25

A Python package for Bayesian estimation using Markov chain Monte Carlo

Christopher M. Strickland[1], Robert J. Denham[2], Clair L. Alston[1] and Kerrie L. Mengersen[1]

[1]Queensland University of Technology, Brisbane, Australia
[2]Department of Environment and Resource Management, Brisbane, Australia

25.1 Introduction

The most common approach currently used in the estimation of Bayesian models is Markov chain Monte Carlo (MCMC). **PyMCMC** is a Python module that is designed to simplify the construction of Markov chain Monte Carlo (MCMC) samplers, without sacrificing flexibility or performance. Python has extensive scientific libraries, is easily extensible, and has a clean syntax and powerful programming constructs, making it an ideal programming language to build an MCMC library; see van Rossum (1995) for further details on the programming language Python. **PyMCMC** contains objects for the Gibbs sampler, Metropolis–Hastings (MH), independent MH, random walk MH, orientational bias Monte Carlo (OBMC) as well as the slice sampler; see for example Robert and Casella (1999) for details on standard MCMC algorithms. The user can simply piece together the algorithms required and can easily include their own modules, where necessary. Along with the standard algorithms,

Case Studies in Bayesian Statistical Modelling and Analysis, First Edition. Edited by Clair L. Alston, Kerrie L. Mengersen and Anthony N. Pettitt.
© 2013 John Wiley & Sons, Ltd. Published 2013 by John Wiley & Sons, Ltd.

PyMCMC includes a module for Bayesian regression analysis. This module can be used for the direct analysis of linear models, or as a part of an MCMC scheme, where the conditional posterior has the form of a linear model. It also contains a class that can be used along with the Gibbs sampler for Bayesian variable selection.

The flexibility of **PyMCMC** is important in practice, as MCMC algorithms usually need to be tailored to the problem of interest in order to ensure good results. Issues such as block size and parameterization can have a dramatic effect on the convergence of MCMC sampling schemes. For instance, Lui *et al.* (1994) show theoretically that jointly sampling parameters in a Gibbs scheme typically leads to a reduction in correlation in the associated Markov chain in comparison with individually sampling parameters. This is demonstrated in practical applications in Carter and Kohn (1994) and Kim *et al.* (1998). Reducing the correlation in the Markov chain enables it to move more freely through the parameter space and as such enables it to escape from local modes in the posterior distribution. Parameterization can also have a dramatic effect on the convergence of MCMC samplers; see for example Gelfand *et al.* (1995), Sahu and Roberts (1997), Pitt and Shephard (1999), Robert and Mengersen (1999), Schnatter (2004) and Strickland *et al.* (2008), who show that the performance of the sampling schemes can be improved dramatically with the use of efficient parameterization.

PyMCMC aims to remove unnecessary repetitive coding and hence reduce the chance of coding error, and, importantly, greatly speed up the construction of efficient MCMC samplers. This is achieved by taking advantage of the flexibility of `Python`, which allows for the implementation of very general code. Another feature of `Python`, which is particularly important, is that it is also extremely easy to include modules from compiled languages such as `C` and `Fortran`. This is important to many practitioners who are forced, by the size and complexity of their problems, to write their MCMC programs entirely in compiled languages, such as `C/C++` and `Fortran`, in order to obtain the necessary speed for feasible practical analysis. With`Python`, the user can simply compile `Fortran` code using a module called **F2py** (Peterson 2009), or inline `C` using **Weave**, which is a part of **Scipy** (Oliphant 2007), and use the subroutines directly from `Python`. **F2py** can also be used to directly call `C` routines with the aid of a `Fortran` signature file. This enables the use of **PyMCMC** and `Python` as a rapid application development environment, without compromising on performance, by requiring only very small segments of code written in a compiled language. It should be mentioned that for most reasonably sized problems **PyMCMC** is sufficiently fast for practical MCMC analysis without the need for specialized modules.

Figure 25.1 is a flow chart that depicts the structure of **PyMCMC**. Essentially, the implementation of an MCMC sampler can be seen to centre around the class `MCMC`, which acts as a container for various algorithms that are used in sampling from the conditional posterior distributions that make up the MCMC sampling scheme.

The structure of the chapter is as follows. In Section 25.2 the algorithms contained in **PyMCMC** and the user interface are described. This includes the Gibbs sampler, the Metropolis-based algorithms and the slice sampler. Section 25.2 also contains a description of the Bayesian regression module. Section 25.3 contains three empirical examples that demonstrate how to use **PyMCMC**. The first example demonstrates how to use the regression module for the Bayesian analysis of the linear model. In

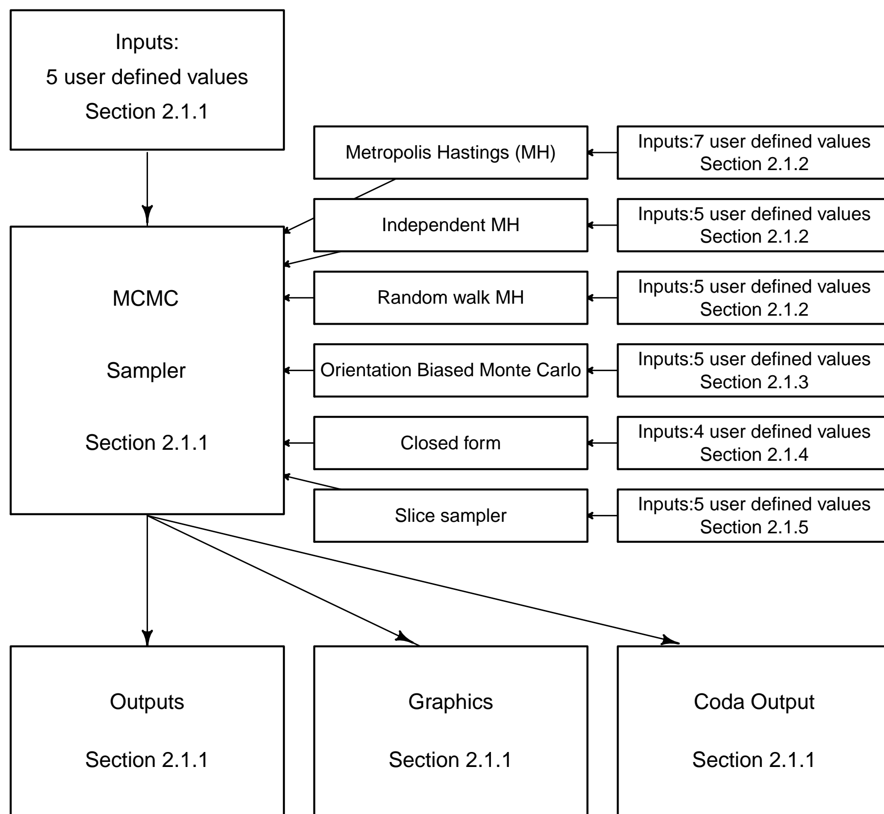

Figure 25.1 Flow chart illustrating the implementation of **PyMCMC**.

particular, the stochastic search variable selection algorithm, see George and McCulloch (1993) and Marin and Robert (2007), is used to select a set of 'most likely models'. The second example demonstrates how to use **PyMCMC** to analyse the log-linear model and the third example demonstrates how to use **PyMCMC** to analyse a linear model with first-order autoregressive errors. Section 25.4 contains a discussion on the efficient implementation of the MCMC algorithms using **PyMCMC**. Section 25.5 describes how to use **PyMCMC** interactively with R and Section 25.6 concludes.

25.2 Bayesian analysis

Bayesian analysis quantifies information about the unknown parameter vector of interest, $\boldsymbol{\theta}$, for a given data set, $\boldsymbol{y}$, through the joint posterior probability density function (pdf), $p(\boldsymbol{\theta}|\boldsymbol{y})$, which is defined such that

$$p(\boldsymbol{\theta}|\boldsymbol{y}) \propto p(\boldsymbol{y}|\boldsymbol{\theta}) \times p(\boldsymbol{\theta}), \tag{25.1}$$

where $p(\boldsymbol{y}|\boldsymbol{\theta})$ denotes the pdf of $\boldsymbol{y}$ given $\boldsymbol{\theta}$ and $p(\boldsymbol{\theta})$ is the prior pdf for $\boldsymbol{\theta}$. The most common approach used for inference about $\boldsymbol{\theta}$ is MCMC.

25.2.1 MCMC methods and implementation

In the following subsections, a brief description of each algorithm and the associated programming interface is included.

MCMC sampling

If we partition $\boldsymbol{\theta}$ into s blocks, that is $\boldsymbol{\theta} = (\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \ldots, \boldsymbol{\theta}_s)^T$, then the jth step for a generic MCMC sampling scheme is as given by Algorithm 27.

Algorithm 27: Gibbs sampler

1. Sample $\boldsymbol{\theta}_1^j$ from $p\left(\boldsymbol{\theta}_1|\boldsymbol{y},\boldsymbol{\theta}_2^{j-1}, \boldsymbol{\theta}_3^{j-1}, \ldots, \boldsymbol{\theta}_s^{j-1}\right)$.
2. Sample $\boldsymbol{\theta}_2^j$ from $p\left(\boldsymbol{\theta}_2|\boldsymbol{y},\boldsymbol{\theta}_1^{j}, \boldsymbol{\theta}_3^{j-1}, \boldsymbol{\theta}_4^{j-1}, \ldots, \boldsymbol{\theta}_s^{j-1}\right)$.

$\vdots$

s. Sample $\boldsymbol{\theta}_s^j$ from $p\left(\boldsymbol{\theta}_s|\boldsymbol{y},\boldsymbol{\theta}_1^{j}, \boldsymbol{\theta}_2^{j}, \ldots, \boldsymbol{\theta}_{s-1}^{j}\right)$.

An important special case of Algorithm 27 is the Gibbs sampler, which is an algorithm that is proposed in Gelfand and Smith (1990). Specifically, when each of $\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \ldots, \boldsymbol{\theta}_s$ is sampled from a closed form then this algorithm corresponds to that of the Gibbs sampler. **PyMCMC** contains a class that facilitates the implementation of Algorithm 27, in which the user must define functions to sample from each block, that is a function for each of $\boldsymbol{\theta}_i$, for $i = 1, \ldots, s$. These functions may be defined using the Metropolis-based or slice sampling algorithms that are part of **PyMCMC**. The class is named `MCMC` and the following arguments are required in the initialization of the class:

`nit`**:** The number of iterations.
`burn`**:** The burn-in length of the MCMC sampler.
`data`**:** A dictionary (`Python` data structure) containing any data, functions or objects that the user would like to have access to when defining the functions that are called from the Gibbs sampler.
`blocks`**:** A list (`Python` data structure) containing functions that are used to sample from the full conditional posterior distributions of interest.
`**kwargs` Optional arguments:
`loglike` A tuple (a `Python` data structure) containing a function that evaluates the log-likelihood, number of parameters and the name of the data set. For example, `loglike = (loglike, nparam, 'yvec')`. If this is defined

then the log-likelihood and the Bayesian information criterion (BIC) will be reported in the standard output.

`transform` A dictionary, where the keys are the names of the parameters and the associated values are functions that transform the iterates stored in the MCMC scheme. This can be useful when the MCMC algorithm is defined under a particular parameterization, but where it is desirable to report the results under a different parameterization.

Several functions are included as a part of the class:

`sampler()`: Used to run the MCMC sampler.

`get_mean_cov(listname)`: Returns the posterior covariance matrix for the parameters named in `listname`, where `listname` is a list that contains the parameter names of interest.

`get_parameter(name)`: Returns the iterates for the named parameter including the burn-in.

`get_parameter_exburn(name)`: Returns the iterates for the named parameter excluding the burn-in.

`get_mean_var(name)`: Returns the estimate from the MCMC estimation for the posterior mean and variance for the parameter defined by `name`.

`set_number_decimals(num)`: Sets the number of decimal places for the output.

`output(**kwargs)`: Used to produce output from the MCMC algorithm.

`**kwargs`: Optional arguments that control the output.

`parameters`: A dictionary, list or string specifying the parameters that are going to be presented.

- If a string is passed (e.g. `parameters = 'beta'`), all elements of that parameter are given.
- If a list (e.g. `parameters = ['alpha', 'beta']`), all elements of each parameter in the list are given.
- If a dictionary (e.g. `parameters = {'alpha': {'range': range(5)}}`), then there is the possibility to add an additional argument `'range'` that tells the output to only print a subset of the parameters. The above example will print information for `alpha[0], alpha[1], ..., alpha[4]` only.

`custom`: A user-defined function that produces custom output.

`filename`: A filename to which the output is printed. By default output will be printed to `stdout`.

`plot(blockname, **kwargs)`: Create summary plots of the MCMC sampler. By default, a plot of the marginal posterior density, an autocorrelation function (ACF) plot and a trace plot are produced for each parameter in the block. The plotting page is divided into a number of subfigures. By default, the number of columns is approximately equal to the square root of the total number of subfigures divided by the number of different plot types. Arguments to `plot` are:

`blockname`: The name of the parameter, for which summary plots are to be generated.

`**kwargs`: An optional dictionary (`Python` data structure) containing information to control the summary plots. The available keys are summarized below:

`elements`: A list of integers specifying the elements that will be plotted. For example, if the blockname is `'beta'` and $\boldsymbol{\beta} = (\beta_0, \beta_1, \ldots, \beta_n)$ then you may specify elements as `elements = [0,2,5]`.

`plottypes`: A list giving the type of plot for each parameter. By default the plots are `'density'`, `'acf'` and `'trace'`. A single string is also acceptable.

`filename`: A string providing the name of an output file for the plot. As a plot of a block may be made up of a number of subfigures, the output name will be modified to give a separate filename for each subfigure. For example, if the filename is passed as `'plot.png'`, and there are multiple pages of output, it will produce the files `plot001.png`, `plot002.png`, etc. The type of file is determined by the extension of the filename, but the output format will also depend on the plotting backend being used. If the filename does not have a suffix, a default format will be chosen based on the graphics backend. Most backends support png, pdf, ps, eps and svg (see the documentation for **Matplotlib** for further details: http://matplotlib.sourceforge.net).

`individual`: A Boolean option. If true, then each subplot will be done on an individual page.

`rows`: Integer specifying the number of rows of subfigures on a plotting page.

`cols`: Integer specifying the number of columns of subfigures on a plotting page.

`CODAoutput(**kwargs)`: Outputs the results in a format suitable for reading in with the statistical package Convergence Diagnostic and Output Analysis (**CODA**) (Plummer *et al.* 2006). By default, there will be two files created, `coda.txt` and `coda.ind`.

`**kwargs`: An optional dictionary controlling the **CODA** output.

`filename`: A string to provide an alternative filename for the output. If the file has an extension this will form the basis for the data file and the index file will be named by replacing the extension with `ind`. If no extension is in the filename then two files will be created and named by adding the extensions `.txt` and `.ind` to the given filename.

`parameters`: A string, a list or a dictionary that specifies the items written to file. It can be a string such as `'alpha'` or it can be a list (e.g., `['alpha','beta']`) or it can be a dictionary (e.g. `{'alpha':{'range':[0,1,5]}}`. If you supply a dictionary the key is the parameter name. It is also permissible to have a range key with a range of elements. If the range is not supplied it is assumed that the user wants all of the elements.

`thin`: Integer specifying how to thin the output. For example, if `thin = 10`, then every 10th element will be written to the **CODA** output.

MH

A particularly useful algorithm that is often used as a part of MCMC samplers is the MH algorithm (Algorithm 28); see for example Robert and Casella (1999). This algorithm is usually required when we cannot easily sample directly from $p(\boldsymbol{\theta}|\boldsymbol{y})$; however, we have a candidate density $q(\boldsymbol{\theta}|\boldsymbol{y}, \boldsymbol{\theta}^{j-1})$, which in practice is close to $p(\boldsymbol{\theta}|\boldsymbol{y})$ and is more readily able to be sampled. The MH algorithm at the jth iteration for $j = 1, 2, \ldots, M$ is given by the steps in Algorithm 28.

Algorithm 28: Metropolis–Hastings

1. Draw a candidate $\boldsymbol{\theta}^*$ from the density $q\left(\boldsymbol{\theta}|\boldsymbol{y}, \boldsymbol{\theta}^{j-1}\right)$.
2. Accept $\boldsymbol{\theta}^j = \boldsymbol{\theta}^*$ with probability equal to

$$\min\left\{1, \frac{p\left(\boldsymbol{\theta}^*|\boldsymbol{y}\right)}{p\left(\boldsymbol{\theta}^{j-1}|\boldsymbol{y}\right)} \Big/ \frac{q\left(\boldsymbol{\theta}^*|\boldsymbol{y}, \boldsymbol{\theta}^{j-1}\right)}{q\left(\boldsymbol{\theta}^{j-1}|\boldsymbol{y}, \boldsymbol{\theta}^*\right)}\right\}.$$

3. Otherwise $\boldsymbol{\theta}^j = \boldsymbol{\theta}^{j-1}$.

PyMCMC includes a class for the MH algorithm, which is called `MH`. To initialize the class the user needs to define the following:

`func`: User-defined function that returns a sample for the parameter of interest.
`actualprob`: User-defined function that returns the log probability of the parameters of interest evaluated using the target density.
`probcandprev`: User-defined function that returns the log of $q(\boldsymbol{\theta}^*|\boldsymbol{y}, \boldsymbol{\theta}^{j-1})$.
`probprevcand`: User-defined function that returns the log of $q(\boldsymbol{\theta}^{j-1}|\boldsymbol{y}, \boldsymbol{\theta}^*)$.
`init_theta`: Initial value for the parameters of interest.
`name`: The name of the parameter of interest.
`**kwargs`: Optional arguments:
- `store`
  - `'all'` (default) – stores every iterate for the parameter of interest. This is required for certain calculations.
  - `'none'` – does not store any of the iterates from the parameter of interest.
- `fixed_parameter` – is used if the user wants to fix the parameter value that is returned. This is used for testing MCMC sampling schemes. This command will override any other functionality.

Independent MH

The independent MH is a special case of the MH described in Algorithm 28. Specifically, the independent MH algorithm is applicable when we have a candidate

density $q(\boldsymbol{\theta}|\boldsymbol{y}) = q(\boldsymbol{\theta}|\boldsymbol{y}, \boldsymbol{\theta}^{j-1})$. The independent MH algorithm at the jth iteration for $j = 1, 2, \ldots, M$ is given by Algorithm 29.

Algorithm 29: Independent MH algorithm

1. Draw a candidate $\boldsymbol{\theta}^*$ from the density $q(\boldsymbol{\theta}|\boldsymbol{y})$.
2. Accept $\boldsymbol{\theta}^j = \boldsymbol{\theta}^*$ with probability equal to

$$\min\left\{1, \frac{p(\boldsymbol{\theta}^*|\boldsymbol{y})}{p\left(\boldsymbol{\theta}^{j-1}|\boldsymbol{y}\right)} \Big/ \frac{q(\boldsymbol{\theta}^*|\boldsymbol{y})}{q\left(\boldsymbol{\theta}^{j-1}|\boldsymbol{y}\right)}\right\}.$$

3. Otherwise accept $\boldsymbol{\theta}^j = \boldsymbol{\theta}^{j-1}$.

PyMCMC contains a class for the independent MH algorithm, named `IndMH`. To initialize the class the user needs to define the following:

`func`: A user-defined function that returns a sample for the parameter of interest.
`actualprob`: A user-defined function that returns the log probability of the parameters of interest evaluated using the target density.
`candpqrob`: A user-defined function that returns the log probability of the parameters of interest evaluated using the candidate density.
`init_theta`: Initial value for the parameters of interest.
`name`: Name of the parameter of interest.
`**kwargs`: Optional arguments:
  `store`
    `'all'` (default) – stores every iterate for the parameter of interest. This is required for certain calculations.
    `'none'` – does not store any of the iterates from the parameter of interest.
  `fixed_parameter` – is used if the user wants to fix the parameter value that is returned. This is used for testing MCMC sampling schemes. This command will override any other functionality.

Random walk MH

A useful and simple way to construct an MH candidate distribution is via

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}^{j-1} + \boldsymbol{\varepsilon}, \tag{25.2}$$

where $\boldsymbol{\varepsilon}$ is a random disturbance vector. If $\boldsymbol{\varepsilon}$ has a distribution that is symmetric about zero then the MH algorithm has a specific form that is referred to as the random walk MH algorithm. In this case, note that the candidate density is both independent of $\boldsymbol{y}$ and, due to symmetry, $q(\boldsymbol{\theta}^*|\boldsymbol{\theta}^{j-1}) = q(\boldsymbol{\theta}^{j-1}|\boldsymbol{\theta}^*)$. The random walk MH algorithm at the jth iteration for $j = 1, 2, \ldots, M$ is given by Algorithm 30.

Algorithm 30: Random walk MH

1. Draw a candidate $\boldsymbol{\theta}^*$ from Equation (25.2) where the random disturbance $\boldsymbol{\varepsilon}$ has a distribution symmetric about zero.
2. Accept $\boldsymbol{\theta}^j = \boldsymbol{\theta}^*$ with probability equal to

$$\min\left\{1, \frac{p\left(\boldsymbol{\theta}^*|\boldsymbol{y}\right)}{p\left(\boldsymbol{\theta}^{j-1}|\boldsymbol{y}\right)}\right\}.$$

3. Otherwise accept $\boldsymbol{\theta}^j = \boldsymbol{\theta}^{j-1}$.

A typical choice for the distribution of $\boldsymbol{\varepsilon}$ is a normal distribution, that is $\boldsymbol{\varepsilon} \sim \text{iid}\boldsymbol{N}(0, \boldsymbol{\Omega})$, where the covariance matrix $\boldsymbol{\Omega}$ is viewed as a tuning parameter. **PyMCMC** includes a class for the random walk MH algorithm, named `RWMH`. The class `RWMH` is defined assuming $\boldsymbol{\varepsilon}$ follows a normal distribution. Note that more general random walk MH algorithms could be constructed using the MH class. To initialize the class the user must specify the following:

`post`: A user-defined function for the log of full conditional posterior distribution for the parameters of interest.
`csig`: Scale parameter for the random walk MH algorithm.
`init_theta`: Initial value for the parameter of interest.
`name`: Name of the parameter of interest.
`kwargs`: Optional arguments:
`store`
`'all'` (default) – stores every iterate for the parameter of interest. This is required for certain calculations.
`'none'` – does not store any of the iterates from the parameter of interest.
`fixed_parameter` – is used if the user wants to fix the parameter value that is returned. This is used for testing MCMC sampling schemes. This command will override any other functionality.
`adaptive` - `'GFS'`: Then the adaptive random walk MH algorithm of Garthwaite *et al.* (2010) will be used to optimize $\boldsymbol{\Omega}$.

OBMC

The multiple try Metropolis (Liang *et al.* 2000) generalizes the MH algorithm to allow for multiple proposals. The OBMC algorithm is a special case of the multiple try Metropolis that is applicable when the candidate density is symmetric. The OBMC algorithm at iteration j is given in Algorithm 31.

Algorithm 31: Orientational bias Monte Carlo

1. Draw L candidates $\boldsymbol{\theta}_l^*$, $l = 1, 2, \ldots, L$, independently from the density $q(\boldsymbol{\theta}|\boldsymbol{y}, \boldsymbol{\theta}^{j-1})$, where $q(\boldsymbol{\theta}|\boldsymbol{y}, \boldsymbol{\theta}^{j-1})$ is a symmetric function.
2. Construct a *probability mass function (pmf)* by assigning to each $\boldsymbol{\theta}_l^*$ a probability proportional to $p(\boldsymbol{\theta}_l^*|\boldsymbol{y})$.
3. Select $\boldsymbol{\theta}^{**}$ randomly from this discrete distribution.
4. Draw $L-1$ reference points $\boldsymbol{r}_l$, $l = 1, 2, \ldots, L-1$, independently from $q(\boldsymbol{\theta}|\boldsymbol{y}, \boldsymbol{\theta}^{**})$ and set $\boldsymbol{r}_L = \boldsymbol{\theta}^{j-1}$.
5. Accept $\boldsymbol{\theta}^j = \boldsymbol{\theta}^{**}$ with probability equal to

$$\min\left\{1, \frac{\sum_{l=1}^{L} p\left(\boldsymbol{\theta}_l^*|\boldsymbol{y}\right)}{\sum_{l=1}^{L} p\left(\boldsymbol{r}_l|\boldsymbol{y}\right)}\right\}.$$

6. Otherwise accept $\boldsymbol{\theta}^j = \boldsymbol{\theta}^{j-1}$.

PyMCMC implements a special case of the OBMC algorithm, for which the candidate density is multivariate normal, making it a generalization of the random walk MH algorithm. The class for the OBMC algorithm is named `OBMC`. To initialize the class the user must specify the following:

`post`: A user-defined function for the log of the full conditional posterior distribution for the parameters of interest.
`ntry`: Number of candidates, L.
`csig`: A scale parameter for the OBMC algorithm.
`init_theta`: Initial value for the parameter of interest.
`**kwargs`: Optional arguments:
  `store`
    `'all'` (default) – stores every iterate for the parameter of interest. This is required for certain calculations.
    `'none'` – does not store any of the iterates from the parameter of interest.
  `fixed_parameter` – is used if the user wants to fix the parameter value that is returned. This is used for testing MCMC sampling schemes. This command will override any other functionality.

Closed form sampler

A class is included so that the user can specify a function to sample the parameters of interest when there is a closed form solution. The name of the class is `CFsampler`. To initialize the class the user must specify the following:

`func`: User-defined function that samples from the posterior distribution of interest.
`init_theta`: Initial value for the unknown parameter of interest.

`name`: The name of the parameter of interest.
`**kwargs`: Optional parameters:
  `store`
    `'all'` (default) – stores every iterate for the parameter of interest. This is required for certain calculations.
    `'none'` – does not store any of the iterates from the parameter of interest.
  `fixed_parameter` – is used if the user wants to fix the parameter value that is returned. This is used for testing MCMC sampling schemes. This command will override any other functionality.

Slice sampler

The slice sampler is useful for drawing values from complex densities; see Neal (2003) for further details. The required distribution must be proportional to one or a multiple of several other functions of the variable of interest:

$$p(\theta) \propto f_1(\theta) f_2(\theta) \cdots f_n(\theta).$$

A set of values from the distribution is obtained by iteratively sampling a new value, ω, from the vertical *slice* between 0 and $f_i(\theta)$, and then sampling a value for the parameter θ from the horizontal *slice* that consists of the set of possible values of θ, for which the previously sampled $\omega \leq p(\theta)$. This leads to the slice sampler algorithm, which can be defined at iteration j using Algorithm 32.

Algorithm 32: Slice sampler

1. For $i = 1, 2, \ldots, n$, draw $\omega_i \sim \text{Unif}[0,\ f_i(\theta^{j-1})]$.
2. Sample $\theta^j \sim \text{Unif}[A]$ where

$$A = \{\theta : f_1(\theta) \geq \omega_1 \in f_2(\theta) \geq \omega_2 \in \cdots \in f_n(\theta) \geq \omega_n\}.$$

In cases where the density of interest is not unimodal, determining the exact set A is not necessarily straightforward. The *stepping out* algorithm of Neal (2003) is used to obtain the set A. This algorithm is applied to each of the n slices to obtain the joint maximum and minimum of the slice. This results in a sampling interval that is designed to draw a new θ^j in the neighbourhood of θ^{j-1} and may include values outside the permissible range of A. The user is required to define an estimated typical slice size (ss), which is the width of set A, along with an integer value (N), which limits the width of any slice to $N \times ss$. The stepping out algorithm is given in Algorithm 33.

Algorithm 33: Stepping out

1. Initiate lower bound (LB) and upper bound (UB) for slice defined by set A.
   - $U \sim$ Unif(0, 1):
     - $LB = \theta^{j-1} - ss \times U$;
     - $UB = LB + ss$.
2. Sample $V \sim$ Unif(0, 1).
3. Set $J =$ Floor($N \times V$).
4. Set $Z = (N - 1) - J$.
5. Repeat while $J > 0$ and $\omega_i < f_i(LB)\,\forall i$:
   - $LB = LB - ss$;
   - $J = J - 1$.
6. Repeat while $Z > 0$ and $\omega_i < f_i(UB)\,\forall i$:
   - $UB = UB + ss$;
   - $Z = Z - 1$.
7. Sample $\theta^j \sim$ Unif(LB, UB).

The value of θ^j is accepted if it is drawn from a range $(LB, UB) \in A$. If it is outside the allowable range due to the interval (LB, UB) being larger in range than the set A, we then invoke a shrinkage technique to resample θ^j and improve the sampling efficiency of future draws, until an acceptable θ^j is drawn. The shrinkage algorithm is implemented as in Algorithm 34, repeating this algorithm until exit conditions are met.

Algorithm 34: Shrinkage

1. $U \sim$ Unif(0, 1).
2. $\theta^j = LB + U \times (UB - LB)$:
   - If $\omega_i < f_i(\omega_i)\,\forall i$, accept θ^j and exit.
   - Else if $\theta^j < \theta^{j-1}$, set $LB = \theta^j$ and return to step 1.
   - Else set $UB = \theta^j$ and return to step 1.

PyMCMC includes a class for the slice sampler named `SliceSampler`. To initialize the class the user must define the following:

`func`: A k-dimensional list containing the set of log functions.
`init_theta`: An initial value for θ.
`ssize`: A user-defined value for the typical slice size.
`sN` - An integer limiting slice size to $N \times ss$.
`**kwargs`: Optional arguments:

`store`
- `'all'` (default) – stores every iterate for the parameter of interest. This is required for certain calculations.
- `'none'` – does not store any of the iterates from the parameter of interest.

`fixed_parameter` – is used if the user wants to fix the parameter value that is returned. This is used for testing MCMC sampling schemes. This command will override any other functionality.

25.2.2 Normal linear Bayesian regression model

Many interesting models are partly linear for a subset of the unknown parameters. As such, drawing from the full conditional posterior distribution for the associated parameters may be equivalent to sampling the unknown parameters in a standard linear regression model. **PyMCMC** includes several classes that aid in the analysis of linear or partly linear models. In particular the classes `LinearModel`, `CondRegressionSampler`, `CondScaleSampler` and `StochasticSearch` are useful for this purpose. These classes are described in this section. For the standard linear regression model, see Zellner (1971), assume the $(n \times 1)$ observational vector, $\boldsymbol{y}$, is generated according to

$$\boldsymbol{y} = \boldsymbol{X\beta} + \boldsymbol{\varepsilon}; \quad \boldsymbol{\varepsilon} \sim N(\boldsymbol{0}, \sigma^2 \boldsymbol{I}), \tag{25.3}$$

where $\boldsymbol{X}$ is an $(n \times k)$ matrix of regressors, $\boldsymbol{\beta}$ is a $(k \times 1)$ vector of regression coefficients and $\boldsymbol{\varepsilon}$ is a normally distributed random variable with a mean vector $\boldsymbol{0}$ and an $(n \times n)$ covariance matrix, $\sigma^2 \boldsymbol{I}$. Assuming that both $\boldsymbol{\beta}$ and σ are unknown, then the posterior distribution for Equation (25.3) is given by

$$p(\boldsymbol{\beta}, \sigma | \boldsymbol{y}, \boldsymbol{X}) \propto p(\boldsymbol{y} | \boldsymbol{X}, \boldsymbol{\beta}, \sigma) \times p(\boldsymbol{\beta}, \sigma), \tag{25.4}$$

where

$$p(\boldsymbol{y} | \boldsymbol{X}, \boldsymbol{\beta}, \sigma) \propto \sigma^{-n} \exp\left\{ -\frac{1}{2\sigma^2} (\boldsymbol{y} - \boldsymbol{X\beta})^T (\boldsymbol{y} - \boldsymbol{X\beta}) \right\} \tag{25.5}$$

is the joint pdf for $\boldsymbol{y}$ given $\boldsymbol{X}$, $\boldsymbol{\beta}$ and σ, and $p(\boldsymbol{\beta}, \sigma)$ denotes the joint prior pdf for $\boldsymbol{\beta}$ and σ.

A class named `LinearModel` is defined to sample from the posterior distribution in Equation (25.4). One of four alternative priors may be used in the specification of the model. The default choice is Jeffreys' prior. Denoting the full set of unknown parameters as $\boldsymbol{\theta} = (\boldsymbol{\beta}^T, \sigma)^T$, then Jeffreys' prior is defined such that

$$p(\boldsymbol{\theta}) \propto |I(\boldsymbol{\theta})|^{-1/2}, \tag{25.6}$$

where $I(\boldsymbol{\theta})$ is the Fisher information matrix for $\boldsymbol{\theta}$. For the normal linear regression model in Equation (25.3), given the assumption that β and σ are a priori independent, Jeffreys' prior is flat over the real-number line for β, and σ is distributed such that

$$p(\sigma) \propto \frac{1}{\sigma}. \tag{25.7}$$

See Zellner (1971) for further details on Jeffreys' prior. Three alternative informative prior specifications are allowed, namely the normal–gamma, the normal–inverted-gamma and Zellner's g-prior; see Zellner (1971) and Marin and Robert (2007) for further details. The normal–gamma prior is specified such that

$$\boldsymbol{\beta}|\kappa \sim N(\underline{\boldsymbol{\beta}}, \underline{\boldsymbol{V}}^{-1}), \quad \kappa \sim G\left(\frac{\underline{\nu}}{2}, \frac{\underline{S}}{2}\right), \tag{25.8}$$

where $\kappa = \sigma^{-2}$ and $\underline{\beta}$, $\underline{\boldsymbol{V}}$, $\underline{\nu}$ and $\underline{S}$ are prior hyperparameters that take user-defined values. For the normal–gamma prior, `LinearModel` produces estimates for $(\kappa, \boldsymbol{\beta}^T)^T$ rather than $(\sigma, \boldsymbol{\beta}^T)$. The normal–inverted-gamma prior is specified such that

$$\boldsymbol{\beta}|\sigma^{-2} \sim N(\underline{\boldsymbol{\beta}}, \underline{\boldsymbol{V}}^{-1}), \quad \sigma^{-2} \sim IG\left(\frac{\underline{\nu}}{2}, \frac{\underline{S}}{2}\right), \tag{25.9}$$

where $\underline{\beta}$, $\underline{\boldsymbol{V}}$, $\underline{\nu}$ and $\underline{S}$ are prior hyperparameters, which take values that are set by the user. Zellner's g-prior is specified such that

$$\boldsymbol{\beta}|\sigma \sim N\left(\underline{\boldsymbol{\beta}}, g\sigma^2\left(X^TX\right)^{-1}\right), \quad p(\sigma) \propto \sigma^{-1}, \tag{25.10}$$

where $\underline{\boldsymbol{\beta}}$ and g are hyperparameters with values that are specified by the user. To initialize the class `LinearModel` the user must specify the following:

`yvec`: One-dimensional **Numpy** array containing the data.
`xmat`: Two-dimensional **Numpy** array containing the regressors.
`**kwargs`: Optional arguments:
  `prior`: A list containing the name of the prior and the corresponding hyperparameters. For example,
    `prior = ['normal_gamma', betaubar, Vubar, nuubar, Subar]`,
    `prior = ['normal_inverted_gamma',betaubar, Vubar, nuubar, Subar]` and
    `prior = ['g_prior', betaubar, g]`.
    If none of these options are chosen or they are misspecified then the default prior will be Jeffreys' prior.

`LinearModel` contains several functions that may be of interest to the user. In particular:

`sample()`: Returns a sample of σ and $\boldsymbol{\beta}$ from the joint posterior distribution for the normal–inverted-gamma prior, Jeffreys' prior and Zellner's g-prior. If the normal–gamma prior is specified then `sample()` returns κ and $\boldsymbol{\beta}$.
`update_yvec(yvec)`: Updates yvec in `LinearModel`. This is often useful when the class is being used as a part of the MCMC sampling scheme.
`update_xmat(xmat)`: Updates xmat in `LinearModel`. This is often useful when the class is being used as a part of the MCMC sampling scheme.

`loglike(scale, beta)`: Returns the log-likelihood.
`posterior_mean()`: Returns the posterior mean for the scale parameter (either σ or κ depending on the specified prior) and $\boldsymbol{\beta}$.
`get_posterior_covmat()`: Returns the posterior covariance matrix for $\boldsymbol{\beta}$.
`bic()`: Returns the BIC; see Kass and Raftery (1995) for details.
`plot(**kwargs)`: Produces standard plots. Specifically the marginal posterior density intervals for each element of $\boldsymbol{\beta}$ and for the scale parameter (σ or κ).
`residuals`: Returns the residual vector from the regression analysis. The residuals are calculated with $\boldsymbol{\beta}$ evaluated at the marginal posterior mean.
`output`: Produces standard output for the regression analysis. This includes the means, standard deviations and highest posterior density (HPD) intervals for the marginal posterior densities for each element of $\boldsymbol{\beta}$ and for the scale parameter (σ or κ). The output also reports the log-likelihood and the BIC.

In MCMC sampling schemes it is common that for a subset of the unknown parameters of interest the full conditional posterior distribution will correspond to that of a linear regression model, where the scale parameter is known. For the linear regression model specified in Equation (25.3) the posterior distribution for the case that σ is known is as follows:

$$p\,(\boldsymbol{\beta}|\boldsymbol{y}, \boldsymbol{X}, \boldsymbol{\beta}, \sigma) \propto p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{\beta}, \sigma) \times p(\boldsymbol{\beta}), \tag{25.11}$$

where $p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{\beta}, \sigma)$ is described in Equation (25.5) and $p(\boldsymbol{\beta})$ is the prior pdf for $\boldsymbol{\beta}$. To sample from Equation (25.11) a class named `CondRegressionSampler` can be used. The user may specify one of three alternative priors. The default prior is Jeffreys' prior, which for $\boldsymbol{\beta}$ is simply a flat prior over the real-number line. A normally distributed prior for $\boldsymbol{\beta}$ is another option, and can be specified such that

$$\boldsymbol{\beta} \sim N\left(\underline{\beta}, \boldsymbol{V}^{-1}\right).$$

The user may also specify their a priori beliefs using Zellner's g-prior, where

$$\boldsymbol{\beta}|\sigma \sim N\left(\boldsymbol{\beta}, g\sigma^2 \boldsymbol{X}^T \boldsymbol{X}\right).$$

To initialize the class the user must specify the following:

`yvec`: A one-dimensional **Numpy** array containing the data.
`xmat`: A two-dimensional **Numpy** array containing the regressors.
`**kwargs`: Optional arguments:
  `prior` – a list containing the name of the prior and the corresponding hyperparameters. For example,
  `prior=['normal', betaubar, Vubar]` or `['g_prior', betaubar, g]`.
  If none of these options are chosen or they are misspecified then the default prior will be Jeffreys' prior.

`CondRegressionSampler` contains several functions that may be of interest to the user. In particular:

`sample(sigma)`: Returns a sample of $\boldsymbol{\beta}$ from the posterior distribution specified in Equation (25.11).
`get_marginal_posterior_mean()`: Returns the marginal posterior mean for Equation (25.11).
`get_marginal_posterior_precision()`: Returns the marginal posterior precision for the linear conditional posterior distribution specified in Equation (25.11).
`update_yvec(yvec)`: Updates yvec in `CondRegressionSampler`. This is often useful when the class is being used as a part of an MCMC sampling scheme.
`update_xmat(xmat)`: Updates xmat in `CondRegressionSampler`. This is often useful when the class is being used as a part of an MCMC sampling scheme.

Many Bayesian models contain linear components with unknown scale parameters, hence a class has been specified named `CondScaleSampler`, which can be used to individually sample scale parameters from their posterior distributions. In particular, we wish to sample from

$$p(\sigma|\boldsymbol{y}, \boldsymbol{\theta}), \tag{25.12}$$

where $\boldsymbol{\theta}$ is the set of unknown parameters of interest excluding σ. The user may choose to use one of three priors. Jeffreys' prior, which for σ given the posterior in Equation (25.12) is as follows:

$$p(\sigma) \propto \frac{1}{\sigma}.$$

The second option is to specify an inverted-gamma prior, such that

$$\sigma \sim IG\left(\frac{\nu}{2}, \frac{S}{2}\right).$$

Alternatively, the user may specify a gamma prior for $\kappa = 1/\sigma^2$, where

$$\kappa \sim G\left(\frac{\nu}{2}, \frac{S}{2}\right).$$

To initialize the class `CondScaleSamper` the user may first specify the following:

`**kwargs`: Options arguments:
  `prior` – list containing the name of the prior and the corresponding hyperparameters. For example,
    `prior=['gamma',nuubar,subar]` or
    `prior=['inverted-gamma', nuubar, subar]`. If no prior is specified Jeffreys' prior is used.

PyMCMC also includes another class that can be used for the direct analysis of the linear regression model. The class is called `StochasticSearch` and can be used in conjunction with the class `MCMC`, for the purpose of variable selection.

The stochastic search algorithm can be used for variable selection in the linear regression model. Given a set of k possible regressors there are 2^k models to choose from. The stochastic search algorithm, as proposed by George and McCulloch (1993), uses the Gibbs sampler to select a set of 'most likely' models. The stochastic search algorithm is implemented in the class `StochasticSearch`. The specific implementation follows Marin and Robert (2007). The algorithm introduces the vector $\boldsymbol{\gamma}$, which is used to select the explanatory variables that are to be included in the model. In particular, $\boldsymbol{\gamma}$ is defined to be a binary vector of order k, whereby the inclusion of the ith regressor implies that the ith element of $\boldsymbol{\gamma}$ is a one, while the exclusion of the ith regressor implies that the ith element is zero. It is assumed that the first element of the design matrix is always included and should typically be a column of ones which is used to represent the constant or intercept in the regression. The algorithm specified to sample $\boldsymbol{\gamma}$ is a single move Gibbs sampling scheme; for further details see Marin and Robert (2007).

To use the class `StochasticSearch` the user must specify their a priori beliefs that the unknown parameters of interest, $(\sigma, \boldsymbol{\beta}^T)^T$, are distributed following Zellner's g-prior, which is described in Equation (25.10). `StochasticSearch` is designed to be used in conjunction with the MCMC sampling class. To initialize `StochasticSearch` the user must specify the following:

`yvec`: One-dimensional **Numpy** array containing the dependent variable.
`xmat`: Two-dimensional **Numpy** array containing the regressors.
`prior`: A list with the structure `[betaubar, g]`.

The class `StochasticSearch` also contains the following function:

`sample_gamma(store)`: Returns a sample of $\boldsymbol{\gamma}$. The only argument to pass into the function `sample_gamma` is the storage dictionary that is passed by default to each of the classes called from the class `MCMC` in **PyMCMC**.

25.3 Empirical illustrations

PyMCMC is illustrated though three examples. Specifically, a linear regression example with variable selection, a loglinear example and a linear regression model with first-order autoregressive errors. For each example, the model of interest is specified, then the code used for estimation is shown, following which a brief description of the code is given. Each example uses the module for **PyMCMC**, along with the `Python` libraries **Numpy**, **Scipy** and **Matplotlib**; see Oliphant (2007) and Hunter (2007) for further details of these `Python` libraries. Example 3 further uses the library **Pysparse** (Geus 2011).

25.3.1 Example 1: Linear regression model – variable selection and estimation

The data used in this example are a response of crop yield modelled using various chemical measurements from the soil. As the results of the chemical analysis of soil cores are obtained in a laboratory, many input variables are available and the data analyst would like to determine the variables that are most appropriate to use in the model.

The normal linear regression model in Equation (25.3) is used for the analysis. To select the set of 'most probable' regressors we use the stochastic search variable selection approach described in Section 25.2.2 using a subset of 19 explanatory variables.

In addition to **PyMCMC**, this example uses functions from **Numpy**, so the first step is to import the relevant packages:

```
import os
from numpy import loadtxt, hstack, ones, random, zeros,
    asfortranarray, log
from pymcmc.mcmc import MCMC, CFsampler
from pymcmc.regtools import StochasticSearch,
    LinearModel
```

The remaining code is typically organized so that the user-defined functions are at the top and the main program is at the bottom. We begin by defining functions that are called from the class `MCMC`, all of which take the argument `store`. In this example there is one such function:

```
def samplegamma(store):
    return store['SS'].sample_gamma(store)
```

In this case, `samplegamma` simply uses the pre-defined function available in the `StochasticSearch` class to sample from γ.

The data and an instance of the class `StochasticSearch` are now initialized:

```
data = loadtxt('yld2.txt')
yvec = data[:, 0]
xmat = data[:, 1:20]
xmat = hstack([ones((xmat.shape[0], 1)), xmat])
data ={'yvec':yvec, 'xmat':xmat}

prior = ['g_prior', zeros(xmat.shape[1]), 100.]
SSVS = StochasticSearch(yvec, xmat, prior);
data['SS'] = SSVS
```

Note that data is augmented to include the class instance for Stochastic-Search. In this example we use Zellner's g-prior with $\underline{\boldsymbol{\beta}} = 0$ and $g = 100$. The normal–inverted-gamma prior could also be used.

The next step is to initialize γ and set up the appropriate sampler for the model. In this case, as the full conditionals are of closed form, we can use the CFsampler class:

```
initgamma = zeros(xmat.shape[1], dtype ='i')
initgamma[0] = 1
simgam = CFsampler(samplegamma, initgamma, 'gamma')
```

The required arguments to CFSampler are a function that samples from the posterior (samplegamma), the initial value for γ (initgamma) and the name of the parameter of interest ('gamma'). The single argument to samplegamma, store, is a dictionary (Python data structure) that is passed to all functions that are called from the MCMC sampler. The purpose of store is to contain all the data required to define functions that sample from, or are used in the evaluation of, the posterior distribution. For example, we see that samplegamma accesses store['SS'], which contains the class instance for StochasticSearch. In addition to all the information contained in data, store contains the value of the previous iterate for each block of the MCMC scheme. It is possible, for example, to access the current iteration for any named parameter (in this case 'gamma') from any of the functions called from MCMC by using store['gamma']. This feature is not used in this example, but can be seen in the following examples.

The actual sampling is done by setting a random seed and running the MCMC sampler:

```
random.seed(12346)
ms = MCMC(20000, 5000, data, [simgam])
ms.sampler()
```

The MCMC sampler is initialized by setting the number of iterations (20 000), the burn-in (5000), providing the data dictionary data and a list containing the information used to sample from the full conditional posterior distributions. In this case, this list consists a single element, simgam, the CFsampler instance.

The default output from the MCMC object is a summary of each parameter providing the posterior mean, posterior standard deviation, 95% credible intervals and inefficiency factors. This can output directly to the screen, or be captured in an output file. A sample of this output, giving only the first four elements of γ, is

```
ms.output()

--------------------------------------------------------

The time (seconds) for the MCMC sampler = 7.4
```

```
Number of blocks in MCMC sampler = 1

             mean      sd  2.5%  97.5%  IFactor
gamma[0]        1       0     1      1       NA
gamma[1]   0.0929    0.29     0      1     3.75
gamma[2]   0.0941   0.292     0      1     3.54
gamma[3]   0.0939   0.292     0      1     3.54
```

In this case, the standard output is not all that useful as a summary of the variable selection. A more useful output, giving the 10 most likely models ordered by decreasing posterior probabilities, is available using the output function in the `StochasticSearch` class. This can be called by using the `custom` argument to `output`:

```
ms.output(custom = SSVS.output)

Most likely models ordered by decreasing posterior
    probability

-------------------------------------------------------
probability | model
-------------------------------------------------------
0.09353333  | 0, 12
0.0504      | 0, 11, 12
0.026       | 0, 10, 12
0.01373333  | 0, 9, 12
0.01353333  | 0, 8, 12
0.013       | 0, 4, 12
0.01293333  | 0, 12, 19
0.01206667  | 0, 7, 12
0.01086667  | 0, 11, 12, 17
0.01086667  | 0, 12, 17
-------------------------------------------------------
```

As indicated earlier, the MCMC analysis is conducted using 20 000 iterations, of which the first 5000 are discarded. The estimation takes 7.4 seconds in total. The results indicate that a model containing variable 12 along with a constant (indicated in the table by 0) is the most likely (prob = 0.09). Furthermore, variable 12 is contained in each of the 10 most likely models, indicating its strong association with crop yield.

Following the variable selection procedure, we may wish to fit a regression to the most likely model. This can be done using the `LinearModel` class. Firstly, we extract the explanatory variables from the most likely model

```
txmat = SSVS.extract_regressors(0)
```

and fit the model using the desired prior

```
g_prior = ['g_prior', 0.0, 100.]
```

```
breg = LinearModel(yvec,txmat,prior = g_prior)
```

The output summarizes the fit:

```
breg.output()

     ---------------------------------------------------
              Bayesian Linear Regression Summary
                          g_prior
     ---------------------------------------------------
                 mean          sd          2.5%       97.5%
beta[0]       -0.1254      0.1058       -0.3361     0.08533
beta[1]        0.7587      0.0225        0.7139      0.8035
  sigma        0.3853     0.03152           NA          NA

loglikelihood = -4.143
log marginal likelihood = nan
BIC  = 21.32
```

The parameter associated with variable 12, $\hat{\beta}_1$, is estimated as a positive value, 0.7587, with a 95% credible interval [0.7139, 0.8035]. We note that zero is not contained in the credible interval, hence the crop yield increases with higher values of variable 12. The marginal posterior densities (Figure 25.2) can be created using

```
breg.plot()
```

This clearly shows this effect is far from zero.

25.3.2 Example 2: Loglinear model

The data analysed in this example are the number of nutgrass shoots counted, in randomly scattered quadrats, at weekly intervals, during the growing season. A loglinear model is used:

$$\log(\text{count}) = \beta_0 + \beta_1 \text{week} \tag{25.13}$$

where the intercept, β_0, is expected to be positive in value, as nutgrass is always present in this study site, and β_1 is also expected to be positive as the population of nutgrass increases during the growing season.

For the loglinear model, see Gelman *et al.* (2004), the ith observation y_i, for $i = 1, 2, \ldots, n$, is generated as follows:

$$p(y_i|\mu_i) = \frac{\mu_i^{y_i} \exp(-\mu_i)}{\mu_i!}, \tag{25.14}$$

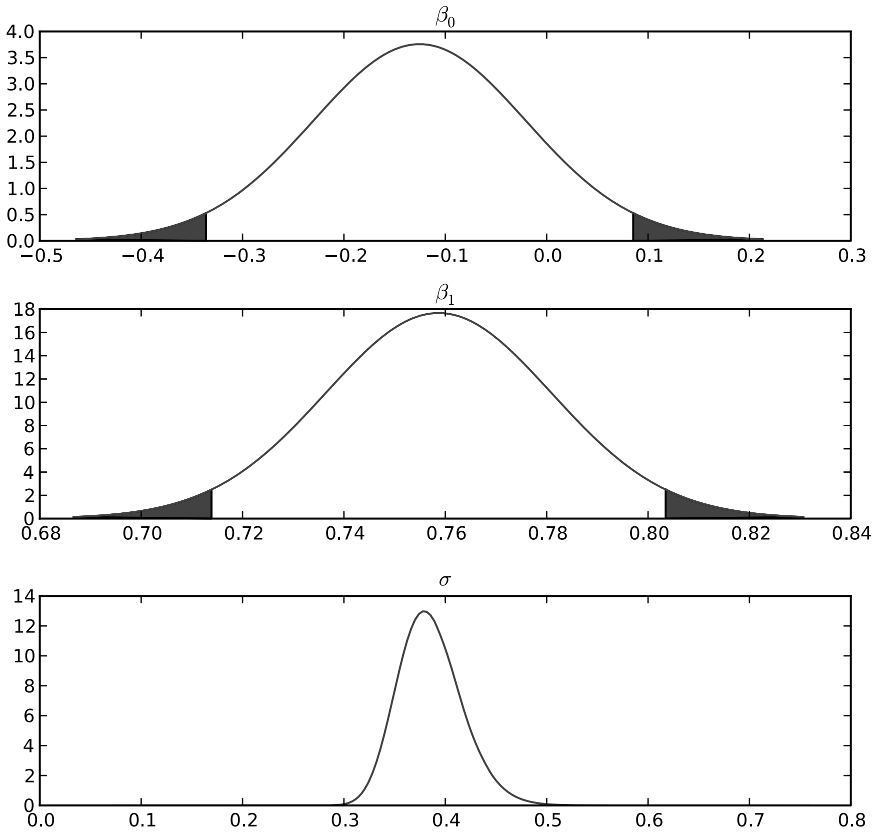

Figure 25.2 Marginal posterior density plots for the regression coefficients in Example 1.

with

$$\log(\mu_i) = \boldsymbol{x}_i^T \boldsymbol{\beta},$$

where $\boldsymbol{x}_i^T$ is the ith row of the $(n \times k)$ matrix $\boldsymbol{X}$.

The joint posterior distribution for the unknown parameter $\boldsymbol{\beta}$ is given by

$$p(\boldsymbol{\beta}|\boldsymbol{y}, \boldsymbol{X}) \propto p(\boldsymbol{y}|\boldsymbol{\beta}, \boldsymbol{X}) \times p(\boldsymbol{\beta}), \tag{25.15}$$

where $p(\boldsymbol{y}|\boldsymbol{\beta}, \boldsymbol{X})$ is the joint pdf for $\boldsymbol{y}$ conditional on the $\boldsymbol{\beta}$ and $\boldsymbol{X}$, and $p(\boldsymbol{\beta})$ denotes the prior pdf for $\boldsymbol{\beta}$. From Equation (25.14) it is apparent that

$$p(\boldsymbol{y}|\boldsymbol{\beta}, \boldsymbol{X}) = \prod_{i=1}^{n} \frac{\mu_i^{y_i} \exp(-\mu_i)}{\mu_i!}.$$

A priori we assume that

$$\boldsymbol{\beta} \sim N(\underline{\boldsymbol{\beta}}, \boldsymbol{V}^{-1}).$$

To sample from Equation (25.15), a random walk MH algorithm is implemented, where the candidate $\boldsymbol{\beta}^*$, at each iteration, is sampled following

$$\boldsymbol{\beta}^* \sim N\left(\boldsymbol{\beta}^{j-1}, \boldsymbol{\Omega}\right), \tag{25.16}$$

where

$$\boldsymbol{\beta}^0 = \boldsymbol{\beta}_{nls} = \arg\min\left(\boldsymbol{y} - \exp\left(\boldsymbol{X}\boldsymbol{\beta}\right)\right)^2$$

and

$$\boldsymbol{\Omega}^{-1} = -\sum_{i=1}^{n} \exp\left(\boldsymbol{x}_i^T \boldsymbol{\beta}_{nls}\right) \boldsymbol{x}_i \boldsymbol{x}_i^T.$$

The example code for **PyMCMC** uses two Python libraries, **Numpy** and **Scipy**, which the user must have installed to run the code.

As before, the code begins by importing the required packages:

```
import os
from numpy import random, loadtxt, hstack, ones, dot,
    exp, zeros, outer, diag
from numpy import linalg
from pymcmc.mcmc import MCMC, RWMH, OBMC
from pymcmc.regtools import LinearModel
from scipy.optimize.minpack import leastsq
```

The import statements are followed by the definition of a number or required functions, as below.

A function minfunc used in the nonlinear least squares routine:

```
def minfunc(beta, yvec, xmat ):
    return yvec - exp(dot(xmat, beta))
```

A function prior to evaluate the log of the prior pdf $\boldsymbol{\beta}$ and a function logl defining the log-likelihood function:

```
def prior(store):
    mu = zeros(store['beta'].shape[0])
    Prec = diag(0.005 * ones(store['beta'].shape[0]))
    return -0.5 * dot(store['beta'].transpose(),
        dot(Prec, store['beta']))

def logl(store):
    xbeta = dot(store['xmat'], store['beta'])
    lamb = exp(xbeta)
    return sum(store['yvec'] * xbeta - Lamb)
```

As in the variable selection example, `store` is a `Python` dictionary used to store all the information of interest that needs to be accessed by functions that are called from the MCMC sampler. For example, the function `logl` uses `store['beta']` which provides access to the vector $\boldsymbol{\beta}$.

A function `posterior` which evaluates the log of the posterior pdf for $\boldsymbol{\beta}$:

```
def posterior(store):
    return logl(store) + prior(store)
```

A function `llhessian` which returns the Hessian for the loglinear model:

```
def llhessian(store, beta):
    nobs = store['yvec'].shape[0]
    kreg = store['xmat'].shape[1]
    lamb = exp(dot(store['xmat'], beta))
    sum = zeros((kreg, kreg))
    for i in xrange(nobs):
        sum = sum + lamb[i] * outer(store['xmat'][i],
            store['xmat'][i])
    return -sum
```

Following the function definitions, the main program begins with the command to set the random seed

```
random.seed(12345)
```

and set up the data:

```
data = loadtxt('count.txt', skiprows = 1)
yvec = data[:, 0]
xmat = data[:, 1:data.shape[1]]
xmat = hstack([ones((data.shape[0], 1)), xmat])
data ={'yvec':yvec, 'xmat':xmat}
```

Bayesian regression is used to initialize the nonlinear least squares algorithm:

```
bayesreg = LinearModel(yvec, xmat)
sig, beta0 = bayesreg.posterior_mean()
```

The function `leastsq` from **Scipy** is used to perform the nonlinear least squares operation:

```
init_beta, info = leastsq(minfunc, beta0, args
    = (yvec, xmat))
data['betaprec'] =-llhessian(data, init_beta)
scale = linalg.inv(data['betaprec'])
```

Initialize the random walk MH algorithm:

```
samplebeta = RWMH(posterior, scale, init_beta, 'beta')
```

Finally, set up and run the sampling scheme. Note that the sampling algorithm is run for 20 000 iterations and the first 4000 are discarded. The MCMC scheme has only one block and is an MH sampling scheme:

```
ms = MCMC(20000, 4000, data, [samplebeta],
      loglike = (logl, xmat.shape[1], 'yvec'))
ms.sampler()
```

A summary of the model fit can be produced using the `output` function:

```
ms.output()
```

```
--------------------------------------------------------

The time (seconds) for the Gibbs sampler =  7.47
Number of blocks in Gibbs sampler =  1

           mean        sd      2.5%    97.5%    IFactor
beta[0]    1.14    0.0456      1.05     1.23       13.5
beta[1]   0.157   0.00428     0.148    0.165       12.2
Acceptance rate  beta  =  0.5625
BIC =  -7718.074
Log likelihood =  3864.453
```

It can be seen from the output that estimation is very fast (7.5 seconds), and that both β_0 and β_1 are positive values, with $\hat{\beta}_0 = 1.14$ [1.05, 1.23] and $\hat{\beta}_1 = 0.157$ [0.148, 0.165].

Summary plots can also be produced (Figure 25.3):

```
ms.plot('beta')
```

The marginal posterior densities of these estimates (Figure 25.3) confirm that both estimates are far from zero. The ACF plots (Figure 25.3) and low inefficiency factors (see Chib and Greenberg (1996) for details on inefficiency factors) of 13.5 ($\hat{\beta}_0$) and 12.2 ($\hat{\beta}_1$) show that autocorrelation in the sample is relatively low for an MCMC sampler.

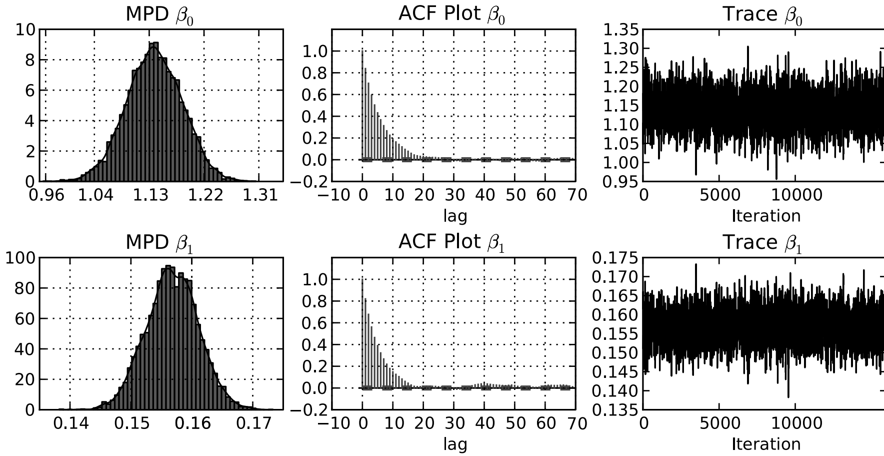

Figure 25.3 Plots of the marginal posterior density, autocorrelation and trace plots for the MCMC estimation of the loglinear model in Example 2. Note that summaries are calculated after removing the burn- in.

25.3.3 Example 3: First-order autoregressive regression

The final example demonstrates first-order autoregressive regression using a simulated data set, with 1000 observations and three regressors.

The linear regression model, with first-order autocorrelated serial correlation in the residuals, see Zellner (1971), is defined such that the tth observation, y_t, for $t = 1, 2, \ldots, n$, is

$$y_t = \boldsymbol{x}_t^T \boldsymbol{\beta} + \varepsilon_t, \tag{25.17}$$

with

$$\varepsilon_t = \rho \varepsilon_{t-1} + \nu_t; \quad \nu_t \sim \text{iid}\, N(0, \sigma^2), \tag{25.18}$$

where $\boldsymbol{x}_t$ is a $(k \times 1)$ vector of regressors, $\boldsymbol{\beta}$ is a $(k \times 1)$ vector of regression coefficients, ρ is a damping parameter and ν_t is an independent and identically normally distributed random variable with a mean of 0 and a variance of σ^2. Under the assumption that the process driving the errors is stationary, that is $|\rho| < 1$, and assuming that the process has been running since time immemorial, then Equations (25.17) and (25.18) can be expressed as

$$\boldsymbol{y} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\varepsilon}; \quad \boldsymbol{\varepsilon} \sim N\left(\boldsymbol{0}, \kappa^{-1}\boldsymbol{\Omega}^{-1}\right), \tag{25.19}$$

where

$$\boldsymbol{\Omega} = \begin{bmatrix} 1 & -\rho & 0 & 0 & \cdots & 0 \\ -\rho & 1+\rho^2 & -\rho & 0 & \ddots & 0 \\ 0 & -\rho & 1+\rho^2 & \ddots & \ddots & \vdots \\ 0 & 0 & \ddots & \ddots & -\rho & 0 \\ \vdots & \vdots & \ddots & -\rho & 1+\rho^2 & -\rho \\ 0 & 0 & \cdots & 0 & -\rho & 1 \end{bmatrix}.$$

Further, if we factorize $\boldsymbol{\Omega} = \boldsymbol{L}\boldsymbol{L}^T$, using the Cholesky decomposition, it is straightforward to derive $\boldsymbol{L}$, where

$$\boldsymbol{L} = \begin{bmatrix} 1 & -\rho & 0 & 0 & \cdots & 0 \\ 0 & 1 & -\rho & \ddots & \ddots & 0 \\ 0 & 0 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & 1 & -\rho \\ 0 & 0 & \cdots & \cdots & 0 & \sqrt{1-\rho^2} \end{bmatrix}.$$

Pre-multiplying Equation (25.19) by $\boldsymbol{L}$ gives

$$\tilde{\boldsymbol{y}} = \tilde{\boldsymbol{X}}\boldsymbol{\beta} + \tilde{\boldsymbol{\varepsilon}}, \tag{25.20}$$

where $\tilde{\boldsymbol{y}} = \boldsymbol{L}^T\boldsymbol{y}$, $\tilde{\boldsymbol{X}} = \boldsymbol{L}^T\boldsymbol{X}$ and $\tilde{\boldsymbol{\varepsilon}} = \boldsymbol{L}^T\boldsymbol{\varepsilon}$. Note that $\boldsymbol{\varepsilon} \sim N(0, \kappa^{-1}\boldsymbol{I})$.

The joint posterior distribution for the full set of unknown parameters is

$$p(\boldsymbol{\beta}, \kappa, \rho|\boldsymbol{y}) \propto p(\boldsymbol{y}|\boldsymbol{\beta}, \kappa, \rho) \times p(\boldsymbol{\beta}, \kappa) \times p(\rho), \tag{25.21}$$

where $p(\boldsymbol{y}|\boldsymbol{\beta}, \kappa, \rho)$ is the joint pdf of $\boldsymbol{y}$ conditional on $\boldsymbol{\beta}$, κ and ρ, $p(\boldsymbol{\beta}, \kappa)$ is the joint prior pdf for $\boldsymbol{\beta}$ and κ, and $p(\rho)$ denotes the prior density function for ρ. The likelihood function, which is defined following Equations (25.17) and (25.18), is defined as follows:

$$\begin{aligned} p(\boldsymbol{y}|\boldsymbol{\beta}, \kappa, \rho) &\propto \kappa^{n/2}|\boldsymbol{\Omega}|^{1/2} \exp\left\{-\frac{\kappa}{2}(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta})^T \boldsymbol{\Omega}(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta})\right\} \\ &= \kappa^{n/2}|\boldsymbol{\Omega}|^{1/2} \exp\left\{-\frac{\kappa}{2}\left(\tilde{\boldsymbol{y}} - \tilde{\boldsymbol{X}}\boldsymbol{\beta}\right)^T\left(\tilde{\boldsymbol{y}} - \tilde{\boldsymbol{X}}\boldsymbol{\beta}\right)\right\} \\ &= \kappa^{n/2}\left(1-\rho^2\right)^{1/2} \exp\left\{-\frac{\kappa}{2}\left(\tilde{\boldsymbol{y}} - \tilde{\boldsymbol{X}}\boldsymbol{\beta}\right)^T\left(\tilde{\boldsymbol{y}} - \tilde{\boldsymbol{X}}\boldsymbol{\beta}\right)\right\}. \end{aligned} \tag{25.22}$$

For the analysis a normal–gamma prior is assumed for $\boldsymbol{\beta}$ and κ, such that

$$\boldsymbol{\beta}|\kappa \sim N\left(\underline{\boldsymbol{\beta}}, \kappa^{-1}\right), \quad \kappa \sim G\left(\frac{\nu}{2}, \frac{S}{2}\right). \tag{25.23}$$

It follows from Equations (25.22) and (25.23) that sampling $\boldsymbol{\beta}$ and κ conditional on ρ is simply equivalent to sampling from a linear regression model with a normal–gamma prior. A beta prior is assumed for ρ, thereby restricting the autocorrelation of the time series to be both positive and stationary. Specifically

$$\rho \sim \mathcal{B}e\,(\alpha, \beta)\,.$$

An MCMC sampling scheme, for the posterior distribution in Equation (25.21), defined at iteration j is as follows:

1. Sample $\boldsymbol{\beta}^{(j)}, \kappa^{(j)}$ from $p(\boldsymbol{\beta}, \kappa|\boldsymbol{y}, \rho^{(j-1)})$.
2. Sample $\rho^{(j)}$ from $p(\rho|\boldsymbol{y}, \boldsymbol{\beta}, \kappa)$.

The code for this model follows the structure of the previous examples, and begins by importing the required packages:

```
from numpy import random, ones, zeros, dot, hstack,
    eye, log
from scipy import sparse
from pysparse import spmatrix
from pymcmc.mcmc import MCMC, SliceSampler, RWMH, OBMC,
    MH, CFsampler
from pymcmc.regtools import LinearModel
```

As this example uses simulated data, the first function is used to derive these data:

```
def simdata(nobs, kreg):
    xmat = hstack((ones((nobs, 1)), random.randn(nobs,
        kreg - 1)))
    beta = random.randn(kreg)
    sig = 0.2
    rho = 0.90
    yvec = zeros(nobs)
    eps = zeros(nobs)
    eps[0] = sig ** 2 / (1. - rho ** 2)
    for i in xrange(nobs - 1):
        eps[i + 1] = rho * eps[i] + sig * random.randn(1)
    yvec = dot(xmat, beta) + eps
    return yvec, xmat
```

Next, a function is defined to calculate $\tilde{\boldsymbol{y}}$ and $\tilde{\boldsymbol{X}}$:

```
def calcweighted(store):
    nobs = store['yvec'].shape[0]
    store['Upper'].put(-store['rho'], range(0, nobs - 1),
        range(1, nobs))
    store['Upper'].matvec(store['yvec'],
        store['yvectil'])
    for i in xrange(store['xmat'].shape[1]):
        store['Upper'].matvec(store['xmat'][:, i],
            store['xmattil'][:, i])
```

Note that $\boldsymbol{L}^T$ is updated based on the latest iteration in the MCMC scheme. Further, $\boldsymbol{L}^T$ is stored in the Python dictionary store and is accessed using the key 'Upper'. It is stored in sparse matrix format using the library **Pysparse**.

Next a function is defined and used to sample $\boldsymbol{\beta}$ from its conditional posterior distribution:

```
def WLS(store):
    calcweighted(store)
    store['regsampler'].update_yvec(store['yvectil'])
    store['regsampler'].update_xmat(store['xmattil'])
    return store['regsampler'].sample()
```

Now functions are defined to evaluate the log-likelihood (loglike), the log of the prior pdf for ρ (prior_rho) and the log of the posterior pdf for ρ (post_rho):

```
def loglike(store):
    nobs = store['yvec'].shape[0]
    calcweighted(store)
    store['regsampler'].update_yvec(store['yvectil'])
    store['regsampler'].update_xmat(store['xmattil'])
    return store['regsampler'].loglike(store['sigma'],
        store['beta'])

def prior_rho(store):
    if store['rho'] > 0. and store['rho'] < 1.0:
        alpha = 1.0
        beta = 1.0
        return (alpha - 1.) * log(store['rho'])
                + (beta - 1.) * log(1.-store['rho'])
    else:
        return -1E256

def post_rho(store):
    return loglike(store) + prior_rho(store)
```

The main program begins by setting the seed and constructing the `Python` dictionary `data`, which is used to store information that will be passed to functions that are called from the MCMC sampler:

```
random.seed(12345)
nobs = 1000
kreg = 3
yvec, xmat = simdata(nobs, kreg)
priorreg = ('g_prior', zeros(kreg), 1000.0)
regs = LinearModel(yvec, xmat, prior = priorreg)
data ={'yvec':yvec, 'xmat':xmat, 'regsampler':regs}
U = spmatrix.ll_mat(nobs, nobs, 2 * nobs - 1)
U.put(1.0, range(0, nobs), range(0, nobs))
data['yvectil'] = zeros(nobs)
data['xmattil'] = zeros((nobs, kreg))
data['Upper'] = U
```

Initial values for σ and β are set using Bayesian regression:

```
bayesreg = LinearModel(yvec, xmat)
sig, beta = bayesreg.posterior_mean()
```

The parameters σ and β are jointly sampled using the closed form class:

```
simsigbeta = CFsampler(WLS, [sig, beta], ['sigma',
    'beta'])
```

The parameter ρ is sampled using the slice sampler, with initial value 0.9:

```
rho = 0.9
simrho = SliceSampler([post_rho], 0.1, 5, rho, 'rho')
```

In this example, there are two blocks in the MCMC sampler. The code

```
blocks = [simrho, simsigbeta]
```

constructs a `Python` list that contains the class instances that define the MCMC sampler. In particular, it implies that the MCMC sampler consists of two blocks. Further, ρ will be the first element sampled in the MCMC scheme.

Finally, the sampler can be initialized and sampling undertaken:

```
loglikeinfo = (loglike, kreg + 2, 'yvec')
ms = MCMC(10000, 2000, data, blocks, loglike
     = loglikeinfo)
ms.sampler()
```

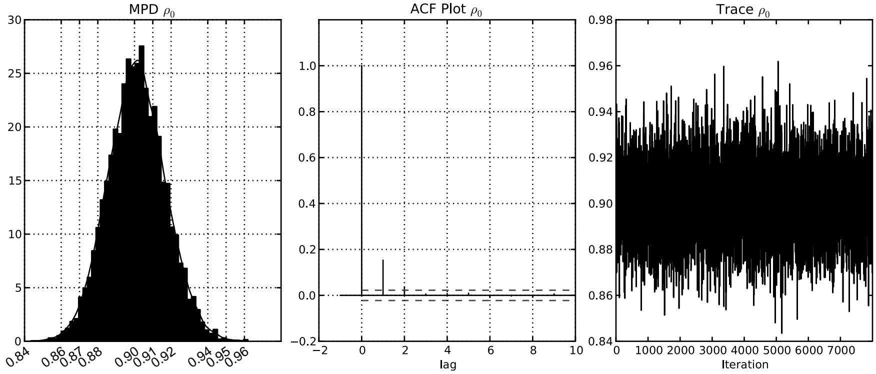

Figure 25.4 Marginal posterior density, autocorrelation function and trace plot based on the MCMC analysis for Example 3. Note that summaries are calculated after removing the burn-in.

The MCMC sampler is run for 10 000 iterations and the first 2000 are discarded. A summary can be generated by the `output` command:

```
ms.output()

The time (seconds) for the Gibbs sampler =  27.72
Number of blocks in Gibbs sampler =  2

             mean        sd       2.5%     97.5%    IFactor
beta[0]    -0.523    0.0716    -0.653    -0.373        3.5
beta[1]      1.85   0.00508      1.84      1.86       3.56
beta[2]     0.455   0.00505     0.445     0.465       3.75
  sigma     0.217   0.00489     0.207     0.226        3.5
    rho     0.901    0.0155     0.872     0.932       3.67

Acceptance rate  beta  =  1.0
Acceptance rate  sigma  =  1.0
Acceptance rate  rho  =  1.0
BIC =  -331.398
Log likelihood =   182.969
```

The total time of estimation is approximately 28 seconds. From the inefficiency factors it is clear that the algorithm is very efficient.

Summary plots generated using `ms.plot('rho')` (Figure 25.4) provide the marginal posterior density, autocorrelation plot and trace plot for the iterates.

25.4 Using PyMCMC efficiently

The fact that MCMC algorithms rely on a large number of iterations to achieve reasonable results and are often implemented on very large problems limits thepractitioner's

choice of a suitable environment, in which they can implement efficient code. This efficiency comes through an understanding of what makes a simulation-efficient MCMC sampler, and also the ability to produce computationally efficient code. Interestingly, the two are related. To achieve both simulation and computationally efficient code in MCMC samplers it is often extremely important that large numbers of parameters are sampled in blocks, rather than the alternative of a single move sampler. From the perspective of simulation efficiency it is well known that individually sampling correlated parameters induces correlation in the resultant Markov chain and thus leads to a poorly mixing sampler. A classic example in the literature uses *simulation smoothers* to jointly sample the state vector in a state space model; see for example Carter and Kohn (1994) and de Jong and Shephard (1995). While implementing a simulation smoother is required to achieve simulation-efficient code, the sequential nature of their implementation often renders higher level languages impractical for large problems and thus forces the analyst to write their entire code in a lower level language. This is an inefficient use of time as usually only a small percentage of code needs to be optimized. This drawback is easily circumvented in **PyMCMC** as `Python` makes it easy to use a lower level language to write the specialized module and use the functions directly from `Python`. This ensures **PyMCMC**'s modules can be used for rapid development from `Python` and lower level languages are only resorted to when necessary.

This section aims to provide guidelines for producing efficient code with **PyMCMC**. We discuss alternative external libraries that are available to the user for producing efficient code using **PyMCMC**. Despite the ease of writing specialized modules, this should not be the first resort of the user. Instead, one should ensure that the `Python` code is as efficient as possible using the resources available with `Python`.

Arguably, the first thing the user of **PyMCMC** should concentrate on when optimizing their **PyMCMC** code is to ensure they use as many inbuilt functions and libraries as possible. As most high-performance libraries are written in `C` or `Fortran` this ensures that computationally expensive procedures are computed using code from compiled languages. `Python` users, and hence **PyMCMC** users, have an enormous resource of scientific libraries available to them as a result of the popularity of `Python` in the scientific community. Two of the most important libraries for most users will quite possibly be **Numpy** and **Scipy**. Making use of such libraries is one of the best ways of avoiding large loops in procedures that are called from the MCMC sampler. If a large loop is used inside a function that is called from inside the MCMC sampler then this could mean that a large proportion of the total computation is being done by `Python`, rather than a library that was generated from optimized compiled code. This can have a dramatic effect on the total computation time. As a simple and somewhat trivial example we modify Example 2 from Section 25.3.2 so that a loop is explicitly used to calculate the log likelihood:

```
def logl(store):
    suml=0.0
    for i in xrange(store['yvec'].shape[0]):
```

```
        xbeta=dot(store['xmat'][i,:],store['beta'])
        suml=suml+store['yvec'][i] * xbeta - Exp(xbeta)
    return suml
```

While the two functions to calculate the log-likelihood are mathematically equivalent, the one with the explicit loop is substantially slower. Specifically, the time taken for the MCMC sampler went from 7.3 seconds to 130.19 seconds. As such, this minor modification leads to an approximate 18-fold decrease in the speed of the program.

If the use of an inbuilt function is not possible and the time taken from the program is unacceptable then there are several alternative solutions available to the user. One such solution is to use the package **Weave**, which is a part of the **Scipy** library, to write inline C code which will accelerate the problem area in the code. An example is given below:

```
def logl(store):
    code = """
double sum = 0.0, xbeta;
for(int i=0; i<nobs; i++){
xbeta = 0.0;
for(int j=0; j<kreg; j++){
                          xbeta += xmat(i,j) * beta(j);
                        }
sum += yvec(i) * xbeta - exp(xbeta);
}
return_val = sum;
    """
    yvec = store['yvec']
    xmat = store['xmat']
nobs, kreg = xmat.shape
beta = store['beta']
    return weave.inline(code,['yvec','xmat', 'beta',
        'nobs','kreg'], compiler='gcc',type_converters=
            converters.blitz)
```

The total time taken for the **Weave** version is 4.33 seconds. The reason for the speed increase over the original version that uses **Numpy** functions is that the **Weave** version avoids the construction of temporary matrices that are typically a by-product of overloaded operators.

Another alternative, which is our preferred approach, is to use the Python module **F2py**; see Peterson (2009) for further details. **F2py** allows for the seamless integration of Fortran and Python code. Following on and using the same trivial example we use the following Fortran77 code. This example requires that the user have basic linear algebra subprograms (**BLAS**) and preferably also the automatically tuned linear algebra software (**ATLAS**).

```
c     fortran 77 code used to calculate the likelihood
c     of a loglinear model. Subroutine uses BLAS.

      subroutine logl(xb,xm,bv,yv,llike,n,k)
      implicit none
      integer n, k, i, j
      real*8 xb(n),xm(n,k), bv(k), yv(n), llike
      real*8 alpha, beta

cf2py intent(in,out) logl
cf2py intent(in) yv
cf2py intent(in) bv
cf2py intent(in) xmat
cf2py intent(in) xb

      alpha=1.0
      beta=0.0
      call dgemv('n',n,k,alpha,xm,n,bv,1,beta,xb,1)

      llike=0.0
      do i=1,n
          llike=llike+yv(i)*xb(i)-exp(xb(i))
      enddo
      end
```

In UNIX-type environments, such as Linux and OSX, the code is compiled with the following command (Windows users, see Section 25.4.1):

```
f2py -c loglinear.f -m loglinear -lblas -latlas
```

The loglinear library can then be imported as a `Python` module, and the function `logl` accessed as a standard `Python` function:

```
import loglinear
print loglinear.logl.__doc__

logl - Function signature:
  llike = logl(xb,xm,bv,yv,llike,[n,k])
Required arguments:
  xb : input rank-1 array('d') with bounds (n)
  xm : input rank-2 array('d') with bounds (n,k)
  bv : input rank-1 array('d') with bounds (k)
  yv : input rank-1 array('d') with bounds (n)
  llike : input float
Optional arguments:
  n := len(xb) input int
  k := shape(xm,1) input int
Return objects:
  llike : float
```

Note that this function requires as input xb a rank 1 array of length n. We add this to the data dictionary:

```
data['xb']=zeros(yvec.shape[0])
```

The array data['xb'] is a work array used for the calculation of $\boldsymbol{X\beta}$. As it is stored in the Python dictionary data, it is only created once rather that each time the function logl is called.

It is useful to pass arrays stored in column major order to **F2py** functions since this is what is used in Fortran, rather than the Python default, which is row major order, the default for the C programming language. This can be achieved using the **Numpy** function asfortranarray:

```
data['xmatf']=asfortranarray(xmat)
```

If store['xmat'] were passed to the function loglinear.logl then **F2py** would automatically produce a copy and convert it to column major order each time the function logl is called.

The function logl can now be rewritten as

```
import loglinear
def logl(store):
loglike=array(0.0)
return loglinear.logl(store['xb'],store['xmatf'],
         store['beta'],store['yvec'],loglike)
```

The total time for the MCMC sampler when using **F2py** is 4.03 seconds. This is slightly faster than the version that uses **Weave**, where most likely the small gain can be attributed to the use of **ATLAS**.

The user has many other choices available to them for writing specialized extension modules. For example, if it is the preference of the user it is not much more difficult to use **F2py** to compile procedures written in C, which then can be used directly from Python. Another popular library that can be used to marry C, as well as C++, code with Python is **SWIG**. In our opinion **SWIG** is more difficult than **f2py** for complicated examples. The user may also opt to manually call C and C++ routines using Python and **Numpy**'s C application interface. Another option for C++ users is to use **Boost** Python. These alternative approaches are beyond the scope of this chapter.

25.4.1 Compiling code in Windows

The previous section described how one might go about using **PyMCMC** efficiently. To do this, access to a compiler is necessary. Under most flavours of UNIX, this should pose no problem, but under Microsoft Windows this can be more difficult.

This section provides some brief guidelines to an approach we found workable under Windows.

In order to run **PyMCMC**, Python, **Numpy** and **Scipy** are all required, but to have a reasonable developer experience under Windows, we suggest a few additional packages, all of which are freely available:

- **mingw** (http://www.mingw.org/), which provides, among other things, the GNU compiler suite. The user should choose at least **gcc** and **g++**.
- **msys** (http://www.mingw.org/wiki/MSYS), which provides a set of GNU utilities commonly found on Linux. This will make building and compiling code more manageable under Windows.
- **ipython** (http://ipython.scipy.org/moin/), an interactive interface to Python, which can be used as an alternative to the idle interface that is distributed with Python.
- **pyreadline** (http://ipython.scipy.org/moin/PyReadline/Intro), which provides Windows readline capabilities for **IPython**.
- **gfortran** (http://gcc.gnu.org/wiki/GFortranBinaries), which provides a native Windows Fortran compiler.

Once these additional utilities are installed, it should be possible to compile code in different languages. To test that **Weave** works as expected, make sure that the **mingw** bin directory is in your path, and try the following code:

```
import scipy.weave
a=100
scipy.weave.inline('printf("a=%d\\n",a);',['a'],
    verbose=1)
```

The output should be similar to

```
In [4]: scipy.weave.inline('printf("a=%d\\n",a);',['a'],
    verbose=1)
<weave: compiling>
No module named msvccompiler in numpy.distutils; trying
    from distutils
Compiling code...
Found executable c:\mingw\bin\g++.exe
finished compiling (sec):  2.73600006104
a=100
```

F2py requires a Fortran compiler. To set this up under Windows, follow the instructions at http://www.scipy.org/F2PY_Windows, and make sure the simple example provided works on your system. The examples presented above additionally require **BLAS** or **ATLAS** to be available. This can be built under Windows (see instructions at http://www.scipy.org/Installing_SciPy/Windows, for example). To check

that **F2py** and **ATLAS** are installed correctly, save the following code as, for example, blas_eg.f90:

```
subroutine dgemveg()
  REAL*8 X(2, 3) /1.D0, 2.D0, 3.D0, 4.D0, 5.D0, 6.D0/
  REAL*8 Y(3) /2.D0, 2.D0, 2.D0/
  REAL*8 Z(2)
  CALL DGEMV('N', 2, 3, 1.D0, X, 2, Y, 1, 0.D0, Z, 1)
  PRINT *, Z
end subroutine dgemveg
```

Set the location of your **ATLAS** libraries appropriately, ensure also that **gfortran** is in your path and compile. The following provides a template:

```
ATLAS_LIB_DIR="/d/tmp/pymcmc_win_install/BUILDS/lib"
export PATH=${PATH}:\
/c/Program\ Files/gfortran/libexec/gcc/i586-pc
    -mingw32/4.6.0:\
/c/Program\ Files/gfortran/bin:/c/python26
python /c/Python26/Scripts/f2py.py -c -m foo \
   --fcompiler=gfortran \
   blas_eg.f90 -L${ATLAS_LIB_DIR}-lf77blas-latlas-lg2c
```

This should produce a Python dll (foo.pyd), which can be imported into Python:

```
import foo
dir(foo)
print foo.__doc__
foo.dgemveg()
```

25.5 PyMCMC interacting with R

There are many functions from the R statistical language (R Development Core Team 2010) that can be useful in Bayesian analysis. The **RPy2** (Gautier 2011) Python library can be used to integrate R functions into **PyMCMC** programs. These can be accessed in **PyMCMC** through the **RPy2** Python library. As an example, consider the loglinear model described in Section 25.3.2. The random walk MH requires the specification of a candidate density function (Equation 25.2) and an initial value. The R functions glm and summary.glm can be used to set this to the maximum likelihood estimate $\hat{\beta}$ and the unscaled estimated covariance matrix of the estimated coefficients. The relevant code is summarized below:

```
import rpy2.robjects as robjects

def initial_values(yvec,xmat):
```

```
    ry = robjects.FloatVector(yvec)
    rv = robjects.FloatVector(xmat[:,1:].flatten())
    rx = robjects.r['matrix'](rv, nrow=xmat.shape[0],
                              byrow=True)
    robjects.globalenv['y'] = ry
    robjects.globalenv['x'] = rx
    mod = robjects.r.glm("y~x", family="poisson")
    init_beta =  array(robjects.r.coefficients(mod))
    modsummary = robjects.r.summary(mod)
    scale = array(modsummary.rx2('cov.unscaled'))
    return init_beta,scale


random.seed(12345)
data=loadtxt('count.txt',skiprows=1)
yvec=data[:,0]
xmat=data[:,1:data.shape[1]]
xmat=hstack([ones((data.shape[0],1)),xmat])

data={'yvec':yvec,'xmat':xmat}

init_beta,scale=initial_values(yvec,xmat)

samplebeta=RWMH(posterior,scale,init_beta,'beta')
ms=MCMC(20000,4000,data, [samplebeta],loglike=
    (logl,xmat.shape[1],'yvec'))
ms.sampler()
ms.CODAoutput(filename="loglinear_eg", parameter="beta")
```

It may also be useful to take advantage of the many MCMC analysis functions in R and associated packages. To facilitate this, **PyMCMC** includes a **CODA** (Plummer *et al.* 2006) output format which can easily be read into R for further analysis. A sample R session after **PyMCMC** might look like

```
library(coda)
aa <- read.coda("loglinear_eg.txt","loglinear_eg.ind")
plot(aa)
summary(aa)
raftery.diag(aa)
xyplot(aa)
densityplot(aa)
acfplot(aa,lag.max=500)
```

25.6 Conclusions

In this chapter, we describe the Python software package **PyMCMC**. **PyMCMC** takes advantage of the flexibility and extensibility of Python to provide the user with a code-efficient way of constructing MCMC samplers. The **PyMCMC** package includes classes for the MCMC sampler MH, independent MH, random walk MH,

OBMC and slice sampling algorithms. It also contains an inbuilt module for Bayesian regression analysis. We demonstrate **PyMCMC** using an example of Bayesian regression analysis with stochastic search variable selection, a loglinear model and also a time series regression analysis with first-order autoregressive errors. We demonstrate how to optimize **PyMCMC** using **Numpy** functions, inline `C` code using **Weave** and `Fortran77` using **F2py**, where necessary. We further demonstrate how to call `R` functions using **RPy2**.

25.7 Obtaining PyMCMC

The source code for **PyMCMC** is held in a **git** repostitory, and can be cloned by

```
git clone
https://bitbucket.org/christophermarkstrickland/pymcmc.git
```

As an alternative, for UNIX-based operating systems, a pre-packaged source distribution is available at

```
https://bitbucket.org/
christophermarkstrickland/pymcmc/downloads/pymcmc-1.0.tar.gz.
```

For most users, installation should require only

```
python setup.py install
```

More detailed installation instructions, including information on building for Microsoft Windows or Macintosh systems, are included in the `INSTALL` file in the source distribution. Additionally, binaries are available for Mac and Windows systems from

```
https://bitbucket.org/christophermarkstrickland/pymcmc/
 wiki/installing.
```

References

Carter C and Kohn R 1994 On Gibbs sampling for state space models. *Biometrika* **81**, 541–553.

Chib S and Greenberg E 1996 Markov chain Monte Carlo simulation methods in econometrics. *Econometric Theory* **12**, 409–431.

de Jong P and Shephard N 1995 The simulation smoother for time series models. *Biometrika* **82**, 339–350.

Garthwaite PH, Fan Y and Scisson SA 2010 Adaptive optimal scaling of Metropolis-Hastings algorithms using the Robbins-Monroe process. Technical Report, University of New South Wales.

Gautier L 2011 Rpy2: a simple and efficient access to R from Python. (accessed 6 March 2011).

Gelfand AE and Smith AFM 1990 Sampling-based approaches to calculating marginal densities. *Journal of the American Statistical Association* **85**, 398–409.

Gelfand AE, Sahu SK and Carlin BP 1995 Efficient parametrisations for normal linear mixed models. *Biometrika* **65**(3), 479–488.

Gelman A, Carlin JB, Stern HS and Rubin DB 2004 *Bayesian Data Analysis*. Chapman & Hall/CRC, Boca Raton, FL.

George EI and McCulloch RE 1993 Variable selection via Gibbs sampling. *Journal of the American Statistical Association* **88**, 881–889.

Geus R 2011 Pysparse. (accessed 6 March 2011).

Hunter JD 2007 Matplotlib: a 2D graphics environment. *Computing in Science and Engineering* **9**, 90–95.

Kass RE and Raftery AE 1995 Bayes factors. *Journal of the American Statistical Association* **90**, 773–795.

Kim S, Shephard N and Chib S 1998 Stochastic volatility: likelihood inference and comparison with arch models. *Review of Economic Studies* **65**(3), 361–393.

Liang F, Lui JS and Wong WH 2000 The use of multiple-try method and local optimization in Metropolis sampling. *Journal of the American Statistical Association* **95**, 121–134.

Lui JS, Wong WH and Kong A 1994 Covariance structure of the Gibbs sampler with applications to the comparisons of estimators and augmentations schemes. *Journal of the Royal Statistical Society, Series B* **57**(1), 157–169.

Marin JM and Robert CP 2007 *Bayesian core*. Springer, New York.

Neal RM 2003 Slice sampling. *Annals of Statistics* **31**(3), 705–741.

Oliphant TE 2007 Python for scientific computing. *Computing in Science and Engineering* **9**, 10–20.

Peterson P 2009 F2py: a tool for connecting Fortran and Python programs. *International Journal of Computational Science and Engineering* **4**, 296–605.

Pitt M and Shephard N 1999 Analytic convergence rates and parameterisation issues for the Gibbs sampler applied to state space models. *Journal of Time Series Analysis* **20**, 63–85.

Plummer M, Best N, Cowles K and Vines K 2006 CODA: convergence diagnosis and output analysis for MCMC. *R News* **6**(1), 7–11.

R Development Core Team 2010 *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna.

Robert CP and Casella G 1999 *Monte Carlo Statistical Methods*. Springer, New York.

Robert CP and Mengersen KL 1999 Reparameterisation issues in mixture modelling and their bearing on MCMC algorithms. *Computational Statistics & Data Analysis* **29**(3), 325–343.

Sahu SK and Roberts GO 1997 Updating schemes, correlation structure, blocking and parameterisation for the Gibbs sampler. *Journal of the Royal Statistical Society, Series B* **59**, 291–317.

Schnatter SF 2004 *Efficient Bayesian Parameter Estimation for State Space Models Based on Reparameterisations, State Space and Unobserved Component Models: Theory and Applications*. Cambridge University Press, Cambridge.

Strickland CM, Martin GM and Forbes CS 2008 Parameterisation and efficient MCMC estimation of non-Gaussian state space models. *Computational Statistics & Data Analysis* **52**, 2911–2930.

van Rossum G 1995 Python tutorial. Technical Report cs-r9526. Centrum voor Wiskunde en Informatica (CWI), Amsterdam.

Zellner A 1971 *An Introduction to Bayesian Inference in Econometrics*. John Wiley & Sons, Inc., New York.