on the copy and *add a statement confirming that both students contributed equally to answering all questions.*

**Academic conduct:** Assessed work is subject to University regulations on academic conduct: http://web.inf.ed.ac.uk/infweb/admin/policies/academic-misconduct
*In particular, do not show your code, answers or write-up to anyone else.*

**Late submissions:** We follow the policy of the School of Informatics: http://web.inf.ed.ac.uk/infweb/student-services/ito/admin/coursework-projects/late-coursework-extension-requests

**Data and code:** Associated files for the assignment are available as the compressed zip archive file `assignment-files.zip` from the course homepage. The zip file contains:

- `q3_poisson.txt`
- `pca.stan`
- `q4_pca.txt`

**Notation:** Unless said otherwise, we follow the notation from part 1. In particular,

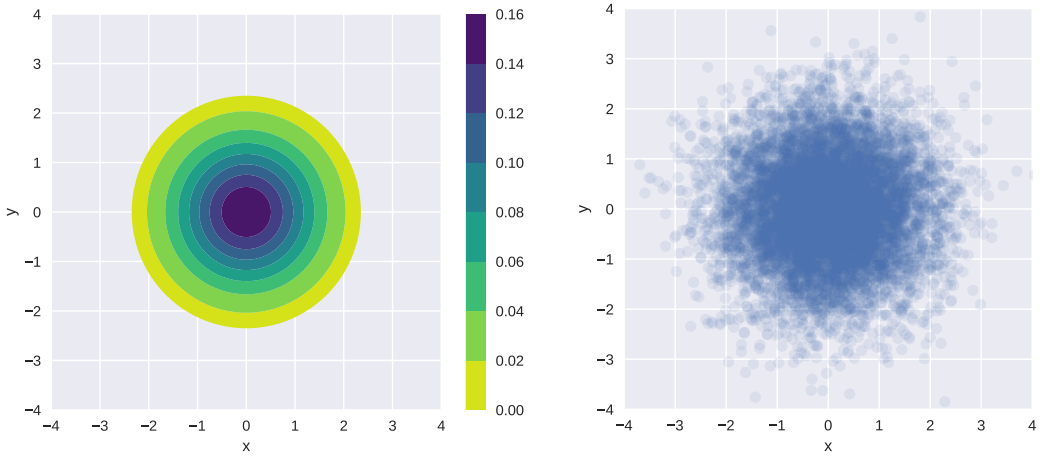
$$\mathcal{N}(x; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \tag{1}$$

denotes the density of a zero mean Gaussian with variance $\sigma^2$.

**Remark:** While the lectures cover background and more specific material related to the question, it is expected that you independently study the references provided in the questions below. Do not wait for all the material to be explained in the lectures—it may not. The questions partly test your ability to extend your knowledge yourself.

# Question 3: Basic Markov chain Monte Carlo inference (25 marks)

The second half of this assignment focuses on sampling and approximate inference by Markov chain Monte Carlo (MCMC). This class of methods can be used to obtain samples from a probability distribution, e.g. a posterior distribution. The samples aproximately represent the distribution, as illustrated in Figure 1, and can be used to approximate expectations. In this question, you will write your own MCMC algorithm, use it to perform inference, and run diagnostics to detect problems.




(a) True density.

(b) Density represented by 10,000 samples.

Figure 1: The joint density of a standard bivariate normal distribution: $p(x, y) = \mathcal{N}(x; 0, 1)\mathcal{N}(y; 0, 1)$.

**Brief introduction to MCMC and the Metropolis-Hastings algorithm** Consider a vector of $d$ random variables $\boldsymbol{\theta} = (\theta_1, \ldots, \theta_d)$ and some observed data $\mathcal{D}$. In many cases, we are interested in computing expectations under the posterior distribution $p(\boldsymbol{\theta} \mid \mathcal{D})$, e.g.

$$\mathbb{E}_{p(\boldsymbol{\theta}|\mathcal{D})}\left[f(\boldsymbol{\theta})\right] = \int f(\boldsymbol{\theta}) p(\boldsymbol{\theta} \mid \mathcal{D}) \mathrm{d}\boldsymbol{\theta} \tag{2}$$

for some function $f(\boldsymbol{\theta})$. If $d$ is small, e.g. $d \leq 3$, deterministic numerical methods can be used to approximate the integral to high accuracy.[1] But for higher dimensions, these methods are generally not applicable any more. The expectation, however, can be approximate as a sample average if we have samples $\boldsymbol{\theta}^{(i)}$ from $p(\boldsymbol{\theta} \mid \mathcal{D})$:

$$\mathbb{E}_{p(\boldsymbol{\theta}|\mathcal{D})}\left[f(\boldsymbol{\theta})\right] \approx \frac{1}{S} \sum_{i=1}^{S} f(\boldsymbol{\theta}^{(i)}) \tag{3}$$

In MCMC methods, the samples $\boldsymbol{\theta}^{(1)}, \ldots, \boldsymbol{\theta}^{(S)}$ used in the above approximation are typically not statistically independent.

Metropolis-Hastings is an MCMC algorithm that generates samples from a distribution $p(\boldsymbol{\theta})$, where $p(\boldsymbol{\theta})$ can be any distribution (prior or posterior) on the parameters. The algorithm is iterative and at iteration $t$, it uses:

- a proposal distribution $q(\boldsymbol{\theta}; \boldsymbol{\theta}^{(t)})$, parametrised by the current state of the Markov chain, i.e. $\boldsymbol{\theta}^{(t)}$;
- a function $p^*(\boldsymbol{\theta})$, which is proportional to $p(\boldsymbol{\theta})$. In other words, $p^*(\boldsymbol{\theta})$ is such that $p(\boldsymbol{\theta}) = \frac{p^*(\boldsymbol{\theta})}{\int p^*(\boldsymbol{\theta}) \mathrm{d}\boldsymbol{\theta}}$.[2]

Read Section 27.4 from Barber (2012) and Section 29.4 from MacKay (2003) to familiarise yourself with Metropolis-Hastings.

For all tasks in this Question, we work with a Gaussian proposal distribution $q(\boldsymbol{\theta}; \boldsymbol{\theta}^{(t)})$, whose mean is the previous sample in the Markov chain, and whose variance is $\epsilon^2$. That is at iteration $t$ of our Metropolis-Hastings algorithm, $q(\boldsymbol{\theta}; \boldsymbol{\theta}^{(t-1)}) = \mathcal{N}(\boldsymbol{\theta}; \boldsymbol{\theta}^{(t-1)}, \epsilon^2) = \prod_{k=1}^{d} \mathcal{N}(\theta_k; \theta_k^{(t-1)}, \epsilon^2)$.

[1] See e.g. https://en.wikipedia.org/wiki/Numerical_integration.

[2] $p^*$ is thus unnormalised. We used the notation $\tilde{p}$ in the lecture; $p^*$ is also commonly used.

(a) Write a function mh implementing the Metropolis Hasting algorithm, as e.g. given in Algorithm 27.3 of Barber (2012), using the Gaussian proposal distribution above. The function should take as arguments

- p_star: a function on $\boldsymbol{\theta}$ that is proportional to the density of interest $p^*(\boldsymbol{\theta})$;
- param_init: the initial sample — a value for $\boldsymbol{\theta}$ from where the Markov chain starts;
- num_samples: the number $S$ of samples to generate;
- stepsize: a hyperparameter specifying the *variance* of the Gaussian proposal distribution $q$;

and return $[\boldsymbol{\theta}^{(1)}, \ldots, \boldsymbol{\theta}^{(S)}]$ — a list of $S$ samples from $p(\boldsymbol{\theta}) \propto p^*(\boldsymbol{\theta})$. For example:

```
def mh(p_star, param_init, num_samples=5000, stepsize=1.0):
    # your code here
    return samples
```

When used with the above Gaussian proposal distribution, the Metropolis Hastings algorithm is more specifically called Random Walk Metropolis-Hastings algorithm. [5 MARKS]

*[Expected response: Computer code implementing* mh *as described above. You are free to work with your preferred coding language. Note, however, that we recommend using Python in Question 4.]*

(b) Test your algorithm by sampling $5,000$ samples from $p(x, y) = \mathcal{N}(x; 0, 1)\mathcal{N}(y; 0, 1)$. Initialise at $x = 0, y = 0$ and use $\epsilon = 1$. Generate a scatter plot of the obtained samples. The plot should be similar to Figure 1b. Highlight the first 20 samples only. Do these 20 samples alone adequately approximate the true density?

Sample another $5,000$ points from $p(x, y) = \mathcal{N}(x; 0, 1)\mathcal{N}(y; 0, 1)$ using mh with $\epsilon = 1$, but this time initialise at $x = 7, y = 7$. Generate a scatter plot of the drawn samples and highlight the first 20 samples. If everything went as expected, your plot probably shows a "trail" of samples, starting at $x = 7, y = 7$ and slowly approaching the region of space where most of the probability mass is.

In practice, we don't know where the distribution we wish to sample from has high density, so we typically initialise the Markov Chain somewhat arbitrarily, or at the maximum a-posterior sample if available. The samples obtained in the beginning of the chain are typically discarded, as they are not considered to be representative of the target distribution. This initial period between initialisation and starting to collect samples is called "warm-up", or also "burn-in".

Extended your function mh to include an additional warm-up argument $W$, which specifies the number of MCMC steps taken before starting to collect samples. Your function should still return a list of $S$ samples as in (a). [5 MARKS]

*[Expected response: Two scatter plots of* $5,000$ *samples from* $\mathcal{N}(x; 0, 1)\mathcal{N}(y; 0, 1)$ *each, with the first 20 samples highlighted + a one-sentence description. Code implementing the* mh *function with additional warm-up argument.*

*Note: You may use built-in functions that evaluate densities (such as* scipy.stats.norm.pdf *or equivalent), but the drawn samples must have been generated with your Metropolis Hastings algorithm and not some built-in library.]*

(c) Consider a Bayesian Poisson regression model, where outputs $y_n$ are generated from a Poisson distribution of rate $\exp(\alpha x_n + \beta)$, where the $x_n$ are the inputs (covariates), and $\alpha$ and $\beta$ the parameters of the regression model for which we assume a Gaussian prior:

$$\alpha \sim \mathcal{N}(\alpha; 0, 100) \tag{4}$$
$$\beta \sim \mathcal{N}(\beta; 0, 100) \tag{5}$$
$$y_n \sim \text{Poisson}(y_n; \exp(\alpha x_n + \beta)) \quad \text{for } n = 1, \ldots, N \tag{6}$$

$\text{Poisson}(y; \lambda)$ denotes the probability mass function of a Poisson random variable with rate $\lambda$,

$$\text{Poisson}(y; \lambda) = \frac{\lambda^k}{k!} \exp(-\lambda), \qquad y \geq 0. \tag{7}$$

In the provided archive file, there is a file q3_poisson.txt containing observed data $\mathcal{D} = \{(x_n, y_n)\}_{n=1}^N$ where $N = 5$ and

$$(x_1, \ldots, x_5) = (-0.50519053, -0.17185719, 0.16147614, 0.49480947, 0.81509851) \tag{8}$$
$$(y_1, \ldots, y_5) = (1, 0, 2, 1, 2) \tag{9}$$

We are interested in computing the posterior density of the parameters $(\alpha, \beta)$ given the data $\mathcal{D}$ above. Derive and implement a function $p^*$ of the parameters $\alpha$ and $\beta$ that is proportional to the posterior density $p(\alpha, \beta \mid \mathcal{D})$, and which can thus be used as target density in the Metropolis Hastings algorithm. (If your coding environment provides an implementation of the above Poisson pmf, there is no need to implement the pmf yourself)

Use your Metropolis Hastings algorithm to draw $5,000$ samples from the posterior density $p(\alpha, \beta \mid \mathcal{D})$. Set the hyperparameters of the Metropolis-Hastings algorithm to:

- param_init= $(\alpha_{\text{init}}, \beta_{\text{init}}) = (0, 0)$,
- stepsize 1, and
- number of warm-up steps $W = 1000$.

Plot the drawn samples with x-axis $\alpha$ and y-axis $\beta$ and report the posterior mean of $\alpha$ and $\beta$, as well as their correlation coefficient under the posterior. [5 MARKS]

*[Expected response: A derivation of a function $p^*$, which can be used to draw samples from the posterior distribution. A scatter plot of the $5,000$ samples from the posterior. Numerical values for the posterior means and correlation coefficient.]*

(d) Any MCMC algorithm is an *asymptotically exact* inference algorithm, meaning that if it is run forever, it will converge to the desired probability distribution. In practice, we want to run the algorithm long enough to be able to approximate the posterior adequately. How long is long enough for the chain to converge varies drastically depending on the algorithm, the hyperparameters (e.g. stepsize), and the target posterior distribution. It is impossible to determine exactly whether the chain has run long enough, but there exist various diagnostics that can help us determine if we can "trust" the sample-based approximation to the posterior.

A very quick and common way of assessing convergence of the Markov chain is to visually inspect the *trace plots* for each parameter. A trace plot shows how the drawn samples evolve through time, i.e. they are a time-series of the samples generated by the Markov chain. Figure 2 shows examples of trace plots obtained by running the Metropolis Hastings algorithm for different values of the hyperparameters stepsize and param_init. Ideally, the time series covers the whole domain of the target distribution and it is hard to "see" any structure in it so that predicting values of future samples from the current one is difficult. If so, the samples are likely independent from each other and the chain is said to be well "mixed".

Consider the trace plots in Figure 2: Is the hyperparameter stepsize used in Figure 2b larger or smaller than stepsize used in Figure 2a? Is stepsize used in Figure 2c larger or smaller than stepsize used in Figure 2a?

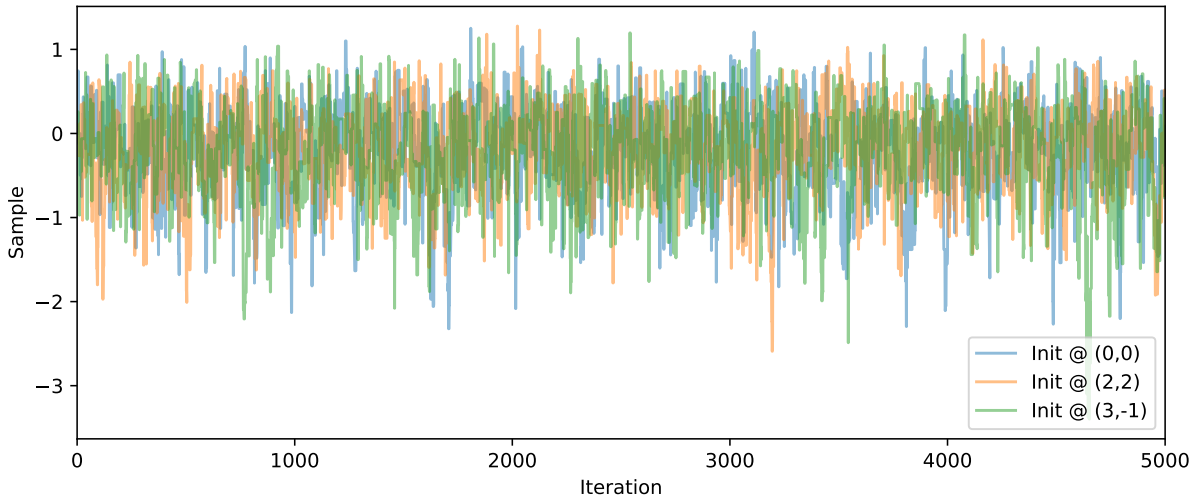
In both cases, explain the behaviour of the trace plots in terms of the workings of the Metropolis Hastings algorithm and the effect of stepsize. [6 MARKS]

*[Expected response: Two smaller/larger statements and two explanations]*

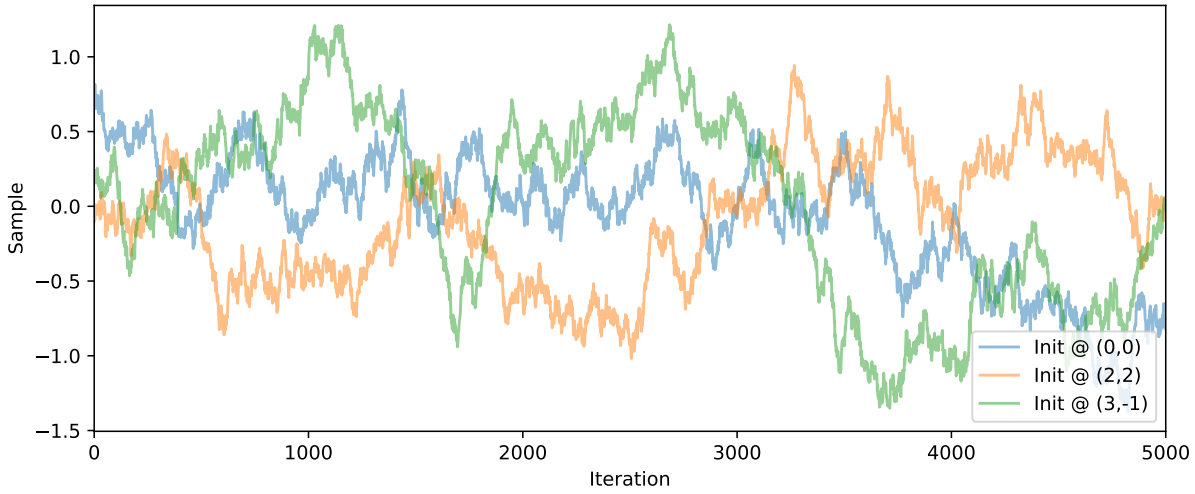
(e) In Metropolis-Hastings, any sample depends on the previously generated sample, and hence the algorithm generates samples that are generally statistically dependent. The *effective sample size* of a sequence of dependent samples is the number of independent samples that are, in some sense, equivalent to our number of dependent samples. A definition of the effective sample size is

$$\text{ESS} = \frac{S}{1 + 2\sum_{k=1}^{\infty} \rho(k)} \tag{10}$$

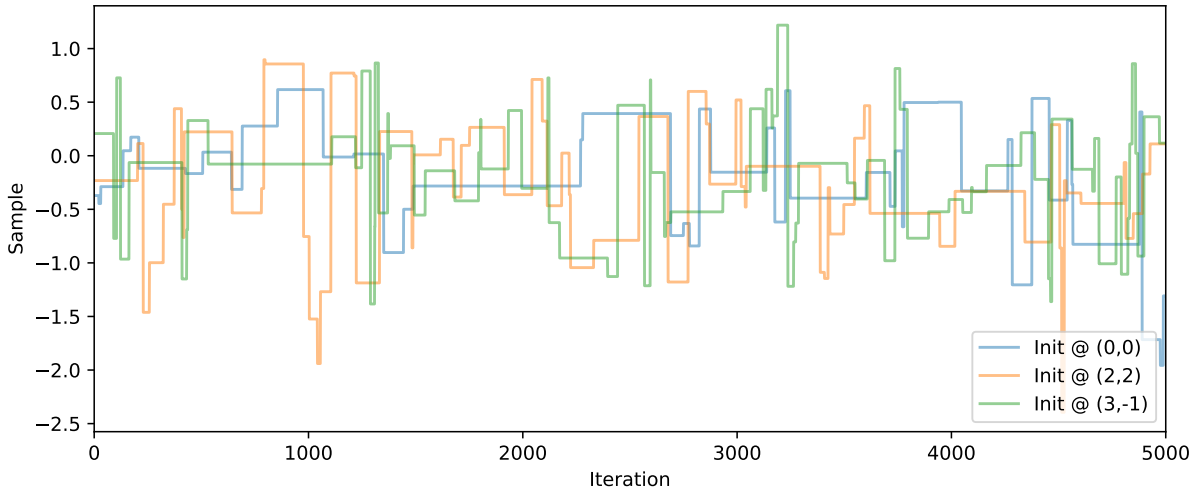
where $S$ is the number of dependent samples drawn and $\rho(k)$ the correlation coefficient between two samples in the Markov chain that are $k$ time points apart. We can see that if the samples are




(a) stepsize: 1




(b) Alternative stepsize




(c) Alternative stepsize

Figure 2: For Question (d): Trace plots of $\beta$ drawn using Metropolis-Hastings with different step sizes.

strongly correlated, $\sum_{k=1}^{\infty} \rho(k)$ is large and the effective sample size is small. On the other hand, if $\rho(k) = 0$ for all $k$, the effective sample size is $S$.

ESS, as defined above, is the number of independent samples which are needed to obtain a sample average that has the same variance as the sample average computed from correlated samples.

To illustrate how correlation between samples is related to an effective reduction of sample size, consider two pairs of samples $(\theta_1, \theta_2)$ and $(\omega_1, \omega_2)$. All variables have variance $\sigma^2$ and the same mean $\mu$, but $\omega_1$ and $\omega_1$ are uncorrelated while the covariance matrix for $\theta_1, \theta_2$ is $\mathbf{C}$,

$$\mathbf{C} = \sigma^2 \begin{pmatrix} 1 & \rho \\ \rho & 1 \end{pmatrix}, \tag{11}$$

with $\rho > 0$. The variance of the average $\bar{\omega} = 0.5(\omega_1 + \omega_2)$ is

$$\mathbb{V}(\bar{\omega}) = \frac{\sigma^2}{2}, \tag{12}$$

where the 2 in the denominator is the sample size.

Derive an equation for the variance of $\bar{\theta} = 0.5(\theta_1 + \theta_2)$ and compute the reduction $\alpha$ of the sample size when working with the correlated $(\theta_1, \theta_2)$. In other words, derive an equation of $\alpha$ in [4 MARKS]

$$\mathbb{V}\left(\bar{\theta}\right) = \frac{\sigma^2}{2/\alpha}. \tag{13}$$

What is the effective sample size $2/\alpha$ as $\rho \to 1$?

*[Expected response: Equations with the derivation of the variance of $\bar{\theta}$, formula for $\alpha$, effective sample size $2/\alpha$ for $\rho \to 1$.]*

# Question 4: Probabilistic programming with Stan (25 marks)

MCMC algorithms can be sensitive to their hyperparameters, and implementing and tuning model-specific inference algorithms can be difficult and time-consuming. Statistical/Probabilistic programming languages aim at providing an environment that automates the inference as much as possible so that the user can focus more on the modelling than the technical aspects of the inference.

In this question, you will learn about the probabilistic programming language Stan, and use it for probabilistic principal component analysis (PPCA). The details you need to know about Stan for this assignment are covered in Appendix B. However if you wish to find out more about the language, Section B.4 lists additional resources. Instructions on how to install Stan are provided in Appendix A.

(a) After installing Stan, read through Appendix B and implement the Poisson regression model from Question 3 (c) in Stan. Studying the related linear regression model from `https://mc-stan.org/docs/2_18/stan-users-guide/linear-regression.html` should be helpful too.

Use Stan to draw $5,000$ samples from the posterior density $p(\alpha, \beta \mid \mathcal{D})$, where $\mathcal{D}$ is given by `q3_poisson.txt`. For that, run Stan with **iter**=10000 and chains=1 (see Appendix B).

Make a scatter plot of the obtained samples and compare the trace plots of both $\alpha$ and $\beta$ with those in Figure 2. Has the chain well "mixed"? Compute the posterior mean of $\alpha$ and $\beta$, as well as the posterior correlation coefficient between them. Do you get similar results as in Question 3 (c)? [8 MARKS]

*[Expected response: A Stan program implementing the Poisson regression model. Three plots: a scatter plot for the posterior, a trace plot for $\alpha$, and a trace plot for $\beta$. Numerical values for the posterior means and correlation coefficient. 1-2 concluding sentences.]*

(b) Assume we have data points $\mathbf{x}_n$, $n = 1, \ldots, N$, where each $\mathbf{x}_n$ is a $D$-dimensional vector of numbers, i.e. $\mathbf{x}_k \in \mathbb{R}^D$. The probabilistic PCA model assumes that each $\mathbf{x}_n$ can be represented in terms of fewer unobserved (latent) variables $\mathbf{z}_n \in \mathbb{R}^K$, $K < D$. The generative model is

$$\mathbf{x}_n = \mathbf{W}\mathbf{z}_n + \boldsymbol{\epsilon}, \qquad \boldsymbol{\epsilon} \sim \mathcal{N}(\boldsymbol{\epsilon}; \mathbf{0}, \sigma^2 I), \qquad \mathbf{z}_n \sim \mathcal{N}(\mathbf{z}_n; 0, I), \tag{14}$$

where $I$ denotes the identity matrix.[3] The matrix $\mathbf{W}$ is of size $D \times K$.

In the lecture, we have discussed how to learn the parameters by maximum likelihood estimation. Here, we consider Bayesian inference instead, which involves choosing a prior over $\mathbf{W}$ and $\sigma^2$ (or its inverse $\tau = 1/\sigma^2$) and determining their posterior. The resulting data analysis method is then called Bayesian PCA (Bishop, 1999).

The provided archive file contains the file `pca.stan`[4]— a Stan program implementing Bayesian PCA. Open the file and familiarise yourself with the code. Write down an expression for the factorised joined density $p(\mathbf{x}_1, \ldots, \mathbf{x}_N, \mathbf{z}_1, \ldots, \mathbf{z}_N, \mathbf{W}, \tau, \boldsymbol{\alpha})$ described by the program, in terms of the conditional Gaussian pdfs and Gamma pdfs. Identify any hyperparameters that may exist in the model. (Note Stan's convention of parametrising Gaussians in terms of the standard deviation.) [2 MARKS]

*[Expected response: A factorised expression for $p(\mathbf{x}_1, \ldots, \mathbf{x}_N, \mathbf{z}_1, \ldots, \mathbf{z}_N, \mathbf{W}, \tau, \boldsymbol{\alpha})$ and a list of hyperparameters]*

(c) In the provided archive file, you will find the file `q4_pca.txt` that contains the matrix $X$ with 15 data points $\mathbf{x}_i \in \mathbb{R}^3$ (i.e. $D = 3, N = 15$). Use the provided Stan code `pca.stan` to obtain posterior samples for $\mathbf{W}$ and $\tau = 1/\sigma^2$ given $X$.

If the Stan model was compiled in Python with `sm_pca = pystan.StanModel(file='pca.stan')`, for the sampling, use

```
data = {"N": 15, "D": 3, "K": 2, "X": X.T, "mu_W": 0, "alpha0": 1, "
    beta0": 1}
result = sm_pca.sampling(data=data, iter=10000, chains=2, n_jobs=2)
```

[3] Compared to the lecture, we here use $\mathbf{x}$ instead of $\mathbf{v}$ to denote the "visibles", $\mathbf{z}$ instead of $\mathbf{h}$ to denote the latents, and $\mathbf{W}$ instead of $\mathbf{F}$ for the matrix with the factor loadings. This is a common alternative notation for PPCA. We also assume that the data $\mathbf{x}_n$ has been preprocessed so that its mean is zero; hence we can set the additive constant $\mathbf{c}$ from the lecture to zero.

[4] Note that there are more efficient ways of writing the model in Stan than what is given in `pca.stan`. In particular, the program could be optimised by vectorising the **for**-loops using the `to_vector` function.

where `mu_W`, `alpha0` and `beta0` are hyperparameters for the priors in the model. If you work under windows, you may have to set `n_jobs=1`.

This will generate 10′000 posterior samples $\mathbf{W}^{(i)}, \tau^{(i)}$, $i = 1, \ldots, 10'000$, where each matrix $\mathbf{W}^{(i)}$ is $D \times K$, i.e. in our case: $\mathbf{W}^{(i)} = (\mathbf{w}_1^{(i)}, \mathbf{w}_2^{(i)})$.

In the lecture, we have have seen that for a given factor matrix $\mathbf{W}$ and variance $\sigma^2 = 1/\tau$ of $\boldsymbol{\epsilon}$, the visibles $\mathbf{x}$ have covariance equal to $\mathbf{W}\mathbf{W}^\top + \sigma^2 I$. The posterior samples returned by Stan represent factor matrices $\mathbf{W}^{(i)}$ and variances $\sigma_i^2 = 1/\tau^{(i)}$ that are plausible given the data. Each sample thus corresponds to a plausible covariance matrix $\mathbf{C}^{(i)}$,

$$\mathbf{C}^{(i)} = \mathbb{V}\left(\mathbf{x}|\mathbf{W}^{(i)}, \tau^{(i)}\right) \tag{15}$$

$$= \mathbf{W}^{(i)}(\mathbf{W}^{(i)})^\top + \frac{1}{\tau^{(i)}} I \tag{16}$$

We are now going to treat each matrix $\mathbf{C}^{(i)}$ as if it were an empirical covariance matrix and apply standard (non-probabilistic) PCA on it. For that purpose, pick randomly 25 of the 10′000 samples $\mathbf{W}^{(i)}, \tau^{(i)}$, compute the corresponding covariance matrix $\mathbf{C}^{(i)}$, and determine the two principal eigenvectors $\mathbf{v}_k^{(i)}$ and eigenvalues $\lambda_k^{(i)}$ of $\mathbf{C}^{(i)}$ (i.e. the two eigenvectors with the largest eigenvalues). The eigenvector associated with the largest eigenvalue is the direction in the data space along which the variance is largest (equivalently for the eigenvectors with the next largest eigenvalues). The eigenvalues themselves are the variances along those directions. The rescaled eigenvectors $\mathbf{u}_k^{(i)} = \mathbf{v}_k^{(i)} \sqrt{\lambda_k^{(i)}}$ define the same subspace as the $\mathbf{v}_k^{(i)}$ but their length conveniently reflects their importance.

Make a figure that includes:

- A scatter plot of the data $\mathbf{x}_n$, projected onto the first two dimensions (i.e. the $xy$-plane)
- 25 pairs of rescaled eigenvectors $\mathbf{u}_1^{(i)}, \mathbf{u}_2^{(i)}$
- the rescaled eigenvectors for the empirical covariance matrix computed from $X$ (i.e. the results for standard PCA)

(The rescaled eigenvectors are three dimensional. In the figure, just show their projection onto the $xy$-plane, i.e. the same space as the data). How does the distribution of $\mathbf{u}_1^{(i)}, \mathbf{u}_2^{(i)}$ compare to the vectors obtained by standard PCA? [8 MARKS]

*[Expected response: A figure showing a scatter plot of the data, together with samples of the rescaled eigenvectors. Brief comparison to the solution with standard PCA. Computer code used to get the results.]*

(d) Denoting the $(i, j)$-th element of the matrix $\mathbf{W}$ by $W_{ij}$, make a scatter plot showing the sampled elements $W_{11}$ and $W_{12}$, i.e. the first two elements of the first *row* of $\mathbf{W}$. Explain the observed shape of the scatter plot.

Change the value of the hyperparameter `mu_W` such that the prior on $\mathbf{W}$, `W[,k] ~ normal(mu_W, t_alpha[k]);`, is of mean $\mathbf{1}$, instead of mean $\mathbf{0}$ as above. Again, make a scatter plot showing the samples of the first row of $\mathbf{W}$. Explain the difference in the shape of the obtained scatter plot. [7 MARKS]

*[Expected response: Two scatter plots and two explanations]*

# Appendix

## A Stan: Installation and setup instructions

We strongly recommend using Stan from Python for this assignment. However, you can use Stan from any other of its supported host languages (Python, R, Matlab, Julia, Stata, Mathematica, Scala), though support for installation problems won't be provided.

In this appendix, you can find instructions to how to install and use Stan through its Python interface, PyStan. You can find more about other available interfaces here: https://mc-stan.org/users/interfaces/.

### Using Stan in Google Colaboratory

The easiest way to use Stan through its Python interface is Google Colaboratory (https://colab.research.google.com/). Colab is an online environment, which allows creating and running Python notebooks in the cloud. Stan is already installed in Google Colab, thus all you need to do is import the PyStan package:

```
import pystan
```

### Installing PyStan on DICE

Note: You will require around 400MB of free space.

Installation instructions on DICE:

1. Make a new Python virtual environment named, say, stan. You can do that using virtualenv for Python 2 and pyvenv for Python 3. For Python 3:

   ```
   > pyvenv stan
   ```

2. Activate the virtual environment:

   ```
   > cd stan
   > source ./bin/activate
   ```

3. Install pystan using pip:

   ```
   > pip install pystan
   ```

4. Check if your setup is working correctly:

   ```
   > python
   > >>> import pystan
   > >>> code = """parameters {real x;} model {x ~ normal(0, 1);}"""
   > >>> sm = pystan.StanModel(model_code=code)
   > >>> fit = sm.sampling()
   > >>> print(fit)
   ```

You should see an output similar to the following:

```
Inference for Stan model: anon_model_2a7b38549c0f58813ec10359085df7b9.
4 chains, each with iter=2000; warmup=1000; thin=1;
post-warmup draws per chain=1000, total post-warmup draws=4000.

       mean se_mean     sd   2.5%    25%    50%    75%  97.5%  n_eff   Rhat
x     -0.01    0.03   1.01  -2.02  -0.69 3.5e-3   0.66   1.95   1390    1.0
lp__  -0.51    0.02   0.74   -2.6  -0.68  -0.23 -0.05-4.5e-4   2267    1.0

Samples were drawn using NUTS at Tue Mar 12 11:54:05 2019.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
```

## Installing PyStan on your own machine

Stan compiles to C++, meaning that Python needs access to a C++ compiler. If you already have a C++ compiler installed, which is available to Python, installing pystan should be as easy as pip install pystan.

For more information on how to install pystan on your machine (Unix-based or Windows), refer to the official pystan installation instructions here: https://pystan.readthedocs.io/en/latest/getting_started.html

Note that if you use Stan through Python 2 on Windows, you *must* use only one parallel chain while sampling, by specifying n_jobs=1. That is, draw samples using `sm.sampling(data=data, n_jobs=1)`.

# B Introduction to Stan

Stan (Carpenter et al., 2017) is a typed, imperative probabilistic programming language, which is used for statistical modelling in physics, biology, social science, business, and other areas. It offers efficient automatic inference, diagnostics toolkit, and visualisation packages. It can be used from many host languages, including Python, Matlab, R, and Julia.

One of the reasons why Stan has been very successful is that it uses the efficient inference algorithm Hamiltonian Monte Carlo (HMC) (Neal et al., 2011).[5] You do not need to know how HMC works for this assignment, but it is important to know it uses the gradient of the target density to sample more efficiently. Stan also provides support for other gradient-based methods, such as automatic-differentiation variational inference.

Sections B.1 and B.2 give a short introduction to Stan's syntax, and explain how to run Stan from its Python interface, PyStan. You might want to read these sections before attempting Question 4, while Section B.3 is meant as a reference. Finally, if you are interested in reading more about Stan or HMC, Section B.4 lists some useful resources.

## B.1 Basic Stan syntax

Programs in Stan consist of up to 7 *named program blocks* that appear in order. Some of the blocks are:

- The **data** block: it contains type declarations for the observed data $\mathcal{D}$ and fixed quantities. They correspond to inputs.
- The **parameters** block: it contains type declarations for the model parameters $\boldsymbol{\theta}$. These are the variables being sampled and returned as part of the output by Stan.
- The **transformed parameters** block: it contains imperative code that transforms the data or parameters in some *deterministic* way. The transformed parameters are also returned as part of the output by Stan.
- The **model** block: it contains imperative code that describes the joint distribution $p(\mathcal{D}, \boldsymbol{\theta})$. For the purposes of this coursework, you can think of the Stan **model** block as describing a *generative process*, where a statement such as `x ~ normal(0, 1);` means "generate a random variable `x` from a normal distribution with mean zero and standard deviation 1". However, keep in mind this is not true in general.

Other than this, the syntax is close to C++. For example, suppose we are to code a simple model with two parameters, $x$ and $y$, and with no observed data, where:

$$x \sim \mathcal{N}(x; 0, 1) \qquad y \sim \mathcal{N}(y; x, 1)$$

To sample from the joint density $p(x, y) = \mathcal{N}(x; 0, 1)\mathcal{N}(y; x, 1)$ using Stan, we simply omit all other blocks, except the **parameters** and **model** blocks, and we write:

```
parameters {
  real x;
  real y;
}
model {
  x ~ normal(0, 1);
  y ~ normal(x, 1);
}
```

Now, consider the same model, but where $y$ is observed data. To sample from the posterior $p(x \mid y)$ using Stan, we write:

[5] More specifically, Stan uses an enhanced version of the No-U-Turn Sampler (Hoffman and Gelman, 2014).

```
data {
  real y;
}
parameters {
  real x;
}
model {
  x ~ normal(0, 1);
  y ~ normal(x, 1);
}
```

We can, of course, have a vector of data points $\mathbf{y}$ instead, as long as we input the length of that vector as observed data. Consider $N$ variables $y_n$, with the same mean $x$:

$$x \sim \mathcal{N}(x; 0, 1) \qquad y_n \sim \mathcal{N}(y_n; x, 1) \text{ for n} = 1, \ldots, \text{N}$$

To sample from $p(x \mid y_1, \ldots, y_N)$, we write:

```
data {
  int N;
  vector[N] y;
}
parameters {
  real x;
}
model {
  x ~ normal(0, 1);
  for (n in 1:N){
    y[n] ~ normal(x, 1);
  }
}
```

The final **for**-loop can be alternatively written in a *vectorised* way: `y ~ normal(x, 1)`. This is similar to NumPy's broadcasting. Vectorising your code can result in speeding up inference significantly.

Finally, we can also have standard imperative code in any of Stan's blocks, with the exception of **data** and **parameters**. For example, we add a precision parameter $\tau$, and derive the standard deviation $\sigma = 1/\sqrt{\tau}$, so that:

$$x \sim \mathcal{N}(x; 0, 1) \qquad \tau \sim \text{Gamma}(\tau; 1, 1) \qquad y_n \sim \mathcal{N}\left(y_n; x, \frac{1}{\tau}\right) \text{ for n} = 1, \ldots, \text{N} \tag{17}$$

where $\text{Gamma}(\tau; \alpha, \beta)$ denotes the probability density function of a Gamma random variable,

$$\text{Gamma}(\tau; \alpha, \beta) = \frac{\beta^\alpha}{\Gamma(\alpha)} \tau^{\alpha-1} \exp(-\beta\tau) \qquad \Gamma(\alpha) = \int_0^\infty u^{\alpha-1} \exp(-u)\, du \tag{18}$$

To include $\sigma$ in Stan's output, we define it in the **transformed parameters** block:

```
data {
  int N;
  vector[N] y;
}
parameters {
  real x;
  real<lower=0> tau;
}
```

```
transformed parameters {
  real<lower=0> sigma;
  sigma = 1 / sqrt(tau);
}
model {
  x ~ normal(0, 1);
  tau ~ gamma(1, 1);
  for (n in 1:N){
    y[n] ~ normal(x, sigma);
  }
}
```

Note that Stan expects the *standard deviation* and not the variance as second argument in `normal(x,sigma)`.

## B.2 Compiling Stan, running inference, and interpreting the results

This section assumes usage of Stan's Python interface: PyStan.

For models that are not very long, the simplest way to compile the model with Stan is by providing the model as a string to PyStan's `StanModel` method. The following is Python code that you can copy-paste to your Python console (assuming that you have installed Stan as described in Section A.)

```
import pystan

code = """
data {
  int N;
  vector[N] y;
}
parameters {
  real x;
}
model {
  x ~ normal(0, 1);
  y ~ normal(x, 1);
}
"""
sm = pystan.StanModel(model_code=code)
```

This will compile the Stan model to C++ code, which can take a couple of minutes. However, recompilation is *not* required afterwards, unless the model is changed. If the Stan-code is stored in a file `my_code.stan`, you can compile the model with

```
import pystan
sm = pystan.StanModel(file='my_code.stan')
```

Once the model is compiled, you can use it to sample from the posterior given some data. The data is fed in as a Python dictionary:

```
observed_ys = [0.2484, -1.2353, -1.2180, -0.7246, 0.0731]
results = sm.sampling(data={"N": len(observed_ys), "y": observed_ys})
```

You can also change the number of iterations per chain (by setting the `iter` argument) and the number of chains (the `chains` argument), for example:

```
results = sm.sampling(data={"N": len(observed_ys),"y": observed_ys}, iter=8000, chains
    =10)
```

The object results contains the drawn samples, together with additional inference information. You can extract the samples from the posterior using the extract() method:

```
x_samples = results.extract()["x"]
```

Finally, if you print the entire results object with print(results), you will get a summary, which includes the sample mean and standard deviation, the number of effective samples, and an $\hat{R}$ value:

```
Inference for Stan model: anon_model_c3e4d07fbbaa7e25705de34878f65a30.
4 chains, each with iter=2000; warmup=1000; thin=1;
post-warmup draws per chain=1000, total post-warmup draws=4000.

       mean se_mean     sd   2.5%    25%    50%    75%  97.5%  n_eff   Rhat
x     -0.46    0.01    0.4  -1.25  -0.74  -0.47   -0.2   0.34   1401    1.0
lp__   -1.6    0.01   0.69  -3.65  -1.74  -1.34  -1.17  -1.12   2183    1.0

Samples were drawn using NUTS at Tue Mar 12 11:02:59 2019.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at convergence, Rhat=1).
```

$\hat{R}$ (Rhat) is called the potential scale reduction factor, and is a convergence diagnostic that is calculated from the samples variance within and across different Markov chains. Ideally, $\hat{R}$ must be close to 1. If it is too small or too big, that indicates the chains have not converged.

## B.3 Distributions, functions and types

This section contains a quick reference guide to some common functions, distributions and types that Stan supports and you may need in this assignment. For a full specification you can check the Stan language reference manual or functions reference (see Section B.4).

### B.3.1 Distributions

- **Gaussian** (with mean mu and standard deviation sigma): `x ~ normal(mu, sigma);`
- **Gamma** (with shape alpha and rate beta): `x ~ gamma(alpha, beta);`
- **Exponential** (with rate beta): `x ~ exponential(beta);`
- **Poisson** (with rate lambda): `x ~ poisson(lambda);`

### B.3.2 Functions and statements

- Standard imperative statements, such as **if-else** statements, **for** loops and **while** loops.
- Standard maths functions, such as sqrt(x), exp(x), log, square, pow and others.
- Elementwise multiplication and inverse of vectors or matrices: `x .* y` and `x ./ y`.
- Reshape functions, such as to_vector(x), which transforms a matrix x to a column vector.

### B.3.3 Types

- **Real scalar**: **real** `x;`
- **Integer scalar**: **int** `x;`

- **Vector of length $N$**: `vector[N] x;`
- **Matrix of $M$ rows and $N$ columns**: `matrix[M, N] x;` [6]
- **Array of $N$ elements of type T**: `T x[N];`
- **Constrained real, such that it is a real between $i$ and $j$**: `real<lower=i, upper=j> x;`

  Other types can also be constrained and they can be constrained from below, from above, or both. For example a vector with positive values: `vector<lower=0>[N] x;`

If $\mathbf{A}$ is a matrix $\mathbf{A}[k]$ picks *row* $k$ from that matrix while $\mathbf{A}[, k]$ picks *column* $k$.

### B.4 Resources

1. Stan
   - Official web page: https://mc-stan.org/
   - GitHub repositories: https://github.com/stan-dev/
   - Reference manual: https://mc-stan.org/docs/2_18/reference-manual/
   - User's guide: https://mc-stan.org/docs/2_18/stan-users-guide/
   - Functions reference: https://mc-stan.org/docs/2_18/functions-reference/
   - Python interface to Stan: https://pystan.readthedocs.io/
2. Hamiltonian Monte Carlo (HMC):
   - The original HMC paper by Neal et al. (2011) gives some excellent intuition behind the algorithm.
   - MacKay (2003, § 30.1) provides a short introduction to HMC together with some Octave implementation.
   - Betancourt (2017) presents an in-depth discussion of HMC, starting from the very basics and building up to the state of the art.
   - A short animation illustrating the difference between Metropolis-Hastings and Hamiltonian Monte Carlo: https://www.youtube.com/watch?v=Vv3f0QNWvWQ/.

## References


Barber, D. (2012). *Bayesian Reasoning and Machine Learning*. Cambridge University Press.

Betancourt, M. (2017). A conceptual introduction to Hamiltonian Monte Carlo. *arXiv preprint arXiv:1701.02434*.

Bishop, C. M. (1999). Bayesian PCA. In *Advances in neural information processing systems*, pages 382–388.

Carpenter, B., Gelman, A., Hoffman, M., Lee, D., Goodrich, B., Betancourt, M., Brubaker, M., Guo, J., Li, P., and Riddell, A. (2017). Stan: A probabilistic programming language. *Journal of Statistical Software, Articles*, 76(1):1–32.

Hoffman, M. D. and Gelman, A. (2014). The No-U-turn sampler: adaptively setting path lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research*, 15(1):1593–1623.

MacKay, D. J. (2003). *Information theory, inference and learning algorithms*. Cambridge university press.

Neal, R. M. et al. (2011). MCMC using Hamiltonian dynamics. *Handbook of Markov Chain Monte Carlo*, 2(11).


[6] The elements of both vectors and matrices are real variables.