1

Bayesian inference and computation: a beginner's guide

Brendon J. Brewer
Department of Statistics, The University of Auckland

1.1 Introduction

Most scientific observations are not sufficient to give us definite answers to all our questions. It is rare that we get a dataset which *totally* answers every question with certainty. Even if that did happen, we would quickly move on to other questions. What a dataset usually *can* do is make hypotheses more or less plausible, even if we don't achieve total certainty. Bayesian inference is a model of this reasoning process, and also a tool we can use to make quantitative statements about how much uncertainty we should have about our conclusions. This takes the mystery out of data analysis, because we no longer have to come up with a new method every time we face a new problem. Instead, we simply specify exactly what information we are going to use, and then compute the results. In the last two decades, Bayesian inference has become immensely popular in many fields of science, and astrophysics is no exception. Therefore it is becoming increasingly important for researchers to have at least a basic understanding of these methods.

Accessible textbooks for those with a physics background are those by Gregory (2005) and Sivia and Skilling (2006), and parts of the textbook by Mackay (2003)†. The online tutorial by Jake Vanderplas is also useful‡. For those with a strong statistics background, I recommend the books by O'Hagan and Forster (2004) and Gelman et al (2013). I also maintain a set of lecture notes for an undergraduate Bayesian statistics course§. The aim of this chapter is to present a fairly minimal yet widely applicable set

† Freely available online at
`www.inference.phy.cam.ac.uk/itila/`
‡ Available online at
`jakevdp.github.io/blog/2014/03/11/frequentism-and-bayesianism-a-practical-intro/`
§ Available at
`www.github.com/eggplantbren/STATS331/`

of techniques to allow you to start using Bayesian inference in your own research.

Any particular application of Bayesian inference involves making choices about what data you are analysing, what questions you are trying to answer, and what assumptions you are willing to make. Data analysis problems in astronomy vary widely, so in this chapter we cannot cover a huge variety of examples. Instead, we will only study a single example, but will spend a lot of time looking at the methods and thinking that go into such an analysis, and will be applicable in other examples. The specific assumptions we make in the example will not always be appropriate, but they should be sufficient to show you the points at which assumptions are needed, and what you will need to consider when you work on a particular problem.

In principle, it's usually best to work with your data in the most raw form possible, although this is often too difficult in practice. Therefore, most scientists work with data that has been processed (by a "pipeline") and reduced to a manageable size. While many Bayesian practitioners often have strong ideals about data analysis, a large dose of pragmatism is still very necessary in the real world.

To do Bayesian inference, you need to specify what *prior information* you have (or are willing to assume) about the problem, in addition to the data. Prior information is necessary; what you can learn from a dataset depends on what you know about how it was produced. Once you have your data, and have specified your prior information as well, you are faced with the question of how to calculate the results. Usually you want to calculate the *posterior distribution* for some unknown quantities (also known as "parameters") given your data. This posterior distribution describes your uncertainty about the parameters, but takes the data into account. Since we are often dealing with (potentially) complicated probability distributions in high dimensional spaces, we need to *summarise* the posterior distribution in an understandable way. In certain problems, the summaries can be calculated analytically, but numerical methods are more general, so will be the focus of this chapter. The most popular and useful numerical techniques are the Markov Chain Monte Carlo methods, often abbreviated as MCMC†. The rediscovery of MCMC in the 1990s is one of the main reasons why Bayesian inference is so popular now. While there were many strong philosophical arguments in favour of a Bayesian approach before then, many people were still uncomfortable with the subjective elements involved. However, once

† Some people claim that MCMC stands for Monte Carlo Markov Chains, but they are wrong.

MCMC made it easy to compute the consequences of Bayesian models more easily, people simply became more relaxed about these subjective elements.

A large number of MCMC methods exist, and it would be unwise to try to cover them all in this winter school. Therefore I will focus on a small number of methods that are relatively simple to implement, yet quite powerful and widely applicable. I will try to emphasise methods that are *general*, i.e. methods that will work okay on most problems you might encounter. One disadvantage of this approach is that the methods we cover are not necessarily the most efficient methods possible. If you're mostly interested in one specific application, you'll probably be able to achieve better performance by using a more sophisticated algorithm, or by taking advantage of the particular properties of your problem. There are many popular software packages (and many more unpopular ones) available for doing Bayesian Inference, such as JAGS (Just Another Gibbs Sampler), Stan, emcee, MultiNest, my own DNest3, and many more. I won't be teaching you how to use any of these programs. If you or your collaborators already use one of them, that's a great reason to learn it. Please see the appendix for a brief discussion of the advantages and disadvantages of some of these packages.

1.2 Python

Due to its popularity and relatively shallow learning curve, I have implemented the algorithms in this chapter in the Python language. The code is written so that it works in either Python 2 or 3. The programs make use of the common numerical library `numpy`, and also the plotting package `matplotlib`. Any Python code snippets in this chapter will assume that the following packages have been imported:

```
import numpy as np
import numpy.random as rng
import matplotlib.pyplot as plt
import copy
import scipy.special
```

Full programs implementing the methods (and the particular problems) used in this chapter are provided online at the following URL:

`http://www.iac.es/winterschool/2014/pages/teaching-material.php`

1.3 Parameter Estimation

Almost all data analysis problems can be interpreted as *parameter estimation* problems. The term *parameter* has a few different meanings, but you can usually just think of it as a synonym for *unknown quantity*. When you learned how to solve equations in high school algebra, you would have been able to find the value of an unknown quantity (often called x) when you had enough information to determine its value with certainty. In science, we almost never have enough information to determine a quantity without any uncertainty, which is why we need probability theory and Bayesian inference.

We'll denote our unknown parameters by θ, which could be a single parameter or perhaps a vector of parameters (e.g. the distance to a star and the angular diameter of the star). To start, we need to have some idea of the set of possible values we are considering. For example, are the parameters integers? Real numbers? Positive real numbers? In some examples, the definition of the parameters already restricts the set of possible values. For example, *the proportion of extra-solar planets in the Milky Way that contain life* cannot be less than 0 or greater than 1. Strictly speaking, it has to be a rational number, but it probably won't make much difference if we just say it's a real number between 0 and 1 (inclusive). The distance to a star (measured in whatever units you like) is presumably a positive real number, as is its angular diameter. The set of possible values you're willing to consider is called the *hypothesis space*.

To start using Bayesian inference, you need to assign a *probability distribution* on the hypothesis space, which models your initial uncertainty about the parameters. This probability distribution is called the *prior*. We then use Bayes' rule, a consequence of the product rule of probability, to calculate the *posterior* distribution, which describes our updated state of knowledge about the values of the parameters, after taking the data D into account.

For a prior distribution $p(\theta|I)$ (read as "the probability distribution for θ given I), and a sampling distribution $p(D|\theta, I)$, Bayes' rule allows us to calculate the *posterior distribution* for θ:

$$p(\theta|D, I) = \frac{p(\theta|I)p(D|\theta, I)}{p(D|I)}. \quad (1.1)$$

The I in this equation refers to background information and assumptions; basically, it stands for everything you know about the problem apart from the data. The I appears in the background of all of the terms in Equation 1.1, and is often omitted. Note also that the notation used in Equation 1.1 is highly simplified but conventional among Bayesians; see the appendix for a discussion of notation. For brevity we can suppress the I

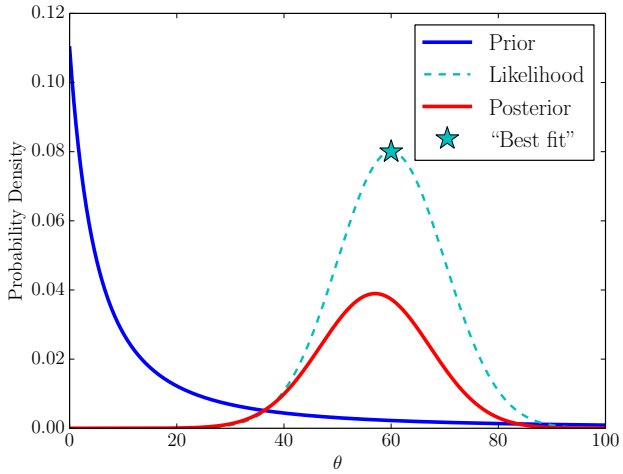

Fig. 1.1. *An example prior distribution for a single parameter (blue) gets updated to the posterior distribution (red) by the data. The data enters through the likelihood function (cyan dotted line). Many traditional "best fit" methods are based on finding the maximum likelihood estimate, which is the peak of the likelihood function, here denoted by a star.*

(remove it from the right hand side of all equations) and just write:

$$p(\theta|D) \quad = \quad \frac{p(\theta)p(D|\theta)}{p(D)}. \tag{1.2}$$

The result is a probability distribution for θ which describes our state of knowledge about θ after taking into account the data. The denominator, since it doesn't depend on θ, is a normalising constant, usually called the *marginal likelihood* or alternatively the *evidence*. Since the posterior is a probability distribution, its total integral (or sum, if the hypothesis space is discrete) must equal 1. Therefore we can write the marginal likelihood as:

$$p(D|I) \quad = \quad \int p(\theta|I)p(D|\theta, I)\, d\theta \tag{1.3}$$

where the integral is over the entire N-dimensional parameter space.

The posterior distribution is usually narrower than the prior distribution, indicating that we have learnt something from the data, and our uncertainty about the value of the parameters has decreased. See Figure 1.1 for an example of the qualitative behaviour we usually see when updating from a prior distribution to a posterior distribution.

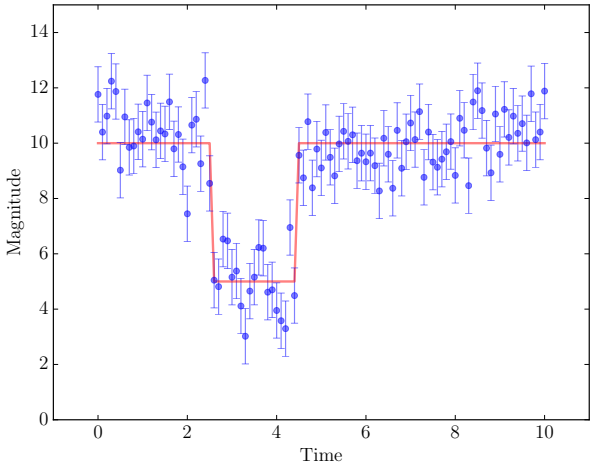

Fig. 1.2. *The "transit" dataset. The red curve shows the model prediction based on the true values of the parameters, and the blue points are the noisy measurements.*

1.4 Transit Example

To become more familiar with Bayesian calculations, we will work through a simple curve-fitting example. Many astronomical data analysis problems can be viewed as examples of curve fitting. Consider a transiting exoplanet, like one observed by *Kepler*. The light curve of the star will show an approximately constant brightness as a function of time, with a small dip as the exoplanet moves directly in front of the star. Clearly, real Kepler data is much more complex than this example, as stars vary in brightness in complicated ways, and the shape of the transit signal itself is more complex than the model we'll use here. Nevertheless, this example contains many of the features and complications that will also arise in a more realistic analysis.

The dataset, along with the true curve, is shown in Figure 1.2. The equation for the true curve is:

$$\mu(t) = \begin{cases} 10, & 2.5 \leq t \leq 4.5 \\ 5, & \text{otherwise.} \end{cases}$$

Let's assume we don't know the equation for the true curve (as we wouldn't in reality), but we at least know it's a function of the following form:

$$\mu(t) = \begin{cases} A, & (t_c - w/2) \leq t \leq (t_c + w/2) \\ A - b, & \text{otherwise.} \end{cases}$$

where A is the brightness away from the transit, b is the depth of the transit, t_c is the time of the centre of the transit, and w is the width of the transit.

Thus, the problem has been reduced from not knowing the true curve $\mu(t)$ to not knowing the values of four quantities (parameters) A, b, t_c, and w. Applying Bayesian inference to this problem involves calculating the posterior distribution for A, b, t_c, and w, given the data D. For this specific setup, Bayes' rule states:

$$p(A, b, t_c, w|D) = \frac{p(A, b, t_c, w)p(D|A, b, t_c, w)}{p(D)} \tag{1.4}$$

So, in order for the posterior distribution to be well defined, we need to choose a prior distribution $p(A, b, t_c, w)$ for the parameters, and a sampling distribution $p(D|A, b, t_c, w)$ for the data. Since the denominator $p(D)$ is not a function of the parameters, it plays the role of a normalising constant that ensures the posterior distribution integrates to 1, as any probability distribution must.

1.4.1 Sampling Distribution

The *sampling distribution* is the probability distribution we would assign for the data, if we knew the true values of the parameters. A useful way to think about the sampling distribution is to write some code whose input is the true parameter values, and whose output is a simulated dataset. Whatever probability distribution you use to simulate your dataset is your sampling distribution.

In many situations, it is conventional to assign a normal distribution (also known as a gaussian distribution) to each data point, where the mean of the normal distribution is the noise-free model prediction, and the standard deviation of the normal distribution is given by the size of the error bar. Later, we will see how to relax these assumptions in a useful way. The probability density for the data given the parameters (i.e. the sampling distribution) is:

$$p(D|A, b, t_c, w) = \prod_{i=1}^{N} \frac{1}{\sigma_i\sqrt{2\pi}} \exp\left[-\frac{1}{2\sigma_i^2}\left(D_i - \mu(t_i)\right)^2\right] \tag{1.5}$$

This is a product of N terms, one for each data point, and is really a probability distribution over the N-dimensional space of possible datasets. We have assumed that each data point is *independent* (given the parameters). That is, if we knew the parameters and a subset of the data points, we would

use *only* the parameters (not the data points) to predict the remaining data points.

When the dataset is known, Equation 1.5 becomes a function of the parameters only, known as the likelihood function. The curve predicted by the model, here written as $\mu(t_i)$ (where I have suppressed the implicit dependence on the parameters), provides the mean of the normal distribution. Remember that the independence assumption is not an assumption about the actual dataset, but an assumption about our prior information about the dataset. It does not make sense to say a particular dataset is or is not independent. Independence is a property of probability distributions.

Equation 1.5 is the sampling distribution (and the likelihood function) for our problem, but it is fairly cumbersome to write down. Statisticians have developed a shorthand notation for writing down probability distributions. This is extremely useful for communicating your assumptions without having to write down the entire probability density equation. To communicate Equation 1.5, we can simply write:

$$D_i \sim \mathcal{N}\left(\mu(t_i), \sigma_i^2\right) \tag{1.6}$$

i.e. each data point has a normal distribution (denoted by $\mathcal{N}$) with mean $\mu(t_i)$ (which depends on the parameters) and standard deviation σ. For the normal distribution, it is traditional to write the *variance* (standard deviation squared) as the second argument, but since the standard deviation is a more intuitive quantity (being in the same units as the mean), we often literally write the standard deviation, squared (e.g. 3^2). For other probability distributions the arguments in the parentheses are whatever parameters make sense for that family of distributions.

1.4.2 Priors

Now we need a prior for the unknown parameters A, b, t_c, and w. This is a probability distribution over a four dimensional parameter space. To simplify things, we can assign independent priors for each parameter, and multiply these together to produce the joint prior:

$$p(A, b, t_c, w) \quad = \quad p(A)p(b)p(t_c)p(w). \tag{1.7}$$

This prior distribution models our uncertainty about the parameters before taking into account the data. The independence assumption implies that if we were to learn the value of one of the parameters, this wouldn't tell us anything about the others. This may or may not be realistic in a given application, but it is a useful starting point.

Another useful starting point for priors is the uniform distribution, which has a constant probability density between some lower and upper limit. Let's use four uniform distributions for our priors:

$$\begin{aligned} A &\sim U(-100, 100) && (1.8)\\ b &\sim U(0, 10) && (1.9)\\ t_c &\sim U(t_{\min}, t_{\max}) && (1.10)\\ w &\sim U(0, t_{\max} - t_{\min}) && (1.11) \end{aligned}$$

The full expression for the joint prior probability density is:

$$p(A, b, t_c, w) = \begin{cases} \frac{1}{2000(t_{\max}-t_{\min})^2}, & (A, b, t_c, w) \in S \\ 0, & \text{otherwise} \end{cases} \quad (1.12)$$

where S is the set of allowed values. Even more simply, we can ignore the normalising constant and the prior boundaries and just write:

$$p(A, b, t_c, w) \propto 1, \quad (1.13)$$

although if we use this shortcut, we must remember that the boundaries are implicit.

Now that we have specified our assumed prior information in the form of a sampling distribution and a prior, we are ready to go. By Bayes' rule, we have an expression for the posterior distribution immediately:

$$p(A, b, t_c, w|D) \propto p(A, b, t_c, w)p(D|A, b, t_c, w) \quad (1.14)$$

which is proportional to the prior times the likelihood. In the likelihood expression we would substitute the actual observed dataset into the equation, so that it is a function of the parameters only. The main problem with using Bayes' rule this way is that a mathematical expression for a probability distribution in a four-dimensional space is not very easy to understand intuitively. For this reason, we usually calculate summaries of the posterior distribution. The main computational tool for doing this is Markov Chain Monte Carlo.

1.5 Markov Chain Monte Carlo

Monte Carlo methods allow us to calculate any property of a probability distribution that is an expectation value. For example, if we have a single variable x with a probability density $p(x)$, the expected value is

$$\mathbb{E}(x) = \int_{-\infty}^{\infty} xp(x)\,dx \quad (1.15)$$

which is a measure of the "center of mass" of the probability distribution.

If we had a set of points $\{x_1, x_2, ..., x_N\}$ "sampled from" $f(x)$, we could replace the integral with a simple average:

$$\mathbb{E}(x) \approx \frac{1}{N}\sum_{i=1}^{N} x_i. \quad (1.16)$$

In one dimension, this may not seem very useful. Evaluating a one dimensional integral analytically is often possible, and doing it numerically using the trapezoidal rule (or a similar approximation) is quite straightforward. However, Monte Carlo really becomes useful in higher dimensional problems. For example, consider a problem with five unknown quantities with probability distribution $p(a, b, c, d, e)$, and suppose we want to know the probability that a is greater than $b + c$. We could do the integral

$$P(a > b + c) = \int p(a, b, c, d, e)\mathbb{1}\left(a > b + c\right)\, da\, db\, dc\, dd\, de \quad (1.17)$$

where $\mathbb{1}\left(a > b + c\right)$ is a function that is equal to one where the condition is satisfied and zero where it isn't. However, if we could obtain a sample of points in the five dimensional space, the Monte Carlo estimate of the probability is simply

$$P(a > b + c) \approx \frac{1}{N}\sum_{i=1}^{N} \mathbb{1}\left(a_i > b_i + c_i\right) \quad (1.18)$$

which is just the fraction of the samples that satisfy the condition.

Another important use of Monte Carlo is marginalisation. Suppose again we had a probability distribution for five variables, but we only cared about one of them. For example, the marginal distribution of a is given by

$$p(a) = \int p(a, b, c, d, e)\, db\, dc\, dd\, de \quad (1.19)$$

which describes your uncertainty about a, rather than your uncertainty about all of the variables. This integral might be analytically intractable. With Monte Carlo, if you have samples in the five dimensional space but you only look at the first coordinate, then you have samples from $p(a)$. This is demonstrated graphically in Figure 1.3.

In Bayesian inference, the most important probability distribution is the posterior distribution for the parameters. We would like to be able to generate samples from the posterior, so we can compute probabilities, expectations, and other summaries easily. Markov Chain Monte Carlo allows us to generate these samples.

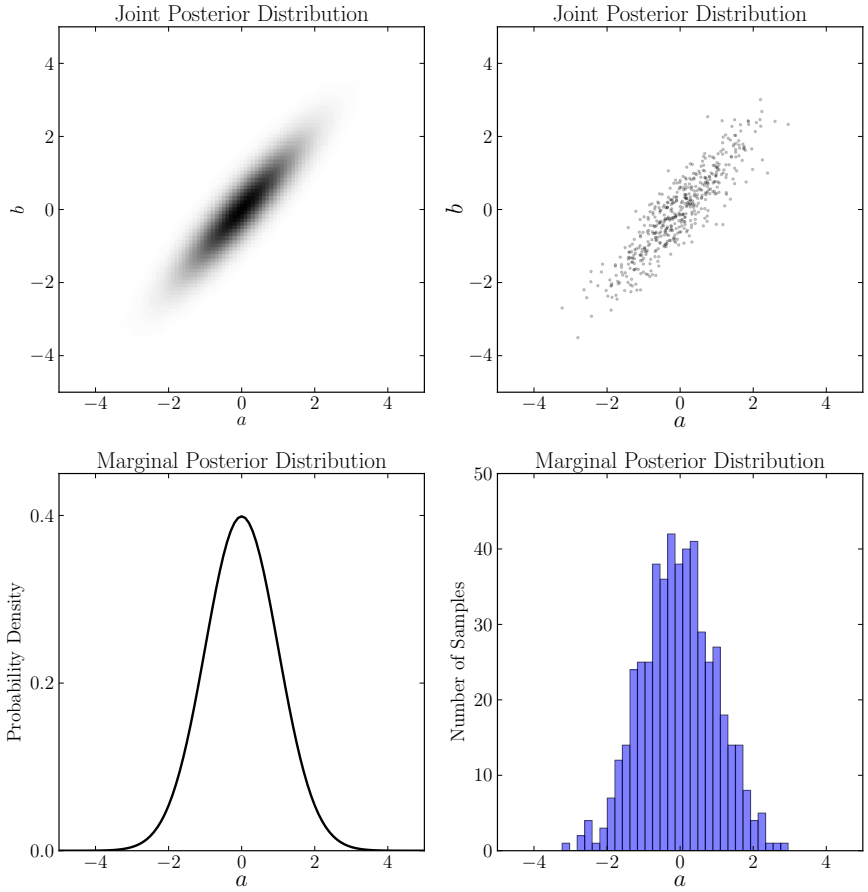

Fig. 1.3. *An example posterior distribution for two parameters a and b, taken from my STATS 331 undergraduate lecture notes. The full joint distribution is shown in the top left, and the marginal distribution for a (bottom left) is calculating by integrating over all possible b values, a potentially non-trivial calculation. The top right panel has points drawn from the joint posterior. Points drawn from the marginal posterior (bottom right) are obtained by ignoring the b values of the points, a trivial operation.*

1.5.1 The Metropolis Algorithm

The Metropolis-Hastings algorithm, also known as the Metropolis algorithm, is the oldest and most fundamental MCMC algorithm. It is quite straightforward to implement, and works well on many problems. The version of the Metropolis algorithm presented here is sometimes called *random walk* Metropolis. More sophisticated choices are possible, but usually require problem-specific knowledge.

Consider a problem with unknown parameters θ. If the prior is some density $\pi(\theta)$ and the likelihood function is $L(\theta)$, then the posterior distribution will be proportional to $\pi(\theta)L(\theta)$. The marginal likelihood $Z = \int \pi(\theta)L(\theta)\,d\theta$ is unknown, but the Metropolis algorithm doesn't need to know it: all we need is the ability to evaluate π and L at a given position in the parameter space. The Metropolis algorithm tells us how to move a "particle" around the parameter space so that we eventually sample the posterior distribution. That is, the amount of time spent in any particular region of parameter space will be approximately proportional to the posterior probability in that region. Note the change of notation from $p(\theta)$, $p(D|\theta)$, and $p(D)$ to π, L, and Z respectively. This is a convention when discussing computational methods (as opposed to discussing priors, datasets, etc).

The Metropolis algorithm can be summarised as follows:

(i) Choose a starting position θ, somewhere in the parameter space.

(ii) Generate a proposed position θ' from a proposal distribution $q(\theta'|\theta)$. A common choice is a "random walk" proposal, where a small perturbation is added to the current position.

(iii) With probability $\alpha = \min\left(1, \frac{q(\theta|\theta')}{q(\theta'|\theta)}\frac{\pi(\theta')}{\pi(\theta)}\frac{L(\theta')}{L(\theta)}\right)$, accept the proposal (i.e. replace θ with θ'). Otherwise, do nothing (i.e. remain at θ).

(iv) Repeat steps (i)–(iii) until you have enough samples.

When a proposed move is rejected (and the particle remains in the same place), it is important to count the particle's position again in the output. This is how the algorithm ends up spending more time in regions of high probability: moves *into* those regions tend to be accepted, whereas moves *out* of those regions are often rejected. The Metropolis algorithm is quite straightforward to implement and I encourage you to attempt this yourself if you haven't done so before.

Python code implementing the Metropolis algorithm is given below (this code has been stripped of "book-keeping" features for keeping track of the output, and shows just the algorithm itself). There are several features of note. Firstly, the functions used to measure the prior density and likelihood of any point, and the function to generate a proposal in the first place, are problem-specific and assumed to have been implemented elsewhere. Secondly, for numerical reasons we deal with the (natural) log of the prior density, the likelihood, and the acceptance probability. Thirdly, note how the `log_prior` and `log_likelihood` functions only need to be called once per iteration, not twice as one might naively think. Finally, no q ratio is required in the acceptance probability: we will assume that we are working

with a *symmetric* proposal distribution, where the probability of proposing a move to position a given the current position is b is the same as the probability of the reverse (proposing b when at position a).

```
# Generate a starting point (if you have a good guess, use it)
# In the full version of the code, the initial point is drawn
# from the prior.
params = np.array([1., 1., 1., 1.])
logp, logl = log_prior(params), log_likelihood(params)

# Total number of iterations
steps = 100000

# Main loop
for i in range(0, steps):
  # Generate proposal
  new = proposal(params)

  # Evaluate prior and likelihood for the proposal
  logp_new = log_prior(new)
  logl_new = -np.Inf
  # Only evaluate likelihood if prior prob isn't zero
  if logp_new != -np.Inf:
    logl_new = log_likelihood(new)

  # Acceptance probability
  log_alpha = (logl_new - logl) + (logp_new - logp)
  if log_alpha > 0.:
    log_alpha = 0.

  # Accept?
  if rng.rand() <= np.exp(log_alpha):
    params = new
    logp = logp_new
    logl = logl_new
```

The "random walk" proposal generates a proposed value θ' from a normal (gaussian) distribution centered around the current position θ. The user is free to choose the width of the normal distribution. Here is a Python code snippet showing a proposal with width `L`:

```
# Generate a proposal
proposal = theta + L*rng.randn()
```

The performance of the Metropolis algorithm depends quite strongly on the width of the proposal distribution. If the width is too small, most moves will be accepted, but won't move very far. If the width is too large, most

moves will be rejected, so you'll end up stuck in one place. Some authors recommend using preliminary runs to find an optimal width. Instead, I recommend that you use a mixture of widths. Basically, every time we make a proposal, the width is drawn from some range, rather than being constant. The biggest possible width we would ever want should be roughly the order of magnitude of the width of the prior (since the posterior is usually narrower than the prior). It's rare that we would need proposals many orders of magnitude smaller than that. My default suggestion is to randomise the logarithm of the step size, as in this code snippet:

```
# A heavy-tailed proposal distribution
# Generate a standard deviation
L = 10.**(1.5 - 6.*rng.rand())

# Use the standard deviation for the proposal
proposal = theta + L*rng.randn()
```

With this proposal, the minimum width is $10^{-4.5} \approx 3.16 \times 10^{-5}$, and the maximum width is $10^{1.5} \approx 31.6$. The effective proposal distribution is now very heavy-tailed. As long as a good width is somewhere within our range, things should be okay. Remember, this is not an optimal suggestion, but a fail-safe conservative one. It is possible to spend time (doing preliminary runs) to eventually save time (by having a more efficient final run), but my personal preference is usually not to do this.

When there are multiple parameters (almost always, since MCMC isn't necessary on single parameter problems!), we need to decide how to construct the proposal. There are two main ways to do this. The first is that the proposal is to change all of the parameters simultaneously. This tends to be inefficient in high dimensions, because the only proposals that are likely to be accepted are those that change the parameter values only slightly: in a high dimensional space, there are many bad directions to travel, and not very many good ones. Usually, it's better to propose to change a subset of the parameters, or even just a single parameter. A Python function that takes a `numpy` array of parameters and input and returns a proposed value for the parameters is specified below.

```
def proposal(params):
  """
  Generate new values for the parameters.
  The proposal for the Metropolis algorithm.
  """
  # Copy the parameters
  new = copy.deepcopy(params)
```

```
  # Which one should we change?
  which = rng.randint(num_params)
  new[which] += jump_sizes[which] \
                *10.**(1.5 - 6.*rng.rand())*rng.randn()
  return new
```

This function relies on `num_params` being the number of parameters, and an array `jump_sizes`, of length `num_params`, which specifies the prior width for each parameter.

For numerical reasons, instead of writing a function to evaluate the likelihood (and the prior density) at a particular point in parameter space, we usually deal with the logarithms of these quantities. For the transit example, the log prior function can be implemented like so:

```
def log_prior(params):
  """
  Evaluate the (log of the) prior distribution
  """
  A, b, tc, width = params[0], params[1], params[2], params[3]

  # Minus infinity, if out of bounds
  if A < -100. or A > 100.:
    return -np.Inf
  if b < 0. or b > 10.:
    return -np.Inf
  if tc < t_min or tc > t_max:
    return -np.Inf
  if width < 0. or width > t_range:
    return -np.Inf

  return 0.
```

Since we chose uniform priors, the prior density is some constant if the parameters (here passed to the function as a numpy `array` of four floating point values) are within the bounds of the uniform priors. Otherwise, the prior density is zero. We do not need to know the normalising constant of the prior density: we returned zero for the log-density, but the progress of the Metropolis algorithm would be the same if we had returned any other finite value, since the Metropolis algorithm only ever uses ratios of densities (i.e. differences in log-density).

The log likelihood function is given below. This implements the logarithm of Equation 1.5. As with the log prior function, we could ignore the nor-

malisation constants – in this case any term that is not a function of the parameters. However, I have included them for completeness.

```
def log_likelihood(params):
  """
  Evaluate the (log of the) likelihood function
  """
  # Rename the parameters
  A, b, tc, width = params[0], params[1], params[2], params[3]

  # First calculate the expected signal
  mu = A*np.ones(N)
  mu[np.abs(data[:,0] - tc) < 0.5*width] = A - b

  # Normal/gaussian distribution
  return -0.5*N*np.log(2.*np.pi) - np.sum(np.log(data[:,2])) \
      -0.5*np.sum((data[:,1] - mu)**2/data[:,2]**2)
```

1.5.2 Useful Plots

After running the Metropolis algorithm (or any other MCMC method), there are several useful plots that you should make. The first is known as a "trace plot" (Figure 1.4), and is just a plot of one parameter over time as the algorithm run. Trace plots are the single most useful diagnostic of whether your algorithm is working well. In the attached code, the MCMC results are stored in a two dimensional `numpy` array called `keep`, each row of which is the parameter vector at a particular iteration. It is trivial to make a trace plot from this output:

```
# Trace plot of parameter zero
plt.plot(keep[:,0])
```

The result is shown in Figure 1.4. When everything is working well, a trace plot should look like white noise when zoomed out. If this is the case, then things are *probably* working well (although you can never be 100% certain of this – like optimisation methods, MCMC methods can get stuck in local maxima). In addition, if your MCMC run was initialised at a point in parameter space that is an "outlier" with respect to the posterior distribution, the first part of the run will be a transient feature where the MCMC chain (hopefully) moves towards the important regions of the space. This transient period is called the burn-in, and should usually be excluded from the sample, otherwise your Monte Carlo summaries will give too much importance to the part of the space the MCMC happened to go through during

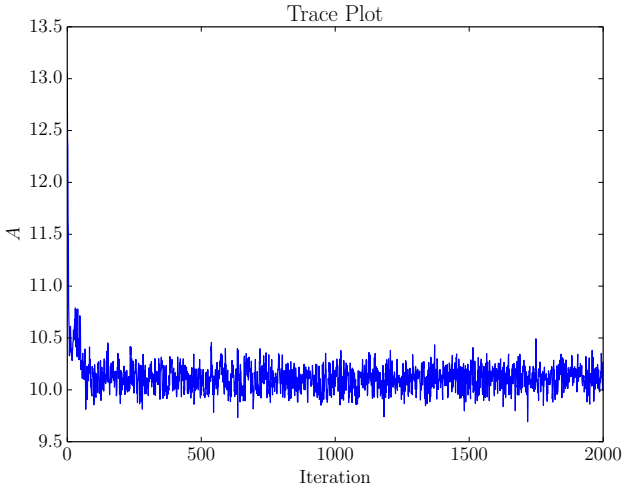

Fig. 1.4. *A "trace plot". At the beginning, because of the initial conditions of the algorithm, the results start in a region of parameter space that has very low probability. This initial phase is often called the "burn-in", and should be excluded from any subsequent calculations.*

burn-in. When using Metropolis, it's worthwhile to monitor the fraction of the proposed moves that are accepted. This fraction shouldn't be too close to zero or one, and generally somewhere between 15 and 50% is usually advised.

As an exercise, try running the Metropolis algorithm on the transit problem with a proposal that is far too small, a proposal that is far too large, and with an initial condition that is very far from the bulk of the posterior distribution. Look at the resulting trace plots and compare them to the healthy one in Figure 1.4.

Another useful type of plot is a scatter plot showing the joint posterior distribution for a pair of parameters (with all of the other parameters marginalised out). See Figure 1.6 for an example. This allows us to visualise some of the dependences in the remaining uncertainty about the parameters. For example, there is a slight correlation between A and b in the posterior, so if we obtained further information about A from elsewhere, this would affect our knowledge of b. In models with 2-10 interesting parameters, it is common to plot a grid of such plots, showing each parameter vs each other parameter. These are sometimes called 'corner' or 'triangle' plots, and a convenient Python package for using them is `triangle.py` by Dan Foreman-Mackey†.

† Available at `https://github.com/dfm/triangle.py`

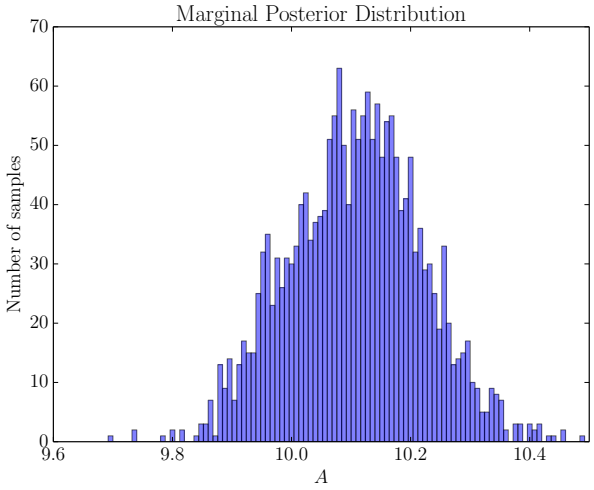

Fig. 1.5. *The marginal posterior distribution for the parameter A constructed from MCMC output.*

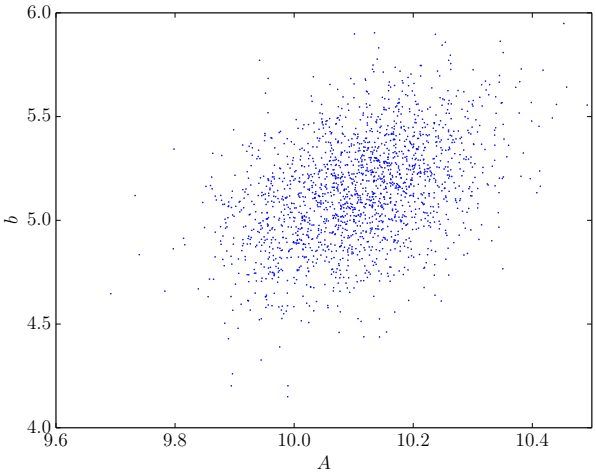

Fig. 1.6. *The joint posterior distribution for A in b constructed from the MCMC output. This is simply a scatterplot. Some authors prefer to apply some smoothing and approximate density contours.*

1.5.3 Posterior Summaries

A probability distribution, such as the posterior, can potentially be arbitrarily complicated. For communication purposes, it is usually easier to summarise the distribution by a few numbers. The most common summaries are point estimates (i.e. a single estimate for the value of the parameter,

such as "we estimate $\theta = 0.43$") and intervals (e.g. the probability that $\theta \in [0.3, 0.5]$ is 68%).

Thankfully, most of these summaries are trivial to calculate from Monte Carlo samples, such as those obtained from MCMC. The most popular point estimate is the posterior mean, which is approximated by the arithmetic mean of the samples. The posterior median can also be easily approximated by the arithmetic median of the samples.

```
post_mean = np.mean(keep[:,0])
post_median = np.median(keep[:,0])
```

There are some theoretical arguments, based on decision theory, that provide guidance about which point estimate is better under which circumstances. To compute a credible interval (the Bayesian version of a confidence interval), you find quantiles of the distribution. For example, if we want to find an interval that contains 68% of the posterior probability, the lower end of the interval is the parameter value for which 16% of the samples are lower. Similarly, the upper end of the interval is the value for which 16% of the samples are higher (i.e. 84% are lower). The simplest way to implement this calculation is by sorting the samples, as shown below.

```
sorted_samples = np.sort(samples)
# Left and right end of interval
left = sorted_samples[int(0.16*len(sorted_samples))]
right = sorted_samples[int(0.84*len(sorted_samples))]
```

This kind of interval is sometimes called a *centered* credible interval, because the same amount of probability lies outside the interval on each side. In astronomy, the 68% credible interval is very popular because it is equivalent to "plus or minus one posterior standard deviation" if the posterior is gaussian. In other statistical fields such as opinion polling, psychology, and medical science, 95% credible intervals are more conventional.

1.6 Assigning Prior Distributions

There are only really two open problems in Bayesian inference. The first is how to assign sensible prior distributions (and sampling distributions) in different circumstances, and the second is how to calculate the results efficiently. With regard to the first problem ,it is sometimes said that there are two different kinds of Bayesian inference, *subjective* and *objective*, with different methods for choosing priors. My view is that Bayesian describes a

hypothetical state of prior knowledge, held by an idealised reasoner. When we apply Bayesian inference, we are studying how the idealised reasoner would update their state of knowledge based on the information we explicitly put into the calculation.

In many cases, we can just use simple "default" choices for the prior and the sampling distribution (e.g. a uniform prior for a parameter, and a "gaussian noise" assumption for the data). A lot of analyses work just fine with these assumptions, and taking more care wouldn't change the results in any important way. However, occasionally it makes sense to spend a lot of time and effort thinking about the prior distributions. So-called subjective Bayesians, who are not necessarily experts in the fields of their clients, conduct elaborate interviews with experts to try and create a prior that models the experts' beliefs well. This process is called *elicitation*. For example, consider the Intergovernmental Panel on Climate Change (IPCC), who periodically write immense reports summarising humanity's state of knowledge about global warming. Consider the question "*How much will the global average temperature increase in the next 100 years?*". This is exactly the kind of situation where elicitation of an expert's probabilities is very important: a lot is known, and a lot is at stake. In this situation, it wouldn't be very wise to rely on convenient "vague" priors!

On the other hand, so-called "objective" Bayesians search for principles based on symmetry or other arguments, which can help choose a prior that is an appropriate choice to describe a large amount of ignorance. Some examples include the principle of indifference, the Jeffreys prior, reference priors, default priors, transformation groups, maximum entropy, and entropic priors.

1.6.1 Probability distributions have consequences

When you assign prior distributions, it is more easy than you might expect to build in an assumption that you don't really agree with, which can end up affecting your results in ways you may not have predicted (but which are ultimately understandable). This is especially true in high dimensional problems, which is why "hierarchical models" (beyond the scope of this chapter) are so useful. Here, we'll look at some common issues that can arise in non-hierarchical models.

Imagine that you wanted to infer the mass M of a galaxy. It might seem reasonable to assume "prior ignorance" about M, using a uniform

distribution between 10^5 and 10^{15} solar masses:

$$M \sim U(10^5, 10^{15}) \tag{1.20}$$

However, this has an unfortunate feature: it implies that the prior probability of M being greater than 10^{14} is 0.9, and the probability of M being greater than 10^{12} is 0.999, which seems overly confident, when we were trying to describe ignorance! In astronomy, we are often in the situation of having to "put our error bars in the exponent". A prior that has this property is the "log-uniform" distribution (named by analogy with the lognormal distribution, and sometimes incorrectly called a Jeffreys prior), which assigns a uniform distribution to the logarithm of the parameter. If we replace the uniform prior by

$$\log(M) \sim U(\log(10^5), \log(10^{15})) \tag{1.21}$$

then the prior probabilities are more moderate: $P(M > 10^{14}) = 0.1$, $P(M > 10^{12}) = 0.3$, and so on. The loguniform prior is appropriate for positive parameters whose uncertainty spans multiple orders of magnitude. The probability density, in terms of M, is proportional to $1/M$.

There are two main ways to implement the log-uniform prior in the Metropolis algorithm. One is to keep M as a parameter, and take the non-uniform prior into account in the acceptance probability (i.e. implement the $1/M$ prior in your `log_prior` function). The other is to treat $\ell = \log(M)$ as the parameter, in which case the prior is still uniform. You'll just need to compute M from ℓ before you can use it in the likelihood function. The second approach (parameterising by ℓ instead) is generally a better idea.

Let's now discuss some consequences of the sampling distribution, for which we used a normal distribution with known standard deviation for each data point, and asserted that the measurements were independent. In many applications (such as discrete-valued photon count data) other distributions such as the Poisson may be more appropriate. However, even for real-valued "model plus noise" situations the normal distribution has some consequences which may be undesirable. For example, there is a high probability that the noise vector (i.e. all of the actual the differences between the true curve and the data points) looks macroscopically like white noise, with little correlation between the data points. To see this, try generating simulated datasets from the sampling distribution for a particular setting of the parameters, and you will see that almost all datasets that you generate have this property. If this is not realistic, correlated noise models are possible (e.g. using gaussian processes), but we will not discuss these here.

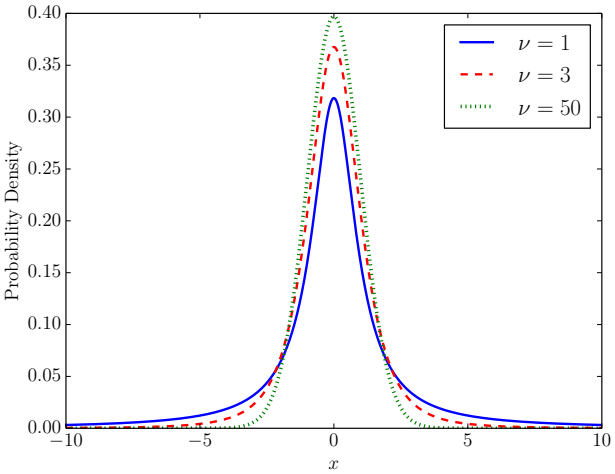

Fig. 1.7. *Three t-distributions, with $\nu = 1$, $\nu = 3$, and $\nu = 50$. Lower ν implies heavier tails and a greater probability for outliers. All three of these distributions have $\mu = 0$ and $\sigma = 1$.*

Another implication of our sampling distribution is that we would not expect very many measurements to depart from the model by more than a few standard deviations. Normal distributions have very "light tails", implying a very low probability for outliers. If the dataset does in fact contain outliers, the model will try to fit them very closely because the normal distribution assumption is telling it to believe the data points. This problem also occurs in least squares fitting, and switching to Bayesian inference will only resolve the problem if we use something other than a normal distribution for the sampling distribution. See Hogg et al. (2010) for a detailed discussion of this issue. A simple alternative to the normal distribution that is more appropriate if outliers are possible is the "student-t" distribution. Like the normal distribution, this has a "location" parameter (the center of the distribution) and a "scale" parameter (the width), but there is a third parameter ν (often called the "degrees of freedom") which controls the shape of the distribution. When ν is high ($\gtrsim 30$) the shape is very close to gaussian, but when ν is low (0-10) it has much heavier tails. Three t-distributions with different values of ν are shown in Figure 1.7.

For a single variable x, the probability density function is:

$$p(x|\nu,\mu,\sigma) = \frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\Gamma\left(\frac{\nu}{2}\right)\sigma\sqrt{\pi\nu}}\left[1+\frac{1}{\nu}\frac{(x-\mu)^2}{\sigma^2}\right]^{-\frac{\nu+1}{2}} \tag{1.22}$$

where μ is the location parameter, σ is the scale parameter, and ν is the

degrees of freedom. Γ is the gamma function. This can be used as a drop-in replacement for the normal distribution in Equation 1.5.

Student-t distributions are most well known because they arise naturally as the solution to some analytical problems in statistics, but they are also useful for allowing the possibility of outliers (as we are doing here), and for specifying informative priors. Traditionally, normal priors are frequently used when we have a lot of prior knowledge about the value of a parameter. The heavier tails of the Student-t distribution can be a more fail-safe option in this situation.

1.6.2 Should we trust the error bars?

In astronomy many datasets are accompanied by error bars, which give us some idea of the accuracy of a measurement. Without knowing all the details of how the error bars were produced, we don't really know to what extent we should trust them, and hence whether it is a good idea to incorporate them literally into our sampling distribution like we did in Equation 1.5.

Instead of trusting the error bars, we can add an extra parameter which describes the degree to which we should trust the error bars. Let K be a constant by which we should multiply all the error bars before using them in the likelihood function. If $K = 1$, this corresponds to complete trust, and if $K > 1$ this means the error bars should have been bigger by a factor K. We will not allow $K < 1$. Let's use the following prior density for K:

$$p(K) = \frac{1}{2}\delta(K-1) + \frac{1}{2}\begin{cases} \exp\left[-(K-1)\right], & K > 1 \\ 0, & K \leq 1 \end{cases} \tag{1.23}$$

This is a 50-50 *mixture* of a Dirac delta function at $K = 1$, and an exponential distribution (with scale length 1) for $K > 1$. This expresses the idea that there's a 50% chance that $K = 1$ precisely, and if not, then it's likely to be fairly close to 1, and very unlikely to be greater than 5. This prior is plotted in Figure 1.8.

To implement this prior in MCMC, we can use a similar trick to the method of implementing the log-uniform prior. That is, we can introduce another parameter with a uniform prior (let's call it u_K), and then transform it to produce K. If we let $u_K \sim U(-1, 1)$, and let $K = 1$ if $u_K < 0$, then we will have the desired 50% probability for $K = 1$. To obtain the exponential distribution part of the prior, we can set $K = 1 - \log(1 - u_K)$ if $u_K > 0$. This is the *inverse transform sampling* method, where the inverse of the

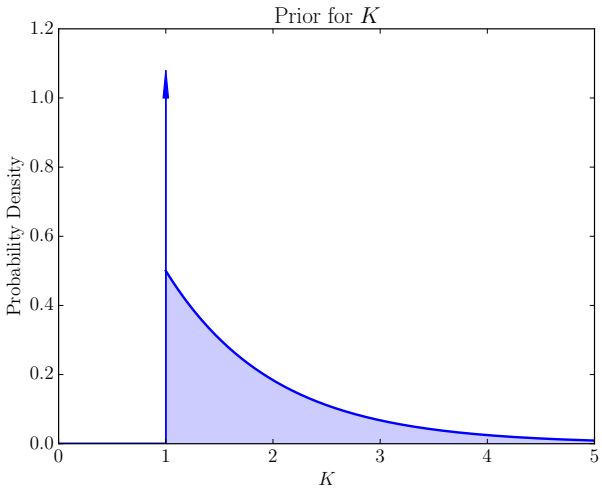

Fig. 1.8. *The prior for K, the amount by which we should scale the error bars given with the dataset. There is a 50% probability that $K = 1$, and a 50% probability that $K > 1$. Given that $K > 1$ the prior distribution is exponential (with unit scale length), so we do not expect K to be greater than 1 by an order of magnitude.*

cumulative distribution function† is used to transform a variable from a $U(0, 1)$ distribution to some other distribution.

We can now go ahead and implement the new `log_likelihood` function for the transit model with the Student-t distribution replacing the normal distribution. The code is given below, and demonstrates the technique used to implement both the log-uniform prior for ν (assuming `params[4]` has a uniform prior) and the mixture prior for K.

```
def log_likelihood(params):
  """
  Evaluate the (log of the) likelihood function
  """
  # Rename the parameters
  A, b, tc, width, log_nu, u_K = params[0], params[1], params[2]\
                                ,params[3], params[4], params[5]

  # Parameter is really log_nu
  nu = np.exp(log_nu)

  # Compute K and 'inflated' error bars
  if u_K < 0.:
    K = 1.
```

† If a variable has probability density $f(x)$ then the cumulative distribution function is $F(x) = \int_{-\infty}^{x} f(t)\, dt$.

```
else:
    K = 1. - np.log(1. - u_K)
sig = K*data[:,2]

# First calculate the expected signal
mu = A*np.ones(N)
mu[np.abs(data[:,0] - tc) < 0.5*width] = A - b

# Student t distribution
return N*scipy.special.gammaln(0.5*(nu+1.))\
     - N*scipy.special.gammaln(0.5*nu)\
     - np.sum(np.log(sig*np.sqrt(np.pi*nu)))\
     - 0.5*(nu + 1.)*np.sum(\
       np.log(1. + (data[:,1] - mu)**2/nu/sig**2))
```

With this modified likelihood function, a log-uniform prior for ν between 0.1 and 100, and a uniform prior for u_K between -1 and 1, we can calculate the consequences of this model for the transit example. The two extra parameters ν and K might be of interest, but unfortunately are not directly accessible since we elected to use $\log(\nu)$ and u_K as parameters instead, and derived ν and K from them. However, since these both had uniform priors, their marginal posteriors should be straightforward to interpret. These are plotted in Figure 1.9. The data has basically ruled out low values of $\log(\nu)$ as the posterior distribution favours high values. The student-t distribution becomes approximately gaussian when ν is high, so this result suggests the original gaussian assumption was fine (even though the original gaussian assumption is not technically even in this hypothesis space). For u_K, the data has ruled out values $\gtrsim 0.2$, implying that K is unlikely to be greater than about 1.2, so there is no real evidence that the error bars needed to be inflated.

Unlike with ν, with K our original assumption (equivalent to $K = 1$) is a part of the parameter space, so we can compute its posterior probability. Its prior probability was 50%, and its posterior probability can be estimated using the fraction of MCMC samples for which $u_K < 0$:

```
np.mean(keep[:,5] < 0.)
```

The result is 92%, i.e. given this data and these assumptions, we should be quite confident that the error bars are of the appropriate size. This should be of no surprise, since I in fact generated the data using a normal distribution, and the error bars in the data file were the same as the standard deviation of the normal distribution I used.

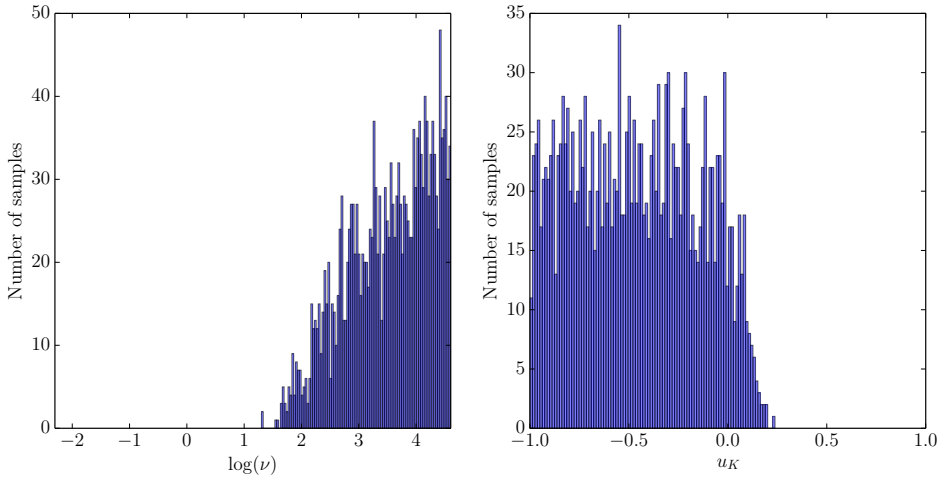

Fig. 1.9. Marginal posterior distributions for $\log(\nu)$, which describes the shape of the Student-t distribution for the noise, and u_K, which determines the error bar inflation factor K.

1.7 When the results are sensitive to the prior

The kind of prior we used for K (Figure 1.8), with a spike at an especially-plausible parameter value, is notorious for producing conclusions that depend sensitively on the shape of the prior. To see this for yourself, try keeping the prior probability for $K = 1$ constant at 50%, but make the scale length of the exponential part of the prior larger. You will find that the posterior probability for $K = 1$ decreases when you do this.

In the limit that the exponential scale length tends to infinity, the posterior probability for $K = 1$ will tend to zero, regardless of the dataset. This is not a mystery, but a logical consequence of the assumptions used. For example, if the scale length of the exponential were 1,000,000, then if $K > 1$ we would expect to see data scattered by an amount much greater than the error bars suggest. Since this was not observed, the data will favour $K = 1$. On the other hand, if the scale length of the exponential were 0.001, we would expect to see basically the same data whether $K = 1$ or not. In this situation, we'd obtain a posterior probability very close to 50% (the same as the prior) for $K = 1$, unless we had a huge amount of data.

When the results of an analysis are very sensitive to the prior information assumed in the prior distribution and the sampling distribution, it is prudent to demonstrate this sensitivity to your readers. The problem may go away with more careful consideration of your priors, or it may just signal that the data cannot answer your question in a way that will satisfy everybody.

1.8 What is the data?

It can sometimes be helpful to think carefully about exactly what your dataset contains, and whether it is all "data" in the sense of Bayesian inference. In the transit example, we chose a sampling distribution for the data, but this sampling distribution was for the y-values of the data (the flux measurements). Note that we never assigned a probability distribution for the *times* of the data points, even though we might think of that as part of the data (it's probably in the same file as the measurements, after all). In fact, because we didn't assign a probability distribution for it, the timestamps were not data at all, but part of the prior information I. Therefore, it is completely justifiable to use the timestamps when assigning the prior and the sampling distribution. It is not "cheating", even slightly, to use the time information as we did to set the priors.

However, if we had used the y-values in the dataset to help choose the prior (perhaps for A), this would have been technically incorrect. Nevertheless, it is common practice to look at the data before assigning priors, and the real test of the legitimacy of this practice is whether it makes any difference to your results. In the transit example, if we had used a $U(-50, 50)$ prior instead of a $U(-100, 100)$ one, the posterior distribution would have been virtually the same. The only substantial difference would have been in the value of the marginal likelihood, had we calculated it (see Section 1.12 for more information about when this is important).

1.9 Model Selection and Nested Sampling

Nested Sampling (NS) is a Monte Carlo algorithm introduced by Skilling (2006). It is not technically an MCMC algorithm, although MCMC can be used as part of the implementation. It has some advantages over related Monte Carlo methods such as parallel tempering (Hansmann, 1997; Gregory, 2005; Vousden et al., 2015), but also brings with it its own challenges. Several variants of NS exist, and most of them are somewhat complicated. Here, I will present a simple version of the algorithm which is sufficient to solve a wide range of problems. This approach is similar to the one presented in the introductory textbook by Sivia and Skilling (2006).

Many more complex and sophisticated versions of NS exist, such as the popular MultiNest (Feroz, Hobson, & Bridges, 2009), my own Diffusive Nested Sampling (Brewer, Pártay, & Csányi, 2011), and several others. These algorithms, while all based on the insights of Skilling (2006), are very different in detail. See the appendix for more details.

Consider two different models M_1 and M_2 which are mutually exclusive

(they can't both be true). Suppose M_1 has parameters θ_1, and M_2 has its own parameters θ_2. The methods described previously can be used to calculate the posterior distribution for M_1's parameters:

$$p(\theta_1|D, M_1) = \frac{p(\theta_1|M_1)p(D|\theta_1, M_1)}{p(D|M_1)} \tag{1.24}$$

You can also fit model M_2 to the data, i.e. get the posterior distribution for M_2's parameters:

$$p(\theta_2|D, M_2) = \frac{p(\theta_1|M_2)p(D|\theta_2, M_2)}{p(D|M_2)} \tag{1.25}$$

This is all very well, but you might want to know whether M_1 or M_2 is more plausible overall, given your data. That is, you want the posterior probabilities $P(M_1|D)$ and $P(M_2|D)$.

In this situation, it's usually easier to calculate the ratio of the two posterior probabilities, which is sometimes called the *posterior odds ratio*:

$$\frac{P(M_2|D)}{P(M_1|D)} = \frac{P(M_2)}{P(M_1)} \times \frac{P(D|M_2)}{P(D|M_1)} \tag{1.26}$$

As you might expect, the posterior odds for M_2 over M_1 depends on the prior odds: was M_2 more plausible than M_1 before taking into account the data? The other ratio is a ratio of likelihoods, sometimes called a Bayes Factor; how probable was the data assuming M_2 vs assuming M_1? These likelihoods aren't the likelihoods for a specific value of the parameters, but instead likelihoods for the model as a whole. To distinguish this kind of likelihood from the standard kind (which is a function of the model parameters), the term *marginal likelihood* or *evidence* is used. The marginal likelihood for M_1 is:

$$p(D|M_1) = \int p(\theta_1|M_1)p(D|\theta_1, M_1)\, d\theta_1 \tag{1.27}$$

which you may recognise as the normalising constant in the denominator of Bayes' theorem in the context of getting the posterior for θ_1. To do model selection, we need this as well as the marginal likelihood for M_2:

$$p(D|M_2) = \int p(\theta_2|M_2)p(D|\theta_2, M_2)\, d\theta_2 \tag{1.28}$$

Marginal likelihoods are integrals over the parameter space. They are expected values, but they are not expected values with respect to the posterior distribution, but rather the prior. Therefore, we cannot use standard

MCMC methods (at least in any simple way) to calculate the marginal likelihood†.

A Monte Carlo approach that samples from the prior distribution will also fail in most cases. The likelihood function $p(D|\theta_1, M_1)$ in Equation 1.27 is usually sharply peaked in a very small region of parameter space. The integral will be dominated by the high values of the likelihood in the tiny region, yet a Monte Carlo approach based on sampling the prior will almost certainly not give any samples in the important region.

1.10 An easy problem

Imagine a parameter estimation problem with a single parameter X with a uniform prior distribution between 0 and 1, and a likelihood function which is a decreasing function of X (like in Figure 1.10). In this situation the marginal likelihood would be a straightforward integral $Z = \int_0^1 L(X)\, dX$. If we could obtain some points X, and measure their likelihoods L, we could approximate the integral numerically, using the trapezoidal rule or some other numerical quadrature method. The key insight of NS is that any high dimensional Bayesian problem can be mapped into a one dimensional problem like that shown in Figure 1.10. In this introductory discussion, I will ignore some mathematical technicalities and focus more on the ideas behind the algorithm.

1.11 Making hard problems easy

Consider a function that takes the entire parameter space and maps it to the real line between 0 and 1, such that the best fit (highest likelihood) point in the parameter space is mapped to $X = 0$, and the worst point in the space is mapped to $X = 1$. For all of the intermediate points, the X value is given by:

$$X(\theta) = \int \pi(\theta) \mathbb{1}\left(L(\theta) > L(\theta')\right)\, d\theta' \tag{1.29}$$

Since X only depends on θ through the likelihood function, it can also be considered a function of a likelihood value:

$$X(\ell) = \int \pi(\theta) \mathbb{1}\left(L(\theta) > \ell\right)\, d\theta \tag{1.30}$$

† There is a method, called the "harmonic mean estimator", for estimating the marginal likelihood from posterior samples. Bayesian statistician Radford Neal has described it as the "worst Monte Carlo method ever".

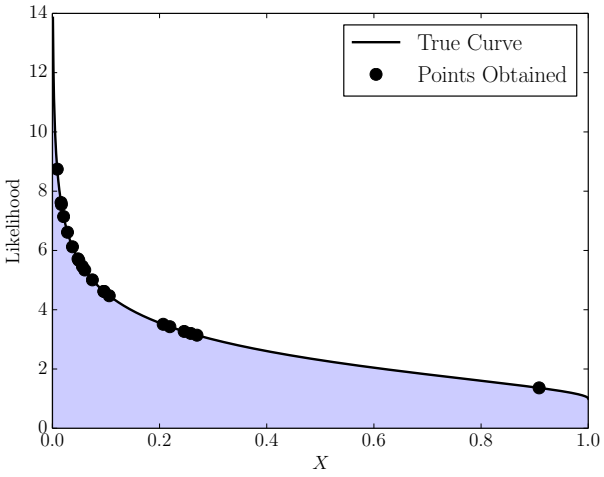

Fig. 1.10. *A simple one dimensional parameter estimation problem where the prior is uniform between 0 and 1, and the decreasing curve shown is the likelihood function (which has the same shape as the posterior). The marginal likelihood is the integral of the likelihood function, and we could calculate this numerically if we had a few points along the curve. Nested Sampling takes a high dimensional space and uses it to compute a curve like this whose integral is the marginal likelihood Z.*

An intuitive interpretation of X is the "fraction of points in the space which beat this point", that is, X is a rank. Importantly, the prior distribution $\pi(\theta)$ for the parameters implies a uniform distribution $U(0,1)$ for the ranks X. To see this, imagine measuring the height of all 900,000 residents of Tenerife, sorting them from tallest to shortest, and assigning a rank X to each. The tallest person would get a rank of 0, the shortest a rank of 1, and the person exactly in the middle would get a rank of 0.5. If you made a histogram of the ranks, it would be uniform since the range of X values (from 0 to 1) is divided evenly between the 900,000 people.

The Nested Sampling algorithm itself works by starting with a population of N "particles" (points in parameter space) drawn from the prior. In terms of X, these points will be uniformly distributed between 0 and 1. If we were to find the worst point (with the lowest likelihood, $L_{\rm worst}$), this would correspond to the highest X value. We could also guess that the highest X value will not be close to $X = 0$, but instead will be close to $X = 1$ (exactly how close depends on N). The simplest prescription given by Skilling (2006) is to estimate $X_{\rm worst} = \exp(-1/N)$ (more sophisticated possibilities exist but do not make much of a difference). We also know the likelihood of this point

(we must have measured it in order to identify the worst point!), and can put it on a graph like the one in Figure 1.10.

To obtain more points, we generate a new point to replace the one just identified as the worst. This point is drawn from the prior, but with the restriction that its likelihood must exceed L_{worst}. In terms of X, the new point's location has a uniform distribution between 0 and $\exp(-1/N)$: just the same as the other $N-1$ points. If we find the worst point now, we'll be doing the same thing but instead of looking at X values between 0 and 1 we are looking at X values between 0 and $\exp(-1/N)$. Therefore our estimate of the X value of the worst particle in the second iteration is $\exp(-1/N) \times \exp(-1/N) = \exp(-2/N)$. Since we know the likelihood, we can add another point to our graph and continue. The NS algorithm is summarised below.

(i) Generate N particles $\{\theta_1, \theta_2, ..., \theta_N\}$ from the prior, and calculate their likelihoods. Initialise a loop counter i at 1.
(ii) Find the particle with the lowest likelihood (call this likelihood L_{worst}). Estimate its X value as $\exp(-i/N)$. Save its properties (X_i, L_i), and its corresponding parameter values as well.
(iii) Generate a new particle to replace the one found in step (ii). The new particle should be drawn from the prior distribution, but its likelihood must be greater than L_{worst}.
(iv) Repeat steps (ii) and (iii) until enough iterations have been performed.

In step (ii), we assume that there is a unique "worst" particle with the lowest likelihood. If you are working on a problem where likelihood ties are possible, you need to add an extra parameter to your model whose sole purpose is breaking ties in step (ii). See Murray (2007) for more details.

Step (iii) also hides a lot of complexity, and is the key point distinguishing different implementations of Nested Sampling from each other. We have to be able to generate a point from a restricted version of the prior, proportional to $\pi(\theta)\mathbb{1}\left(L(\theta) > L_{\text{worst}}\right)$. Naive "rejection sampling", i.e. generating from the prior until you get a point that satisfies the likelihood constraint, will not work because the region satisfying the constraint may be very small in volume. This volume is, in fact, exactly what X measures, and X decreases exponentially during NS. A simple and popular approach, which we will use, is to copy one of the surviving particles (which has a likelihood above L_{worst} by construction) and evolve it using MCMC as though we were trying to sample the prior, but rejecting any proposed move that would take the likelihood below L_{worst}. The main disadvantage of this approach is that the

initial diversity of the N particles can become depleted quite quickly, leading to problems if there are multiple likelihood peaks.

In most problems, the region of the parameter space with high likelihood is quite small. Therefore, in practice, most of the marginal likelihood integral will be dominated by a very small range to the left of the plot in Figure 1.10. For this reason, logarithmic axes are more useful (Figure 1.11). In terms of the plots in Figure 1.11, NS steps towards the left, obtaining (X, L) pairs each time the worst point is found. In terms of $\log(X)$ the steps are of equal size $(1/N)$. Therefore, to cover a certain distance in terms of $\log(X)$, the number of iterations you need is proportional to N. As you might expect, higher N leads to more accurate results but takes more CPU time.

Once the algorithm has been running for a while, the marginal likelihood can be obtained by numerically approximating the integral:

$$Z = \int_0^1 L(X)\, dX. \tag{1.31}$$

An additional output of NS is the "information", also known as the Kullback-Leibler divergence of the posterior distribution from the prior distribution, which measures how compressed the posterior distribution is. Loosely speaking, the information is (minus the log of) the fraction of the prior volume occupied by the posterior. An information of zero means the posterior is the same as the prior, and an information of 100 means the posterior occupies about e^{-100} times the prior volume. The definition is:

$$H = \int \frac{\pi(\theta)L(\theta)}{Z} \log\left[\frac{L(\theta)}{Z}\right] d\theta \tag{1.32}$$

$$= \int_0^1 \frac{L(X)}{Z} \log\left[\frac{L(X)}{Z}\right] dX \tag{1.33}$$

and this can be computed numerically once Z has been calculated. The information also gives an estimate of the uncertainty in $\log(Z)$, given by $\sqrt{H/N}$ (Skilling, 2006).

As you might expect, NS can also be used to generate posterior samples; it's unlikely anyone would use it if it didn't. Each discarded point (the worst point at each iteration) is assigned a prior "width" based on the distance from its neighbours in terms of X, and a posterior weighting factor is given by the prior width times the likelihood. In the code provided with this chapter, a resampling technique is used to create equally-weighted posterior samples which can be used like standard MCMC output.

Python code (minus bookkeeping) implementing basic Nested Sampling is given below.

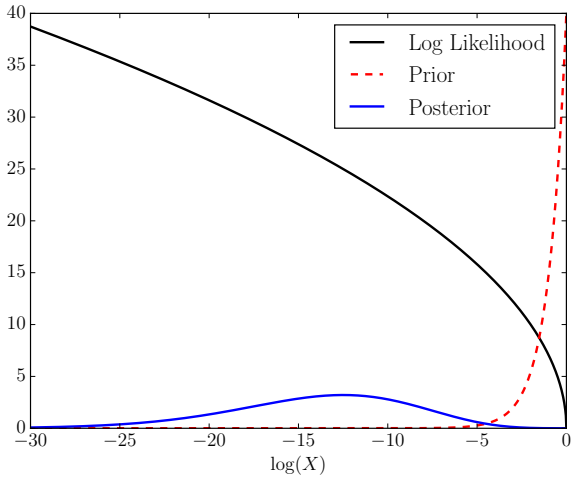

Fig. 1.11. *The same as Figure 1.10, but with logarithmic axes, since the important parts of parameter space tend to occupy a very small volume of parameter space. The black curve is the likelihood function, and the uniform prior for* X *corresponds to the dotted red exponential prior for* $\log(X)$*. The posterior (proportional to prior times likelihood) usually has a bell-shaped peak, but can be more complex.*

```
# Number of particles
N = 5

# Number of NS iterations
steps = 5*30

# MCMC steps per NS iteration
mcmc_steps = 10000

# Generate N particles from the prior
# and calculate their log likelihoods
particles = []
logp = np.empty(N)
logl = np.empty(N)
for i in range(0, N):
 x = from_prior()
 particles.append(x)
 logl[i] = log_likelihood(x)

# Main NS loop
for i in range(0, steps):
  # Find worst particle
  worst = np.nonzero(logl == logl.min())[0]
```

```
  # Save its details
  # ...

  # Copy survivor
  if N > 1:
    which = rng.randint(N)
    while which == worst:
      which = rng.randint(N)
    particles[worst] = copy.deepcopy(particles[which])

  # Likelihood threshold for new point
  threshold = copy.deepcopy(logl[worst])

  # Evolve within likelihood constraint using Metropolis
  for j in range(0, mcmc_steps):
    new = proposal(particles[worst])
    logp_new = log_prior(new)
    # Only evaluate likelihood if prior prob isn't zero
    logl_new = -np.Inf
    if logp_new != -np.Inf:
      logl_new = log_likelihood(new)
    loga = logp_new - logp[worst]
    if loga > 0.:
      loga = 0.

    # Accept
    if logl_new >= threshold and rng.rand() <= np.exp(loga):
      particles[worst] = new
      logp[worst] = logp_new
      logl[worst] = logl_new

  # Use the deterministic approximation
  # Estimate the X-values of the discarded worst points
  logX = -(np.arange(0, i+1) + 1.)/N
```

1.12 Nested Sampling on the transit example

To demonstrate NS, we can run it on the original version (Model 1) of the transit model, and the revised version with the t-distributed sampling distribution (Model 2), to calculate the two marginal likelihoods. I used $N = 5$ particles and 10,000 MCMC steps to generate a new particle in each iteration. For Model 1, I obtained $\log(Z_1) = -163.0 \pm 1.8$, and the graphical output is shown in Figure 1.12.

For Model 2, I got $\log(Z_2) = -165.0 \pm 1.9$. Thus, the data are approximately $e^2 \approx 10$ times as likely if the original model is true, compared to the modified model. At least, that's if we ignore the numerical error bars!

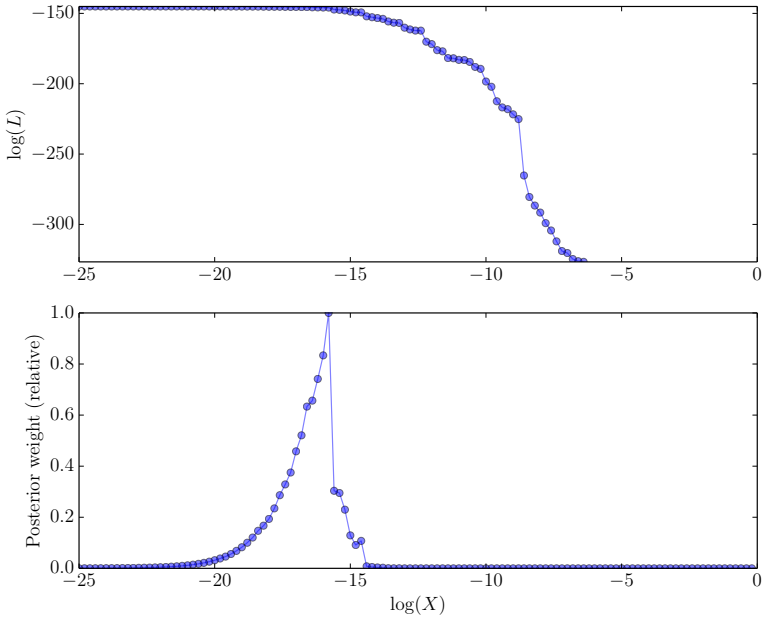

Fig. 1.12. *Numerical versions of the plots in Figure 1.11, produced using Nested Sampling on the transit problem with the original priors. For an NS run to be satisfactorily completed, the posterior weights in the lower panel must peak and decline, as seen here. There is little point continuing the run further unless you suspect another peak may appear. This can happen in "phase change" problems (Skilling, 2006).*

With 30 particles, I got $\log(Z_1) = -163.0 \pm 0.7$ for the original model and $\log(Z) = -165.8 \pm 0.8$ for the modified model, so this conclusion seems robust.

If the Bayes Factor (ratio of marginal likelihoods) favours Model 1 by a factor of 10, that doesn't necessarily imply it is 10 times more plausible. We must always remember the prior odds factor in Equation 1.26. It may be that we completely agree with the assumptions of Model 2, in which case the marginal likelihoods are irrelevant. Or we may consider the union of the two models to be sensible, but 50/50 prior probabilities to be inappropriate. Marginal likelihoods are not everything, but they are still important and worth calculating. As always, if a result is surprising, or depends on assumptions in a way you didn't expect, treat it as a learning opportunity.

1.13 Appendix: Notation for Probability Distributions

There are two main types of notation that are used in probability theory. The first type is mathematically more correct and conceptually clear, yet results in equations that are very large, complicated, and difficult to read. The second type is a very compact notation which makes equations a lot shorter and easier to read, but could be misinterpreted if you are not used to it. I prefer the compact notation, and it is more popular in the Bayesian literature. To prevent any misunderstandings I'll describe both types of notation here.

Suppose we're about to flip a coin N times, and are interested in the quantity X, the number of times the result is heads. According to the standard assumptions, the possible values for X are the integers from 0 to 10, and the probability distribution for X is given by a binomial distribution:

$$\begin{aligned} P(X = x) &= \left(\begin{array}{c} 10 \\ x \end{array} \right) \left(\frac{1}{2}\right)^x \left(\frac{1}{2}\right)^{10-x} \\ &= \left(\begin{array}{c} 10 \\ x \end{array} \right) \left(\frac{1}{2}\right)^{10} \end{aligned} \tag{1.34}$$

The equation gives us all the probabilities we want, i.e. $P(X = 0)$, $P(X = 1)$, and so on. The lower case x is a dummy variable: like the index in a sum, it can be replaced by any other symbol and the equation still holds. For example, replacing x with a gives:

$$P(X = a) = \left(\begin{array}{c} 10 \\ a \end{array} \right) \left(\frac{1}{2}\right)^{10} \tag{1.35}$$

which has exactly the same meaning as Equation 1.34.

The compact notation is an alternative to the left hand sides of Equations 1.34 and 1.35.

$$p(x) = \left(\begin{array}{c} 10 \\ x \end{array} \right) \left(\frac{1}{2}\right)^{10} \tag{1.36}$$

Since X, the actual number of heads, isn't written anywhere, we forget it exists and just use the lower case x for that purpose, even though it was originally a dummy variable! The key to understanding this notation is to read $p(x)$ as "the probability distribution for x", and to understand what the expression following it really means. If we have an expression involving $p(x)$ and $p(y)$, the two ps may not be the same function! If the set of possible x values is continuous (so $p(x)$ gives the probability *density* instead of a probability itself) then the concise notation doesn't change. Whether you're

dealing with a probability density function or a probability mass function (the discrete case) must be understood from the context. For completeness, the full (non-compact) notation for the probability density for a variable X is $f_X(x)$.

The compact notation is especially helpful in Bayesian statistics because we deal with conditional probability distributions a lot. For example, if we have three variables a, b, and c, the following is a true statement (it's an example of the *product rule* of probabilities):

$$p(a, b, c) = p(a)p(b|a)p(c|b, a). \tag{1.37}$$

Written in the non-compact notation, we'd have three variables A, B, and C, and the corresponding equation would be

$$\begin{aligned} P(A = a|B = b, C = c) &= P(A = a)P(B = b|A = a) \\ &\quad \times P(C = c|B = b, A = a) \end{aligned} \tag{1.38}$$

which is much harder to read and manipulate.

1.14 Appendix: A rough guide to popular Bayesian computation packages

This is a very brief list of popular Bayesian computation software packages that you might find useful. This is far from an exhaustive list; it only includes packages that I know something about. I apologise if your favourite package is not listed here.

MultiNest (Feroz, Hobson, & Bridges, 2009) is one of the most popular implementations of Nested Sampling. It does not use MCMC, like the version presented in this chapter. It is designed to handle potentially multimodal posterior distributions in low ($\lesssim 30$) dimensions. It is particularly useful in situations where the likelihood function is expensive to evaluate.

Emcee (Foreman-Mackey et al., 2012) is based on the affine-invariant ensamble sampling method introduced by (Goodman & Weare, 2010). It is useful for getting quick, efficient results on low dimensional ($\lesssim 30$) unimodal posterior distributions. It is written in Python, and one of its main advantages is that the user only needs to write a function that evaluates the log of the posterior density; the proposal distribution is automatically generated by the algorithm and requires no tuning.

JAGS (Plummer, 2003): The main advantage of JAGS is that it uses the BUGS language, a neat way of specifying your modelling assumptions using a language similar to the "$\sim$" notation. This makes JAGS very suitable for quickly implementing analyses of small to medium complexity without having to worry too much about MCMC algorithms themselves. JAGS can also be used easily from R (a programming language for statistics) through the `rjags` package, and as such is very popular among statisticians. For a very gentle introduction to JAGS you can consult my lecture notes at `www.github.com/eggplantbren/STATS331`.

Stan (Stan Development Team, 2014): The modelling language of Stan is similar to that of JAGS, although the underlying MCMC methods used are completely different. Stan is based on the No U-Turn Sampler (NUTS), a variant of the Hamiltonian MCMC method (Neal, 2011). This is a very efficient sampler in problems with a continuous parameter space and hierarchically defined priors. Like JAGS, Stan can also be used from within R, and interfaces also exist for Python, Julia, and MATLAB.

DNest (Brewer, Pártay, & Csányi, 2011): This is another version of Nested Sampling, but it is quite different from MultiNest. It uses MCMC, but rather than embedding MCMC in the standard Nested Sampling framework as we did in this chapter, it uses MCMC to explore a target distribution which is inspired by Nested Sampling. DNest is most useful on problems with large numbers of parameters and/or a complicated posterior distribution, but where the likelihood function can be evaluated quite quickly. It tends to outperform the version of NS presented here by a substantial margin. The main downside is that you need to write a fair bit of C++ code in order to implement a new model.

1.15 Acknowledgements

I would like to thank the organisers of the Winter School for their invitation, generosity and hospitality, and the students and other lecturers for many interesting discussions. I would also like to apologise to Earth's atmosphere and future inhabitants for my antipodal journey to Tenerife.

The 'Astrostatistics' Facebook group was very helpful with a MultiNest query, as was Thomas Lumley (Auckland) who helped me understand the finer mathematical points of the Nested Sampling "mapping" trick. I am grateful to all the friends I have met through this subject, from whom I

(hope I) have learned so much. Finally I would like to thank my wife Lianne for her support and understanding.

Bibliography

Brewer B. J., Pártay L. B., Csányi G., 2011, Statistics and Computing, 21, 4, 649-656. arXiv:0912.2380

Feroz F., Hobson M. P., Bridges M., 2009, MNRAS, 398, 1601

Foreman-Mackey, D., Hogg, D. W., Lang, D., & Goodman, J. 2012, emcee: The MCMC Hammer, arXiv:1202.3665

Homan, Matthew D., and Andrew Gelman, 2014, "The no-U-turn sampler: Adaptively setting path lengths in Hamiltonian Monte Carlo." The Journal of Machine Learning Research 15, no. 1 (2014): 1593-1623.

Gelman, Andrew, John B. Carlin, Hal S. Stern, and Donald B. Rubin. Bayesian data analysis, third edition. Chapman & Hall/CRC, 2013.

Goodman, J., Weare, J., 2010, Ensemble Samplers with Affine Invariance, Comm. App. Math. Comp. Sci., 5, 6.

Gregory, Phil. Bayesian Logical Data Analysis for the Physical Sciences: A Comparative Approach with Mathematica Support. Cambridge University Press, 2005.

Hansmann, Ulrich HE., 1997, Parallel tempering algorithm for conformational studies of biological molecules., Chemical Physics Letters 281, no. 1 (1997): 140-150.

Hogg, D. W., Bovy, J., Lang, D. 2010. Data analysis recipes: Fitting a model to data. ArXiv e-prints arXiv:1008.4686.

Mackay, D. J. C. 2003, Information theory, Inference, and Learning Algorithms, Cambridge University Press

Murray, Iain, 2007, "Advances in Markov chain Monte Carlo methods.", PhD thesis.

Neal, Radford M., 2011. "MCMC using Hamiltonian dynamics." Handbook of Markov Chain Monte Carlo 2.

Plummer, M., 2003, JAGS: A Program for Analysis of Bayesian Graphical Models Using Gibbs Sampling, Proceedings of the 3rd International Workshop on Distributed Statistical Computing (DSC 2003), March 20-22, Vienna, Austria. ISSN 1609-395X.

O'Hagan, A., Forster, J., 2004, Bayesian inference. London: Arnold.

Sivia, D. S., Skilling, J., 2006, Data Analysis: A Bayesian Tutorial, 2nd Edition, Oxford University Press

Skilling, J., 2006, "Nested Sampling for General Bayesian Computation", Bayesian Analysis 4, pp. 833-860.

Stan Development Team, 2014, Stan: A C++ Library for Probability and Sampling, Version 2.5.0 `http://mc-stan.org/`

Vousden, W., Farr, W. M., Mandel, I. 2015. Dynamic temperature selection for parallel-tempering in Markov chain Monte Carlo simulations. ArXiv e-prints arXiv:1501.05823.