LaplacesDemon: A Complete Environment for Bayesian Inference within R

Statisticat, LLC

Abstract

LaplacesDemon, also referred to as LD, is a contributed R package for Bayesian inference, and is freely available at http://www.bayesian-inference.com/software. The user may build any kind of probability model with a user-specified model function. The model may be updated with iterative quadrature, Laplace Approximation, MCMC, PMC, or variational Bayes. After updating, a variety of facilities are available, including MCMC diagnostics, posterior predictive checks, and validation. Hopefully, **LaplacesDemon** is generalizable and user-friendly for Bayesians, especially Laplacians.

Keywords: Bayesian, Big Data, High Performance Computing, HPC, Importance Sampling, Iterative Quadrature, Laplace Approximation, LaplacesDemon, Laplace's Demon, Markov chain Monte Carlo, MCMC, Metropolis, Optimization, Parallel, PMC, R, Rejection Sampling, Statisticat, Variational Bayes.

Bayesian inference is named after Reverend Thomas Bayes (1701-1761) for developing Bayes' theorem, which was published posthumously after his death (Bayes and Price 1763). This was the first instance of what would be called inverse probability[1].

Unaware of Bayes, Pierre-Simon Laplace (1749-1827) independently developed Bayes' theorem and first published his version in 1774, eleven years after Bayes, in one of Laplace's first major works (Laplace 1774, p. 366–367). In 1812, Laplace introduced a host of new ideas and mathematical techniques in his book, *Theorie Analytique des Probabilites* (Laplace 1812). Before Laplace, probability theory was solely concerned with developing a mathematical analysis of games of chance. Laplace applied probabilistic ideas to many scientific and practical problems. Although Laplace is not the father of probability, Laplace may be considered the father of the field of probability.

In 1814, Laplace published his "Essai Philosophique sur les Probabilites", which introduced a mathematical system of inductive reasoning based on probability (Laplace 1814). In it, the

[1] 'Inverse probability' refers to assigning a probability distribution to an unobserved variable, and is in essence, probability in the opposite direction of the usual sense. Bayes' theorem has been referred to as "the principle of inverse probability". Terminology has changed, and the term 'Bayesian probability' has displaced 'inverse probability'. The adjective "Bayesian" was introduced by R. A. Fisher as a derogatory term.

Bayesian interpretation of probability was developed independently by Laplace, much more thoroughly than Bayes, so some "Bayesians" refer to Bayesian inference as Laplacian inference. This is a translation of a quote in the introduction to this work:

> "We may regard the present state of the universe as the effect of its past and the cause of its future. An intellect which at a certain moment would know all forces that set nature in motion, and all positions of all items of which nature is composed, if this intellect were also vast enough to submit these data to analysis, it would embrace in a single formula the movements of the greatest bodies of the universe and those of the tiniest atom; for such an intellect nothing would be uncertain and the future just like the past would be present before its eyes" (Laplace 1814).

The 'intellect' has been referred to by future biographers as Laplace's Demon. In this quote, Laplace expresses his philosophical belief in hard determinism and his wish for a computational machine that is capable of estimating the universe.

This article is an introduction to an R (R Core Team 2013) package called **LaplacesDemon** (Statisticat LLC. 2014), which was designed without consideration for hard determinism, but instead with a lofty goal toward facilitating high-dimensional Bayesian (or Laplacian) inference[2], posing as its own intellect that is capable of impressive analysis. The **LaplacesDemon** R package is often referred to as LD. This article guides the user through installation, data, specifying a model, initial values, updating a numerical approximation algorithm, summarizing and plotting output, posterior predictive checks, general suggestions, discusses independence and observability, high performance computing, covers details of the algorithms, software comparisons, discusses large data sets and speed, and introduces `www.bayesian-inference.com`.

Herein, it is assumed that the reader has basic familiarity with Bayesian inference, numerical approximation, and R. If any part of this assumption is violated, then suggested sources include the vignette entitled "Bayesian Inference" that comes with the **LaplacesDemon** package, Robert (2007), and Crawley (2007).

1. Installation

To obtain the **LaplacesDemon** package, simply download the source code from `http://www.bayesian-inference.com/softwaredownload`, open R, and install the **LaplacesDemon** package from source:

```
> install.packages(pkgs="path/LaplacesDemon_ver.tar.gz", repos=NULL, type="source")
```

where `path` is a path to the zipped source code, and `_ver` is replaced with the latest version found in the name of the downloaded file.

A goal in developing the **LaplacesDemon** package was to minimize reliance on other packages or software. Therefore, the usual `dep=TRUE` argument does not need to be used, because the **LaplacesDemon** package does not depend on anything other than base R. Once installed,

[2]Even though the **LaplacesDemon** package is dedicated to Bayesian inference, frequentist inference may be used instead with the same functions by omitting the prior distributions and maximizing the likelihood.

simply use the `library` or `require` function in R to activate the **LaplacesDemon** package and load its functions into memory:

```
> library(LaplacesDemon)
```

2. Data

The **LaplacesDemon** package requires data that is specified in a list[3]. As an example, there is a data set called `demonsnacks` that is provided with the **LaplacesDemon** package. For no good reason, other than to provide an example, the log of `Calories` will be fit as an additive, linear function of the log of some of the remaining variables. Since an intercept will be included, a vector of 1's is inserted into design matrix **X**.

```
> data(demonsnacks)
> N <- nrow(demonsnacks)
> y <- log(demonsnacks$Calories)
> X <- cbind(1, as.matrix(log(demonsnacks[,c(1,4,10)]+1)))
> J <- ncol(X)
> for (j in 2:J) {X[,j] <- CenterScale(X[,j])}
> mon.names <- "LP"
> parm.names <- as.parm.names(list(beta=rep(0,J), sigma=0))
> pos.beta <- grep("beta", parm.names)
> pos.sigma <- grep("sigma", parm.names)
> PGF <- function(Data) return(c(rnormv(Data$J,0,10), rhalfcauchy(1,5)))
> MyData <- list(J=J, PGF=PGF, X=X, mon.names=mon.names,
+      parm.names=parm.names, pos.beta=pos.beta, pos.sigma=pos.sigma, y=y)
```

There are J=4 independent variables (including the intercept), one for each column in design matrix **X**. However, there are 5 parameters, since the residual variance, σ^2, must be included as well. Each parameter must have a name specified in the vector `parm.names`, and parameter names must be included with the data. This is using a function called `as.parm.names`. Also, note that each predictor has been centered and scaled, as per Gelman (2008). A `CenterScale` function is provided to center and scale predictors[4].

`PGF` is an optional, but highly recommended, user-specified function. PGF stands for Parameter-Generating Function, and is used by the `GIV` function, where GIV stands for Generating Initial Values. Although the `PGF` is not technically data, it is most conveniently placed in the list of data. When `PGF` is not specified and `GIV` is used, initial values are generated randomly without respect to prior distributions. To see why `PGF` was specified as it was, consider the following sections on specifying a model and initial values.

3. Specifying a Model

[3]Though most R functions use data in the form of a data frame, **LaplacesDemon** uses one or more numeric matrices in a list. It is much faster to process a numeric matrix than a data frame in iterative estimation.

[4]Centering and scaling a predictor is `x.cs <- (x - mean(x)) / (2*sd(x))`.

The **LaplacesDemon** package is capable of estimating any Bayesian model for which the likelihood is specified[5]. To use the **LaplacesDemon** package, the user must specify a model. Let's consider a linear regression model, which is often denoted as:

$$\mathbf{y} \sim \mathcal{N}(\mu, \sigma^2)$$
$$\mu = \mathbf{X}\beta$$

The dependent variable, $\mathbf{y}$, is normally distributed according to expectation vector μ and scalar variance σ^2, and expectation vector μ is equal to the inner product of design matrix $\mathbf{X}$ and transposed parameter vector β.

For a Bayesian model, the notation for the residual variance, σ^2, has often been replaced with the inverse of the residual precision, τ^{-1}. Here, σ^2 will be used. Prior probabilities are specified for β and σ (the standard deviation, rather than the variance):

$$\beta_j \sim \mathcal{N}(0, 1000), \quad j = 1, \ldots, J$$
$$\sigma \sim \mathcal{HC}(25)$$

Each of the J β parameters is assigned a vague[6] prior probability distribution that is normally-distributed according to $\mu = 0$ and $\sigma^2 = 1000$. The large variance or small precision indicates a lot of uncertainty about each β, and is hence a vague distribution. The residual standard deviation σ is half-Cauchy-distributed according to its hyperparameter, scale=25. When exploring new prior distributions, the user is encouraged to use the `is.proper` function to check for prior propriety.

To specify a model, the user must create a function called `Model`. Here is an example for a linear regression model:

```
> Model <- function(parm, Data)
+      {
+      ### Parameters
+      beta <- parm[Data$pos.beta]
+      sigma <- interval(parm[Data$pos.sigma], 1e-100, Inf)
+      parm[Data$pos.sigma] <- sigma
+      ### Log(Prior Densities)
+      beta.prior <- dnormv(beta, 0, 1000, log=TRUE)
+      sigma.prior <- dhalfcauchy(sigma, 25, log=TRUE)
+      ### Log-Likelihood
+      mu <- tcrossprod(beta, Data$X)
+      LL <- sum(dnorm(Data$y, mu, sigma, log=TRUE))
```

[5] Examples of more than 100 Bayesian models may be found in the "Examples" vignette that comes with the **LaplacesDemon** package. Likelihood-free estimation is also possible by approximating the likelihood, such as in Approximate Bayesian Computation (ABC).

[6] 'Traditionally, a vague prior would be considered to be under the class of uninformative or non-informative priors. 'Non-informative' may be more widely used than 'uninformative', but here that is considered poor English, such as saying something is 'non-correct' when there's a word for that ... 'incorrect'. In any case, uninformative priors do not actually exist (Irony and Singpurwalla 1997), because all priors are informative in some way. These priors are being described here as vague, but not as uninformative.

```
+      ### Log-Posterior
+      LP <- LL + sum(beta.prior) + sigma.prior
+      Modelout <- list(LP=LP, Dev=-2*LL, Monitor=LP,
+           yhat=rnorm(length(mu), mu, sigma), parm=parm)
+      return(Modelout)
+      }
```

A numerical approximation algorithm iteratively maximizes the logarithm of the unnormalized joint posterior density as specified in this `Model` function. In Bayesian inference, the logarithm of the unnormalized joint posterior density is proportional to the sum of the log-likelihood and logarithm of the prior densities:

$$\log[p(\Theta|\mathbf{y})] \propto \log[p(\mathbf{y}|\Theta)] + \log[p(\Theta)]$$

where Θ is a set of parameters, $\mathbf{y}$ is the data, $\propto$ means 'proportional to'[7], $p(\Theta|\mathbf{y})$ is the joint posterior density, $p(\mathbf{y}|\Theta)$ is the likelihood, and $p(\Theta)$ is the set of prior densities.

During each iteration in which a numerical approximation algorithm is maximizing the logarithm of the unnormalized joint posterior density, two arguments are passed to `Model`: `parm` and `Data`, where `parm` is short for the set of parameters, and `Data` is a list of data. These arguments are specified in the beginning of the function:

```
Model <- function(parm, Data)
```

Then, the `Model` function is evaluated and the logarithm of the unnormalized joint posterior density is calculated as `LP`, and returned in a list called `Modelout`, along with the deviance (`Dev`), a vector (`Monitor`) of any variables desired to be monitored in addition to the parameters, $\mathbf{y}^{rep}$ (`yhat`) or replicates of $\mathbf{y}$, and the parameter vector `parm`. All arguments must be returned. Even if there is no desire to observe the deviance and any monitored variable, a scalar must be placed in the second position of the `Modelout` list, and at least one element of a vector for a monitored variable. This can be seen in the end of the function:

```
LP <- LL + sum(beta.prior) + sigma.prior
Modelout <- list(LP=LP, Dev=-2*LL, Monitor=LP,
    yhat=rnorm(length(mu), mu, sigma), parm=parm)
return(Modelout)
```

The rest of the function specifies the parameters, log of the prior densities, and calculates the log-likelihood. Since design matrix $\mathbf{X}$ has J=4 column vectors (including the intercept), there are 4 `beta` parameters and a `sigma` parameter for the residual standard deviation.

Since a vector of parameters called `parm` is passed to `Model`, the function needs to know which parameter is associated with which element of `parm`. For this, the vector `beta` is declared, and then each element of `beta` is populated with the value associated in the corresponding element of `parm`. Above, the `grep` function was used to populate `pos.beta` and `pos.sigma`, which indicate the positions of β and σ. These positions are stored in the list of data, and used in the `Model` function to extract the appropriate parameters from vector `parm`:

[7] For those unfamiliar with $\propto$, this symbol simply means that two quantities are proportional if they vary in such a way that one is a constant multiplier of the other. This is due to an unspecified constant of proportionality in the equation. Here, this can be treated as 'equal to'.

```
beta <- parm[Data$pos.beta
sigma <- interval(parm[Data$pos.sigma], 1e-100, Inf)
parm[Data$pos.sigma] <- sigma
```

The σ parameter must be positive-only, and so it is constrained to be positive in the `interval` function. The algorithm, outside of the `Model` function needs to be aware that σ has been constrained, so the `parm` vector is updated with the constrained value.

The user does not have to constrain parameters in this way. For example, an alternative is to reparameterize to real values, such as with a logarithm, in this case. If the user does not constrain or reparameterize a parameter that is not on the real line, then the algorithm will be unaware, and probably attempt a value outside of realistic bounds, such as a negative standard deviation in this example.

To work with the log of the prior densities and according to the assigned names of the parameters and hyperparameters, they are specified as follows:

```
beta.prior <- dnormv(beta, 0, 1000, log=TRUE)
sigma.prior <- dhalfcauchy(sigma, 25, log=TRUE)
```

In the above example, the residual standard deviation `sigma` receives a half-Cauchy distributed prior of the form:

$$\sigma \sim \mathcal{HC}(25)$$

Finally, everything is put together to calculate `LP`, the logarithm of the unnormalized joint posterior density. The expectation vector `mu` is the inner product of the design matrix, `Data$X`, and the transpose of the vector `beta`. Expectation vector `mu`, vector `Data$y`, and scalar `sigma` are used to estimate the sum of the log-likelihoods, where:

$$\mathbf{y} \sim \mathcal{N}(\mu, \sigma^2)$$

and as noted before, the logarithm of the unnormalized joint posterior density is:

$$\log[p(\Theta|\mathbf{y})] \propto \log[p(\mathbf{y}|\Theta)] + \log[p(\Theta)]$$

```
mu <- tcrossprod(Data$X, t(beta))
LL <- sum(dnorm(Data$y, mu, sigma, log=TRUE)
LP <- LL + sum(beta.prior) + sigma.prior
```

In retrospect, the `PGF` function was specified so that when the list of data is passed to it, it generates and returns an initial value for each of the `beta` parameters, as well as one for the `sigma` parameter.

Specifying the model in the `Model` function is the most involved aspect for the user of the **LaplacesDemon** package. But this package has been designed so it is also incredibly flexible, allowing a wide variety of Bayesian models to be specified.

4. Initial Values

Each numerical approximation algorithm in the **LaplacesDemon** package requires a vector of initial values for the parameters. Each initial value is a starting point for the estimation of a

parameter. In this example, there are 5 parameters. The order of the elements of the vector of initial values must match the order of the parameters associated with each element of `parm` passed to the `Model` function. With no prior knowledge, it is a good idea to randomize each initial value, such as with the `GIV` function (which stands for "generate initial values").

When all initial values are set to zero for MCMC, the `LaplacesDemon` function optimizes initial values using a spectral projected gradient algorithm in the `LaplaceApproximation` function. Laplace Approximation is asymptotic with respect to sample size, so it is inappropriate in this example with a sample size of 39 and 5 parameters. MCMC will not use Laplace Approximation when the sample size is not at least five times the number of parameters.

```
> Initial.Values <- c(rep(0,J), 1)
```

5. Numerical Approximation

Compared to specifying the model in the `Model` function, updating a model is easy. Since pseudo-random numbers are involved, it's a good idea to set a 'seed' for pseudo-random number generation, so results can be reproduced. Pick any number you like, but there's only one number appropriate for a demon[8]:

```
> set.seed(666)
```

The **LaplacesDemon** package offers a wide variety of numerical approximation algorithms. Details may be found below in section 12, and also in the appropriate function documentation. If the user is new to Bayesian inference, then the best suggestion may be to consider Laplace Approximation with the `LaplaceApproximation` function when sufficient sample size is available, or MCMC with the `LaplacesDemon` function otherwise. This guideline is too simple, but serves as a place to start. For this example, the `LaplacesDemon` function will be used.

As with any R package, the user can learn about a function by using the `help` function and including the name of the desired function. To learn the details of the **LaplacesDemon** function, enter:

```
> help(LaplacesDemon)
```

Here is one of many possible ways to begin:

```
> Fit <- LaplacesDemon(Model, Data=MyData, Initial.Values,
+      Covar=NULL, Iterations=150000, Status=50000, Thinning=150,
+      Algorithm="HARM", Specs=NULL)
```

In this example, an output object called `Fit` will be created as a result of using the **LaplacesDemon** function. `Fit` is an object of class `demonoid`, which means that since it has been assigned a customized class, other functions have been custom-designed to work with it. The above example specifies the HARM algorithm for updating.

[8]Demonic references are used only to add flavor to the software and its use, and in no way endorses beliefs in demons. This specific pseudo-random seed is often referred to, jokingly, as the 'demon seed'.

This example tells the **LaplacesDemon** function to maximize the first component in the list output from the user-specified `Model` function, given a data set called `Data`, and according to several settings.

- The `Initial.Values` argument requires a vector of initial values for the parameters.
- The `Covar=NULL` argument indicates that a user-specified variance vector or covariance matrix has not been supplied. HARM does not use proposal variance or covariance.
- The `Iterations=150000` argument indicates that the `LaplacesDemon` function will update 150,000 times before completion.
- The `Status=50000` argument indicates that a status message will be printed to the R console every 50,000 iterations.
- The `Thinning=150` argument indicates that only ever Kth iteration will be retained in the output, and in this case, every 150th iteration will be retained. See the `Thin` function for more information on thinning.
- The `Algorithm` argument requires the abbreviated name of the MCMC algorithm in quotes.
- Finally, the `Specs` argument contains specifications for each algorithm named in the `Algorithm` argument. The `HARM` algorithm does not require specifications. Details on algorithms and specifications are given later.

By running[9] the `LaplacesDemon` function, the following output was obtained:

```
> Fit <- LaplacesDemon(Model, Data=MyData, Initial.Values,
+      Covar=NULL, Iterations=150000, Status=50000, Thinning=150,
+      Algorithm="HARM", Specs=NULL)

Laplace's Demon was called on Tue May  6 15:11:02 2014

Performing initial checks...
Algorithm: Hit-And-Run Metropolis

Laplace's Demon is beginning to update...
Iteration: 50000Iteration: 50000,   Proposal: Multivariate,   LP:-62.3
Iteration: 100000Iteration: 100000,   Proposal: Multivariate,   LP:-61.7
Iteration: 150000Iteration: 150000,   Proposal: Multivariate,   LP:-61.1

Assessing Stationarity
Assessing Thinning and ESS
Creating Summaries
Estimating Log of the Marginal Likelihood
Creating Output

Laplace's Demon has finished.
```

[9] This is "turning the Bayesian crank", as Dennis Lindley used to say.

`LaplacesDemon` finished quickly, though it had a small data set (N=39), few parameters (K=5), and the model was very simple. The output object, `Fit`, was created as a list. As with any R object, use `str()` to examine its structure:

```
> str(Fit)
```

To access any of these values in the output object `Fit`, simply append a dollar sign and the name of the component. For example, here is how to access the observed acceptance rate:

```
> Fit$Acceptance.Rate

[1] 0.23593
```

5.1. Warnings

Warnings did not occur with this example. If warnings result after updating the model with `LaplacesDemon`, and if the model was specified correctly, then the most likely cause is the posterior predictive distribution, returned in the `Model` function as `yhat`. Early in a model update, this may not be alarming, since extreme values may be generated. Sometimes it is related to the MCMC algorithm, so selecting a different algorithm may be necessary.

If warnings continue to occur, then the priors or parameterization should be considered. An example is when a scale parameter for the posterior predictive distribution is allowed to be too small or large.

6. Summarizing Output

The output object, `Fit`, has many components. The (copious) contents of `Fit` can be printed to the screen with the usual R functions:

```
> Fit
> print(Fit)
```

While a user is welcome to continue this R convention, the **LaplacesDemon** package adds another feature below the `print` function output in the `Consort` function. But before describing the additional feature, the results are obtained as:

```
> Consort(Fit)

############################################################
# Consort with Laplace's Demon                             #
############################################################
Call:
LaplacesDemon(Model = Model, Data = MyData, Initial.Values = Initial.Values,
    Covar = NULL, Iterations = 150000, Status = 50000, Thinning = 150,
    Algorithm = "HARM", Specs = NULL)
```

```
Acceptance Rate: 0.23593
Algorithm: Hit-And-Run Metropolis
Covariance Matrix: (NOT SHOWN HERE; diagonal shown instead)
  beta[1]   beta[2]   beta[3]   beta[4]     sigma
0.0383062 0.0768386 0.0988002 0.1099076 0.0089648

Covariance (Diagonal) History: (NOT SHOWN HERE)
Deviance Information Criterion (DIC):
        All Stationary
Dbar 82.958     82.958
pD    7.369      7.369
DIC  90.328     90.328
Initial Values:
[1] 0 0 0 0 1

Iterations: 150000
Log(Marginal Likelihood): -43.229
Minutes of run-time: 0.26
Model: (NOT SHOWN HERE)
Monitor: (NOT SHOWN HERE)
Parameters (Number of): 5
Posterior1: (NOT SHOWN HERE)
Posterior2: (NOT SHOWN HERE)
Recommended Burn-In of Thinned Samples: 0
Recommended Burn-In of Un-thinned Samples: 0
Recommended Thinning: 150
Specs: (NOT SHOWN HERE)
Status is displayed every 50000 iterations
Summary1: (SHOWN BELOW)
Summary2: (SHOWN BELOW)
Thinned Samples: 1000
Thinning: 150


Summary of All Samples
              Mean      SD       MCSE     ESS         LB
beta[1]    5.04337 0.11362 0.0036514 1000.00   4.833182
beta[2]    0.57981 0.27673 0.0091987  887.34   0.076972
beta[3]    1.17592 0.31228 0.0109790  890.90   0.544792
beta[4]    0.89981 0.33047 0.0129093  851.71   0.269623
sigma      0.71506 0.09430 0.0030039 1000.00   0.566679
Deviance  82.95846 3.83904 0.1154811 1000.00  78.229166
LP       -62.65592 1.91979 0.0577493 1000.00 -67.195015
            Median        UB
beta[1]    5.04048   5.27526
beta[2]    0.57316   1.12600
```

```
beta[3]    1.18230   1.79089
beta[4]    0.88176   1.57237
sigma      0.70556   0.92248
Deviance  82.19656  92.03773
LP       -62.27468 -60.29148


Summary of Stationary Samples
              Mean      SD      MCSE     ESS         LB
beta[1]    5.04337 0.11362 0.0036514 1000.00   4.833182
beta[2]    0.57981 0.27673 0.0091987  887.34   0.076972
beta[3]    1.17592 0.31228 0.0109790  890.90   0.544792
beta[4]    0.89981 0.33047 0.0129093  851.71   0.269623
sigma      0.71506 0.09430 0.0030039 1000.00   0.566679
Deviance  82.95846 3.83904 0.1154811 1000.00  78.229166
LP       -62.65592 1.91979 0.0577493 1000.00 -67.195015
            Median        UB
beta[1]    5.04048   5.27526
beta[2]    0.57316   1.12600
beta[3]    1.18230   1.79089
beta[4]    0.88176   1.57237
sigma      0.70556   0.92248
Deviance  82.19656  92.03773
LP       -62.27468 -60.29148

Demonic Suggestion

Due to the combination of the following conditions,

1. Hit-And-Run Metropolis
2. The acceptance rate (0.23593) is within the interval [0.15,0.5].
3. Each target MCSE is < 6.27% of its marginal posterior
   standard deviation.
4. Each target distribution has an effective sample size (ESS)
   of at least 100.
5. Each target distribution became stationary by
   1 iteration.

Laplace's Demon has been appeased, and suggests
the marginal posterior samples should be plotted
and subjected to any other MCMC diagnostic deemed
fit before using these samples for inference.

Laplace's Demon is finished consorting.
```

Several components are labeled as `NOT SHOWN HERE`, due to their size, such as the covariance matrix `Covar` or the stationary posterior samples `Posterior2`. As usual, these can be printed

to the screen by appending a dollar sign, followed by the desired component, such as:

```
> Fit$Posterior2
```

Although a lot can be learned from the above output, notice that it completed 150000 iterations of 5 variables in 0.26 minutes. Of course this was fast, since there were only 39 records, and the form of the specified model was simple. As discussed later, the `LaplacesDemon` function does better than most other MCMC software with large numbers of records, such as 100,000 (see section 14).

In R, there is usually a `summary` function associated with each class of output object. The `summary` function usually summarizes the output. For example, with frequentist models, the `summary` function usually creates a table of parameter estimates, complete with p-values.

Since this is not a frequentist package, p-values are not part of any table with the `LaplacesDemon` function, and the marginal posterior distributions of the parameters and other variables have already been summarized in `Fit`, there is no point to have an associated `summary` function. Going one more step toward useability, the `Consort` function of **LaplacesDemon** allows the user to consort with Laplace's Demon about the output object.

The additional feature is a second section called `Demonic Suggestion`. The `Demonic Suggestion` is a very helpful section of output. When the **LaplacesDemon** package was developed initially in late 2010, there were not to my knowledge any tools of Bayesian inference that make suggestions to the user.

Before making its `Demonic Suggestion`, Laplace's Demon considers and presents five conditions: the algorithm, acceptance rate, Monte Carlo standard error (MCSE), effective sample size (ESS), and stationarity. In addition to these conditions, there are other suggested values, such as a recommended number of iterations or values for the `Periodicity` and `Status` arguments. The suggested value for `Status` is seeking to print a status message every minute when the expected time is longer than a minute, and is based on the time in minutes it took, the number of iterations, and the recommended number of iterations.

In the above output, Laplace's Demon is appeased. However, if any of these five conditions is unsatisfactory, then Laplace's Demon is not appeased, and suggests it should continue updating, and that the user should copy, paste, and execute its suggested R code. Here are the criteria it measures against. The final algorithm must be non-adaptive, so that the Markov property holds (this is covered in section 12.5.2). The acceptance rate of most algorithms is considered satisfactory if it is within the interval [15%, 50%][10]. LMC or MALA must be in the interval [40%, 80%], and others (AGG, ESS, GG, SGLD, Slice, and UESS) have an acceptance rate of 100%. For more information on acceptance rates, see the `AcceptanceRate` function. MCSE is considered satisfactory for each target distribution if it is less than 6.27% of the standard deviation of the target distribution. This allows the true mean to be within 5% of the area under a Gaussian distribution around the estimated mean. ESS is considered satisfactory for each target distribution if it is at least 100, which is usually enough to describe 95% probability intervals. And finally, each variable must be estimated as stationary.

In this example, notice that all criteria have been met: MCSEs are sufficiently small, ESSs

[10]While Spiegelhalter, Thomas, Best, and Lunn (2003) recommend updating until the acceptance rate is within the interval [20%, 40%], and Roberts and Rosenthal (2001) suggest [10%, 40%], the interval recommended here is [15%,50%]. HMC and Refractive must be in the interval [60%, 70%].

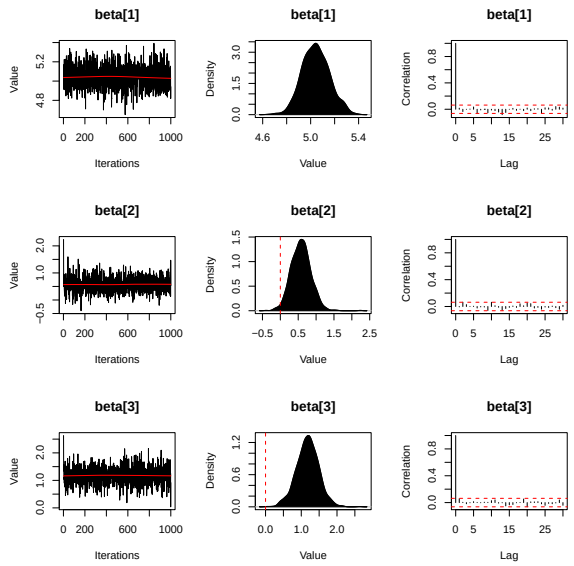

Figure 1: Plots of Marginal Posterior Samples

are sufficiently large, and all parameters were estimated to be stationary. Since the algorithm was the non-adaptive HARM, the Markov property holds, so let's look at some plots.

7. Plotting Output

The **LaplacesDemon** package has a `plot.demonoid` function to enable its own customized plots with `demonoid` objects. The variable `BurnIn` (below) may be left as it is so it will show only the stationary samples (samples that are no longer trending), or set equal to one so that all samples can be plotted. In this case, all thinned samples are plotted: `BurnIn=0`.

The `plot` function also enables the user to specify whether or not the plots should be saved as a .pdf file, and allows the user to select the parameters to be plotted. For example, `Parms=c("beta[1]","beta[2]")` would plot only the first two regression effects, and `Parms=NULL` will plot everything.

```
> plot(Fit, BurnIn=0, MyData, PDF=FALSE, Parms=NULL)
```

There are three plots for each parameter, the deviance, and each monitored variable (which in this example are `LP` and `sigma`). The leftmost plot is a trace-plot, showing the history of the value of the parameter according to the iteration. The middlemost plot is a kernel density plot. The rightmost plot is an ACF or autocorrelation function plot, showing the autocorrelation at different lags. The chains look stationary (do not exhibit a trend), the kernel densities look Gaussian, and the ACF's show low autocorrelation.

The Hellinger distances between batches of chains can be plotted with

```
> plot(BMK.Diagnostic(Fit))
```

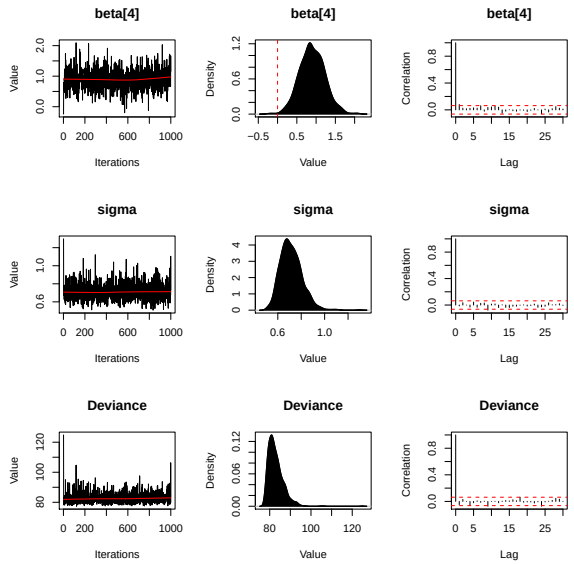

Figure 2: Plots of Marginal Posterior Samples

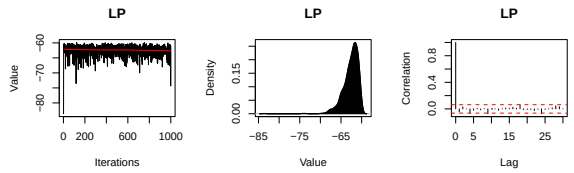

Figure 3: Plots of Marginal Posterior Samples

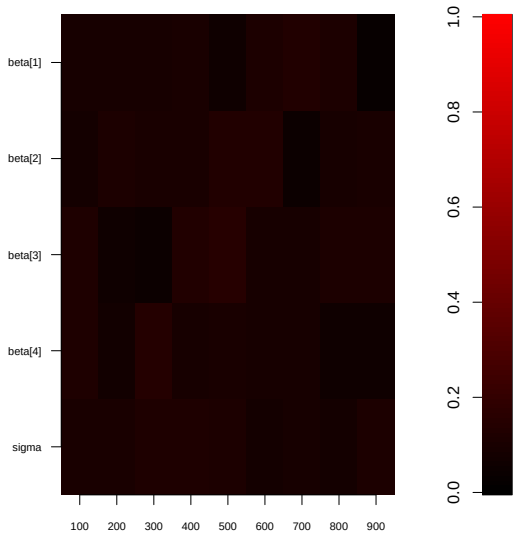

Figure 4: Hellinger Distances

These distances occur in the interval $[0, 1]$, and lower (darker) is better. The `LaplacesDemon` function considers any Hellinger distance greater than 0.5 to indicate non-stationarity and non-convergence. This plot is useful for quickly finding problematic parts of chains. All Hellinger distances here are acceptably small (dark).

Another useful plot is called the caterpillar plot, which plots a horizontal representation of three quantiles (2.5%, 50%, and 97.5%) of each selected parameter from the posterior samples summary. The caterpillar plot will attempt to plot the stationary samples first (`Fit$Summary2`), but if stationary samples do not exist, then it will plot all samples (`Fit$Summary1`). Here, only the first four parameters are selected for a caterpillar plot:

```
> caterpillar.plot(Fit, Parms="beta")
```

If all is well, then the Markov chains should be studied with MCMC diagnostics (such as visual inspections with the `CSF` or Cumulative Sample Function), and finally, further assessments of model fit should be estimated with posterior predictive checks, showing how well (or poorly) the model fits the data. When the user is satisfied, the `BayesFactor` function may be useful in selecting the best model, and the marginal posterior samples may be used for inference.

8. Posterior Predictive Checks

A posterior predictive check is a method to assess discrepancies between the model and the data (Gelman, Meng, and Stern 1996a). To perform posterior predictive checks with the the **LaplacesDemon** package, simply use the `predict` function:

```
> Pred <- predict(Fit, Model, MyData, CPUs=1)
```

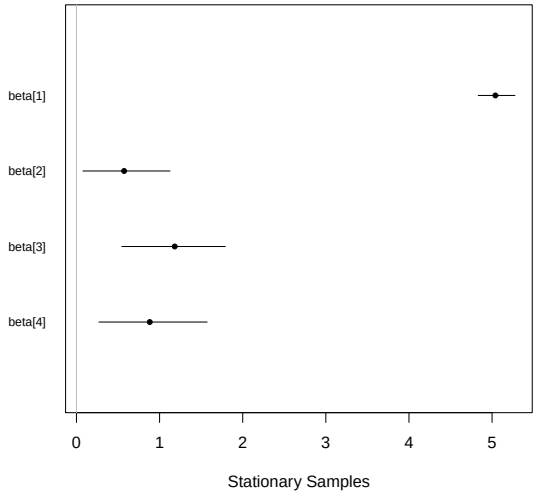

Figure 5: Caterpillar Plot

This creates `Pred`, which is an object of class `demonoid.ppc` (where ppc is short for posterior predictive check). `Pred` is a list that contains three components: `y`, `yhat`, and `Deviance` (though the `LaplaceApproximation` output differs a little). If the data set that was used to estimate the model is supplied in `predict`, then replicates of `y` (also called $\mathbf{y}^{rep}$) are estimated. If, instead, a new data set is supplied in `predict`, then new, unobserved instances of `y` (called $\mathbf{y}^{new}$) are estimated. Note that with new data, a `y` vector must still be supplied, and if unknown, can be set to something sensible such as the mean of the `y` vector in the model.

The `predict` function calls the `Model` function once for each set of stationary samples in `Fit$Posterior2`. When there are few discrepancies between `y` and $\mathbf{y}^{rep}$, the model is considered to fit well to the data. Parallel processing is enabled when multiple CPUs exist and are specified.

Since `Pred$yhat` is a large (39 x 1000) matrix, let's look at the summary of the posterior predictive distribution:

```
> summary(Pred, Discrep="Chi-Square")

Bayesian Predictive Information Criterion:
   Dbar    pD   BPIC
 82.958 7.369 97.696
Concordance:  0.94872
Discrepancy Statistic:  29.34
L-criterion: 37.325, S.L: 0.304
Records:
        y  Mean    SD    LB Median    UB    PQ Discrep
1  4.1744 4.133 0.781 2.627  4.115 5.661 0.468   0.003
2  5.3613 4.959 0.813 3.356  4.969 6.570 0.294   0.244
```

```
3  6.0890 6.126 0.805 4.488  6.120 7.674 0.518   0.002
4  5.2983 4.788 0.793 3.235  4.787 6.303 0.273   0.415
5  4.4067 5.003 0.760 3.383  5.015 6.510 0.795   0.615
6  2.1972 3.672 0.733 2.201  3.680 5.096 0.975   4.050
7  5.0106 4.602 0.745 3.159  4.601 5.999 0.293   0.301
8  1.6094 3.479 0.778 2.047  3.494 5.082 0.992   5.770
9  4.3438 4.666 0.747 3.171  4.683 6.057 0.689   0.186
10 4.8122 3.742 0.767 2.304  3.720 5.293 0.078   1.946
11 4.1897 3.637 0.748 2.218  3.643 5.068 0.220   0.547
12 4.9200 4.458 0.747 2.978  4.433 5.923 0.266   0.382
13 4.7536 4.414 0.747 2.928  4.419 5.865 0.321   0.206
14 4.1271 4.330 0.694 3.015  4.328 5.667 0.608   0.085
15 3.7136 3.373 0.772 1.846  3.372 4.927 0.326   0.195
16 4.6728 4.537 0.753 3.124  4.556 5.986 0.433   0.033
17 6.9305 6.903 0.778 5.288  6.882 8.454 0.482   0.001
18 5.0689 4.159 0.767 2.651  4.184 5.653 0.118   1.407
19 6.7754 6.782 0.744 5.312  6.768 8.228 0.495   0.000
20 6.5539 6.555 0.761 5.072  6.559 7.983 0.506   0.000
21 4.8903 4.518 0.713 3.129  4.521 5.872 0.305   0.273
22 4.4427 4.577 0.758 3.124  4.565 6.083 0.580   0.031
23 2.8332 4.518 0.754 2.918  4.529 5.980 0.983   4.996
24 4.7875 4.322 0.753 2.873  4.322 5.762 0.278   0.382
25 6.9334 6.387 0.744 4.899  6.418 7.840 0.228   0.538
26 6.1800 5.600 0.715 4.232  5.606 7.078 0.192   0.659
27 5.6525 5.539 0.753 4.002  5.533 6.917 0.447   0.023
28 5.4293 5.291 0.757 3.787  5.311 6.770 0.425   0.033
29 5.6348 6.265 0.813 4.642  6.262 7.858 0.792   0.602
30 4.2627 4.140 0.777 2.613  4.136 5.694 0.443   0.025
31 3.8918 4.496 0.782 2.978  4.497 6.104 0.789   0.596
32 6.6134 6.507 0.737 5.094  6.508 7.950 0.429   0.021
33 4.9200 4.451 0.756 2.927  4.457 5.985 0.260   0.385
34 6.5410 6.578 0.782 5.092  6.602 8.060 0.526   0.002
35 6.3456 6.389 0.743 4.894  6.416 7.828 0.538   0.003
36 3.7377 4.782 0.758 3.258  4.784 6.237 0.921   1.900
37 7.3563 7.665 0.791 6.171  7.661 9.166 0.653   0.153
38 5.7398 5.779 0.747 4.352  5.775 7.214 0.519   0.003
39 5.5175 4.425 0.716 3.123  4.413 5.772 0.059   2.327
```

The `summary.demonoid.ppc` function returns a list with 5 components when `y` is continuous (different output occurs for categorical dependent variables when given the argument `Categorical=TRUE`):

- `BPIC` is the Bayesian Predictive Information Criterion of Ando (2007). BPIC is a variation of the Deviance Information Criterion (DIC) that has been modified for predictive distributions. For more information on DIC, see the accompanying vignette entitled "Bayesian Inference".
- `Concordance` is the predictive concordance of Gelfand (1996), that indicates the per-

centage of times that y was within the 95% probability interval of yhat. A goal is to have 95% predictive concordance. For more information, see the accompanying vignette entitled "Bayesian Inference". In this case, roughly 95% of the time, y is within the 95% probability interval of yhat. These results suggest that the model should be attempted again under different conditions, such as using different predictors, or specifying a different form to the model.

- Discrepancy.Statistic is a summary of a specified discrepancy measure. There are many options for discrepancy measures that may be specified in the Discrep argument. In this example, the specified discrepancy measure was the χ^2 test in Gelman, Carlin, Stern, and Rubin (2004, p. 175), and higher values indicate a worse fit.

- L-criterion is a posterior predictive check for model and variable selection that measures the distance between $\mathbf{y}$ and $\mathbf{y}^{rep}$, providing a criterion to be minimized (Laud and Ibrahim 1995).

- The last part of the summarized output reports y, information about the distribution of yhat, and the predictive quantile (PQ). The mean prediction of y[1], or $\mathbf{y}_1^{rep}$, given the model and data, is 4.133. Most importantly, PQ[1] is 0.468, indicating that 46.8% of the time, yhat[1,] was greater than y[1], or that y[1] is close to the mean of yhat[1,]. Contrast this with the 6th record, where y[6]=2.197 and PQ[6]=0.975. Therefore, yhat[6,] was not a good replication of y[6], because the distribution of yhat[6,] is almost always greater than y[6]. While y[1] is within the 95% probability interval of yhat[1,], yhat[6,] is above y[6] 97.5% of the time, indicating a strong discrepancy between the model and data, in this case.

There are also a variety of plots for posterior predictive checks, and the type of plot is controlled with the Style argument. Many styles exist, such as producing plots of covariates and residuals. The last component of this summary may be viewed graphically as posterior densities. Rather than observing plots for each of 39 records or rows, only the first 9 densities will be shown here:

```
> plot(Pred, Style="Density", Rows=1:9)
```

Among many other options, the fit may be observed:

```
> plot(Pred, Style="Fitted")
```

This plot shows a poor fit between the dependent variable and its expectation, and model revision should be considered.

The Importance function is not presented here in detail, but may be a useful way to assess variable importance, which is defined here as the impact of each variable on $\mathbf{y}^{rep}$, when the variable is removed (or set to zero). Variable importance consists of differences in model fit or discrepancy statistics, showing how well the model fits the data with each variable removed. This information may be used for model revision, or presenting the relative importance of variables.

These posterior predictive checks indicate that there is plenty of room to improve this model.

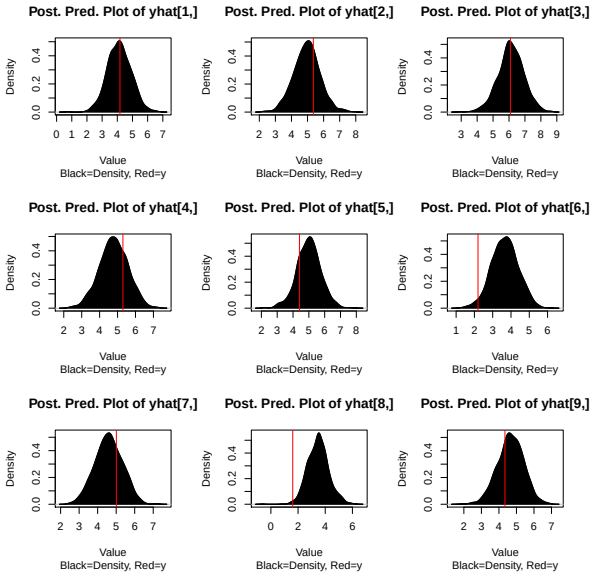

Figure 6: Posterior Predictive Densities

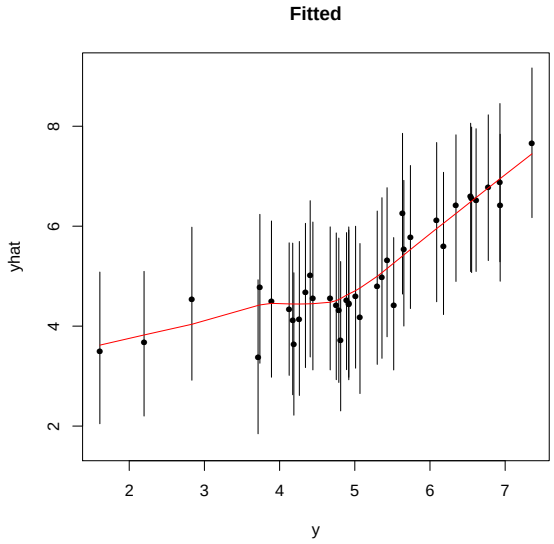

Figure 7: Posterior Predictive Fit

9. General Suggestions

Following are general suggestions on how best to use the **LaplacesDemon** package:

- As suggested by Gelman (2008), continuous predictors should be centered and scaled. Here is an explicit example in R of how to center and scale a single predictor called `x`: `x.cs <- (x - mean(x)) / (2*sd(x))`. However, it is instead easier to use the `CenterScale` function provided in **LaplacesDemon**.

- Do not forget to reparameterize any bounded parameters in the `Model` function to be real-valued in the `parm` vector, and this is a good time to check for prior propriety with the `is.proper` function.

- If sufficient sample size is available, begin with a deterministic numerical approximation algorithm such as Laplace Approximation or variational Bayes.

- MCMC and PMC are stochastic methods of numerical approximation, and as such, results may differ with each run due to the use of pseudo-random number generation. It is good practice to set a seed so that each update of the model may be reproduced. Here is an example in R: `set.seed(666)`.

- Rather than specify the final, intended model in the `Model` function, start by specifying the simplest possible form. Rather than beginning with actual data, start by simulating data given specified parameters. Update the simple model on simulated data and verify that the algorithm converges to the correct target distributions. One by one, add components to the model specification, simulate more complicated data, update, verify, and progress toward the intended model. If using MCMC during this phase, then use the `Juxtapose` function to compare the inefficiency of several MCMC algorithms (via integrated autocorrelation time or `IAT`), and use this information to select the least inefficient algorithm for your particular model. When confident the model is specified correctly and with informed algorithmic selection, finally use actual data, but with few iterations, such as `Iterations=20`.

- After studying MCMC updates with few iterations, the first "actual" update should be long enough that proposals are accepted (the acceptance rate is not zero), adaptation begins to occur (if used), and that enough iterations occur after the first adaptation to allow the user to study the adaptation (assuming an adaptive algorithm is used). In the supplied example, the HARM algorithm is non-adaptive, so this is not a consideration.

- Depending on the model specification function, data, and intended iterations, it is a good idea to use the `LaplacesDemon.RAM` function to estimate the amount of random-access memory (RAM) that `LaplacesDemon` will use. If `LaplacesDemon` uses more RAM than the computer has available, then the computer will crash. This can be used to estimate the maximum number of iterations or thinned samples for a particular model and data set on a given computer.

- Once the final, intended model has begun (finally!), the mixing of the chains should be observed after a larger trial run, say, arbitrarily, for 10,000 iterations. If the chains do not mix as expected, then try a different algorithm, either one suggested by the

`Consort` function (such as when diminishing adaptation is violated), or use the next least inefficient algorithm as indicated previously in the `Juxtapose` function.

- When speed is a concern, such as with complex models, there may be things in the `Model` function that can be commented out, such as sometimes calculating `yhat`. The model can be updated without some features, that can be un-commented and used for posterior predictive checks. By commenting out things that are strictly unnecessary to updating, the model will update more quickly. Other helpful hints for speed are found in the documentation for the `Model.Spec.Time` function.

- If the numerical approximation algorithm is exploring areas of the state space that the user knows *a priori* should not be explored, then the parameters may be constrained in the `Model` function before being passed back to the numerical approximation function. Simply change the parameter of interest as appropriate and place the constrained value back in the `parm` vector.

- For MCMC, `Demonic Suggestion` is intended as an aid, not an infallible replacement for critical thinking. As with anything else, its suggestions are based on assumptions, and it is the responsibility of the user to check those assumptions. For example, the `BMK.Diagnostic` may indicate stationarity (lack of a trend) when it does not exist. Or, the `Demonic Suggestion` may indicate that the next update may need to run for a million iterations in a complex model, requiring weeks to complete.

- If an adaptive MCMC algorithm is used, then use a two-phase approach, where the first phase consists of using an adaptive algorithm to achieve stationary samples that seem to have converged to the target distributions (convergence can never be determined with MCMC, but some instances of non-convergence can be observed). Once it is believed that convergence has occurred, use a non-adaptive algorithm. The final samples should again be checked for signs of non-convergence. If satisfactory, then the non-adaptive algorithm should have estimated the logarithm of the marginal likelihood (LML). This is most easily checked with the `is.proper` function, which considers the joint posterior distribution to be proper if it can verify that the LML is finite.

- The desirable number of final, thinned samples for inference depends on the required precision of the inferential goal. A good, general goal is to end up with 1,000 thinned samples (Gelman *et al.* 2004, p. 295), where the ESS is at least 100 (and more is desirable). See the `ESS` function for more information.

- Disagreement exists in MCMC literature as to whether to update one, long chain (Geyer 1992, 2011), or multiple, long chains with different, randomized initial values (Gelman and Rubin 1992). Multiple chains are enabled with an extension function called `LaplacesDemon.hpc`, which uses parallel processing. The `Gelman.Diagnostic` function may be used to compare multiple chains. Samples from multiple chains may be put together with the `Combine` function.

- After a deterministic numerical approximation algorithm has converged, consider following it up with a stochastic numerical approximation algorithm such as MCMC, if practical. When MCMC seems to have converged, consider updating the model again, this time with Population Monte Carlo (PMC). PMC may improve the model fit obtained

with MCMC, and should reduce the variance of the marginal posterior distributions, which is desirable for predictive modeling.

- After a model has been updated, consider posterior predictive checks and any necessary model revisions. Afterward, consider updating a model with different prior distributions and compare results with the `BayesFactor` and `SensitivityAnalysis` functions, as well as comparing posterior predictive checks. Consider applying the model to different data sets and using the `Validate` function. Consider beyond the model how decision theory applies to the problem. Finally, make inferences, given the model and data.

10. Independence and Observability

The **LaplacesDemon** package was designed with independence and observability in mind. By independence, it is meant that a goal was to minimize dependence on other software. The **LaplacesDemon** package requires only base R, and the parallel package bundled with it. The variety of packages makes R extremely attractive. However, depending on multiple packages can be problematic when different packages have functions with the same name, or when a change is made in one package, but other packages do not keep pace, and the user is dependent on packages being in sync. By avoiding dependencies on packages that are not in or accompanying base R, the **LaplacesDemon** package is attempting to be consistent and dependable for the user.

For example, common MCMC diagnostics and probability distributions (such as Dirichlet, multivariate normal, Wishart, and many others, as well as truncated forms of distributions) in Bayesian inference have been included in the **LaplacesDemon** package so the user does not have to load numerous R packages, except of course for exotic distributions that have not been included.

By observability, it is meant that the **LaplacesDemon** package is written entirely in R. Certain functions could be sped up in another language, but this may prevent some R users from understanding the code. The **LaplacesDemon** package is intended to be open and accessible. If a user desires speed and is familiar with a faster language, then the user is encouraged to program the model specification function in the faster language. See the documentation for the `Model.Spec.Time` function for more information. Moreover, it is demonstrated in section 14 that the `LaplacesDemon` function for MCMC is often significantly faster than other MCMC software programmed in faster languages, and users are encouraged to time comparisons, especially with large samples.

Observability also enables users to investigate or customize functions in the **LaplacesDemon** package. To access any function, simply enter the function name and press enter. For example, to print the source code for the `LaplacesDemon` function to the R console, simply enter:

```
> LaplacesDemon
```

To access undocumented, internal-only functions, use the `:::` operator, such as:

```
> LaplacesDemon:::MWG
```

LaplacesDemon seeks to provide a complete, Bayesian environment within R. Independence from other software facilitates dependability, and its open code makes it easier for a user to investigate and customize.

11. High Performance Computing

High performance computing (HPC) is a broad term that can mean many different things. The **LaplacesDemon** package currently uses the term HPC to refer to two topics: big data and parallel processing.

11.1. Big Data

There are several definitions for big data. Here, big data is defined as data that is too big for the computer memory (RAM). The `BigData` function enables updating a Bayesian model with big data by reading in and processing smaller batches or chunks of data and performing a user-specified function on the batch before combining and outputing the result, so the entire data set does not consume RAM. `BigData` is also parallelized. The `read.matrix` function allows sampling from big data. Finally, the Stochastic Gradient Descent (SGD) algorithm (see 12.4.16) in `LaplaceApproximation` and the Stochastic Gradient Langevin Dynamics (SGLD) algorithm (see 12.5.37) in `LaplacesDemon` are designed specifically for use with big data.

11.2. Parallel Processing

Parallel processing occurs when software is designed to simultaneously use multiple central processing units (CPUs). The motherboard of a computer may contain multiple CPUs, such as a quad-core contains four, and this is called a multicore computer. Several computers may be linked together with network communication, forming what is called a computer cluster. The **LaplacesDemon** package has several functions that optionally take advantage of multicore computers or may utilize large computer clusters.

In the context of MCMC, there are three approaches to parallelization that are avilable in **LaplacesDemon**: parallel approximation within a chain, parallel sets of independent chains, and parallel sets of interactive chains. There are more parallelized functions in **LaplacesDemon** in addition to MCMC.

11.3. Iterative Quadrature

The `IterativeQuadrature` function provides several numerical integration algorithms, and each may be parallelized. At each iteration, the conditional density is evaluated at several nodes, and this processing may take advantage of multiple CPUs. For more information, see 12.3.

11.4. Parallel Approximation within a Chain

The Griddy-Gibbs (GG) sampler of Ritter and Tanner (1992), Adaptive Griddy-Gibbs (AGG), and Multiple-Try Metropolis (MTM) of Liu, Liang, and Wong (2000) are examples of algorithms in which an approximation is made within a chain, and the approximation may be parallelized. For more information, see 12.5.16 or 12.5.23.

11.5. Parallel Sets of Independent Chains

The `LaplacesDemon` function is extended with the `LaplacesDemon.hpc` function for the parallel processing of multiple chains on different central processing units (CPUs). This requires a minimum of two additional arguments: `Chains` to specify the number of parallel chains, and `CPUs` to specify the number of CPUs. The `LaplacesDemon.hpc` function allows the parallelization of most MCMC algorithms in the `LaplacesDemon` function.

An example of using `LaplacesDemon.hpc` is to simultaneously update three independent chains as an aid to checking MCMC convergence, as Gelman recommends (Gelman and Rubin 1992). Aside from aiding convergence, another benefit of parallelization is that more posterior samples are updated in the same time-frame as a non-parallel implementation. A multicore computer, such as a quad-core, will yield more posterior samples (which is valuable only if it converges, because it does not process more iterations), but a supercomputing environment or large computer cluster will yield many orders more. If multiple CPUs are available, then it only makes sense to use them...all.

It is important to note that `Status` messages do not print to the console during parallel processing with `LaplacesDemon.hpc`, and should alternately be directed by the user to a log file with the `LogFile` argument, if desired. The `LaplacesDemon.hpc` function sends the information associated with each chain as well as the `LaplacesDemon` function to each CPU. The `LaplacesDemon` function may very well return status messages, but the `LaplacesDemon.hpc` function is unaware.

After updating a model with `LaplacesDemon.hpc`, the `plot` function may be applied so that multiple chains may be viewed simultaneously, and this is helpful when comparing samplers for a specific model. If this looks good, then the `Gelman.Diagnostic` function may be applied to assess convergence. Otherwise, the `as.initial.values` function may be used to extract the latest values from the chains and use these to begin the next update. Once results seem acceptable, the `Combine` function may be used to combine the posterior samples of multiple chains into one `demonoid` object, from which the remaining facilities of the **LaplacesDemon** package are available.

The Metropolis-Coupled Markov Chain Monte Carlo (MCMCMC) algorithm of Geyer (1991) is an example of an MCMC algorithm in which multiple chains are updated in parallel, but in `LaplacesDemon`, not `LaplacesDemon.hpc`.

11.6. Parallel Sets of Interactive Chains

Parallel sets of independent chains should each run as efficiently as a traditional single set of chains. However, independent chains cannot benefit from the fact that there are other chains, while each chain is running. They are independent of each other.

In contrast, parallel sets of interactive chains are able to learn from each other through interaction. In the **LaplacesDemon** package, some of these algorithms are called with the `LaplacesDemon` function, and some with the `LaplacesDemon.hpc` function.

The Interchain Adaptation (INCA) algorithm (Craiu, Rosenthal, and Yang 2009; Solonen, Ollinaho, Laine, Haario, Tamminen, and Jarvinen 2012) performs Adaptive Metropolis (AM) with parallel chains that share the adaptive component, and this sharing speeds convergence. Whenever the chains are specified to adapt, adaptation is performed by pooling the historical covariance matrix across all parallel chains, and then returns the combined result to all chains.

Network communication time slows the adaptation, but once returned to each CPU, chains iterate at their usual speed. This algorithm must be used with the `LaplacesDemon.hpc` function, and there is not an un-parallelized form of it. For more information, see 12.5.21.

The Affine-Invariant Ensemble Sampler (AIES) of (Goodman and Weare 2010) must be used with the `LaplacesDemon` function, and is available in either a parallelized or un-parallelized form. A large, even number of parallel chains (or walkers) are grouped into two batches, and each iteration, each chain moves in relation to a randomly selected chain (walker) in the other batch. Since these interactive chains interact each iteration, computer network communication is frequent, and this communication may be much slower than processing with one CPU. However, in a large-scale computing environment and when a `Model` function is not trivial to evaluate, this form of parallelization can result in very early convergence.

11.7. Population Monte Carlo

The `PMC` function has been parallelized at each iteration to speed up the evaluation of the model specification function over numerous importance samples.

11.8. Predict Functions

The predict functions (`predict.demonoid`, `predict.laplace`, `predict.pmc`) have been parallelized to speed up the prediction, or scoring, of larger data sets or when models have many posterior samples. The `Importance` function, which extensively uses predict functions, has also been parallelized.

11.9. Model Specification Function

A user may have a model with a model specification function that is computationally expensive, and may write their own parallelization code to speed up its processing by breaking down challenging computations and sending them to separate CPUs.

11.10. Parallelization Details

Parallelization is enabled by the **parallel** package that comes with base R. Parallelization is accomplished by default with socket-transport functions derived from the **snow** package, which is an acronym for a Simple Network of Workstations. Alternatively, Message Passing Interface (MPI) may be used. SNOW is more general, being cross-platform, and works on multicore computers, computer clusters, and supercomputers. More performance may be found with MPI, but it is more specialized. `LaplacesDemon.hpc` was reported to have been used successfully on a cluster with over 200 nodes.

12. Details

The **LaplacesDemon** package uses five broad types of numerical approximation algorithms: Importance Sampling (IS), Iterative Quadrature, Laplace Approximation, Markov chain Monte Carlo (MCMC), and Variational Bayes (VB). Approximate Bayesian Computation (ABC) may be estimated within each. These numerical approximation algorithms are introduced below.

12.1. Approximate Bayesian Computation

Approximate Bayesian Computation (ABC), also called likelihood-free estimation, is a family of numerical approximation techniques in Bayesian inference. ABC is especially useful when evaluation of the likelihood, $p(\mathbf{y}|\Theta)$ is computationally prohibitive, or when suitable likelihoods are unavailable. As such, ABC algorithms estimate likelihood-free approximations. ABC is usually faster than a similar likelihood-based numerical approximation technique, because the likelihood is not evaluated directly, but replaced with an approximation that is usually easier to calculate. The approximation of a likelihood is usually estimated with a measure of distance between the observed sample, $\mathbf{y}$, and its replicate given the model, $\mathbf{y}^{rep}$, or with summary statistics of the observed and replicated samples. See the accompanying vignette entitled "Examples" for an example.

12.2. Importance Sampling

Importance Sampling (IS) is a method of estimating a distribution with samples from a different distribution, called the importance distribution. Importance weights are assigned to each sample. The main difficulty with IS is in the selection of the importance distribution. IS dates back at least to the 1950s, including iterative IS. IS is the basis of a wide variety of algorithms, some of which involve the combination of IS and Markov chain Monte Carlo (MCMC). There are also many variations of IS, including adaptive IS, and parametric and nonparametric self-normalized IS (SNIS). Some popular algorithms, or families of algorithms, that include IS are Particle Filtering, Population Monte Carlo (PMC), and Sequential Monte Carlo (SMC).

Population Monte Carlo

Population Monte Carlo (PMC) uses adaptive IS, and the proposal or importance distribution is a multivariate Gaussian (Cappe, Guillin, Marin, and Robert 2004), or a mixture of multivariate Gaussian distributions (Cappe, Douc, Guillin, Marin, and Robert 2008; Wraith, Kilbinger, Benabed, Cappé, Cardoso, Fort, Prunet, and Robert 2009). **LaplacesDemon** uses the version presented in the appendix of Wraith *et al.* (2009). At each iteration, the importance distribution of N samples and M mixture components is adapted. Parallel processing is available.

Compared with Markov chain Monte Carlo (MCMC), very few iterations are required, convergence and ergodicity are not problems, posterior samples are independent, and PMC lends itself well to parallelization. However, PMC requires much more prior information about the model (better initial values and proposal covariance matrix) than MCMC, and becomes harder to apply as the number of variables increases.

Amazingly, PMC may improve the model fit obtained with MCMC, and should reduce the variance of the marginal posterior distributions. This reduction in variance is desirable for predictive modeling. Therefore, it is recommended that a model is attempted to be updated with PMC after the model seems to have converged with MCMC.

12.3. Iterative Quadrature

Quadrature is a historical term in mathematics that means determining area. Mathematicians of ancient Greece, according to the Pythagorean doctrine, understood determination of area of

a figure as the process of geometrically constructing a square having the same area (squaring). Thus the name quadrature for this process.

In medieval Europe, quadrature meant the calculation of area by any method. With the invention of integral calculus, quadrature has been applied to the computation of a univariate definite integral. Numerical integration is a broad family of algorithms for calculating the numerical value of a definite integral. Numerical quadrature is a synonym for quadrature applied to one-dimensional integrals. Multivariate quadrature, also called cubature, is the application of quadrature to multidimensional integrals.

A quadrature rule is an approximation of the definite integral of a function, usually stated as a weighted sum of function values at specified points within the domain of integration. The specified points are referred to as abscissae, abscissas, integration points, or nodes, and have associated weights. The calculation of the nodes and weights of the quadrature rule differs by the type of quadrature. There are numerous types of quadrature algorithms. Bayesian forms of quadrature usually use Gauss-Hermite quadrature (Naylor and Smith 1982), and placing a Gaussian Process on the function is a common extension (O'Hagan 1991; Rasmussen and Ghahramani 2003) that is called 'Bayesian Quadrature'. Often, these and other forms of quadrature are also referred to as model-based integration.

Gauss-Hermite quadrature uses Hermite polynomials to calculate the rule. However, there are two versions of Hermite polynomials, which result in different kernels in different fields. In physics, the kernel is $\exp(-x^2)$, while in probability the kernel is $\exp(-x^2/2)$. The weights are a normal density. If the parameters of the normal distribution, μ and σ^2, are estimated from data, then it is referred to as adaptive Gauss-Hermite quadrature, and the parameters are the conditional mean and conditional variance. Outside of Gauss-Hermite quadrature, adaptive quadrature implies that a difficult range in the integrand is subdivided with more points until it is well-approximated. Gauss-Hermite quadrature performs well when the integrand is smooth, and assumes normality or multivariate normality. Adaptive Gauss-Hermite quadrature has been demonstrated to outperform Gauss-Hermite quadrature in speed and accuracy.

A goal in quadrature is to minimize integration error, which is the error between the evaluations and the weights of the rule. Therefore, a goal in Bayesian Gauss-Hermite quadrature is to minimize integration error while approximating a marginal posterior distribution that is assumed to be smooth and normally-distributed. This minimization often occurs by increasing the number of nodes until a change in mean integration error is below a tolerance, rather than minimizing integration error itself, since the target may be only approximately normally distributed, or minimizing the sum of integration error, which would change with the number of nodes.

To approximate integrals in multiple dimensions, one approach applies N nodes of a univariate quadrature rule to multiple dimensions (using the `GaussHermiteCubeRule` function for example) via the product rule, which results in many more multivariate nodes. This requires the number of function evaluations to grow exponentially as dimension increases. Multidimensional quadrature is usually limited to less than ten dimensions, both due to the number of nodes required, and because the accuracy of multidimensional quadrature algorithms decreases as the dimension increases. Three methods may overcome this curse of dimensionality in varying degrees: componentwise quadrature, sparse grids, and Monte Carlo.

Componentwise quadrature is the iterative application of univariate quadrature to each pa-

rameter. It is applicable with high-dimensional models, but sacrifices the ability to calculate the conditional covariance matrix, and calculates only the variance of each parameter.

Sparse grids were originally developed by Smolyak for multidimensional quadrature. A sparse grid is based on a one-dimensional quadrature rule. Only a subset of the nodes from the product rule is included, and the weights are appropriately rescaled. Although a sparse grid is more efficient because it reduces the number of nodes to achieve the same accuracy, the user must contend with increasing the accuracy of the grid, and it remains inapplicable to high-dimensional integrals.

Monte Carlo is a large family of sampling-based algorithms. O'Hagan (1987) asserts that Monte Carlo is frequentist, inefficient, regards irrelevant information, and disregards relevant information. Quadrature, he maintains (O'Hagan 1992), is the most Bayesian approach, and also the most efficient. In high dimensions, he concedes, a popular subset of Monte Carlo algorithms is currently the best for cheap model function evaluations. These algorithms are called Markov chain Monte Carlo (MCMC). High-dimensional models with expensive model evaluation functions, however, are not well-suited to MCMC. A large number of MCMC algorithms is available in the `LaplacesDemon` function.

Following are some reasons to consider iterative quadrature rather than MCMC. Once an MCMC sampler finds equilibrium, it must then draw enough samples to represent all targets. Iterative quadrature does not need to continue drawing samples. Multivariate quadrature is consistently reported as more efficient than MCMC when its assumptions hold, though multivariate quadrature is limited to small dimensions. High-dimensional models therefore default to MCMC, between the two. Componentwise quadrature algorithms like CAGH, however, may also be more efficient with cloc-time than MCMC in high dimensions, especially against componentwise MCMC algorithms. Another reason to consider iterative quadrature are that assessing convergence in MCMC is a difficult topic, but not for iterative quadrature. A user of iterative quadrature does not have to contend with effective sample size and autocorrelation, assessing stationarity, acceptance rates, diminishing adaptation, etc. Stochastic sampling in MCMC is less efficient when samples occur in close proximity (such as when highly autocorrelated), whereas in quadrature the nodes are spread out by design.

In general, the conditional means and conditional variances progress smoothly to the target in multidimensional quadrature. For componentwise quadrature, movement to the target is not smooth, and often resembles a Markov chain or optimization algorithm.

Iterative quadrature is often applied after `LaplaceApproximation` to obtain a more reliable estimate of parameter variance or covariance than the negative inverse of the `Hessian` matrix of second derivatives, which is suitable only when the contours of the logarithm of the unnormalized joint posterior density are approximately ellipsoidal (Naylor and Smith 1982).

Adaptive Gauss-Hermite

The Adaptive Gauss-Hermite (AGH) algorithm is the Naylor and Smith (1982) algorithm. The AGH algorithm uses multivariate quadrature with the physicist's (not the probabilist's) kernel.

There are four algorithm specifications: `N` is the number of univariate nodes, `Nmax` is the maximum number of univariate nodes, `Packages` accepts any package required for the model function when parallelized, and `Dyn.libs` accepts dynamic libraries for parallelization, if required. The number of univariate nodes begins at N and increases by one each iteration.

The number of multivariate nodes grows quickly with N. Naylor and Smith (1982) recommend beginning with as few nodes as $N = 3$. Any of the following events will cause N to increase by 1 when N is less than `Nmax`:

- All LP weights are zero (and non-finite weights are set to zero)
- μ does not result in an increase in LP
- All elements in Σ are not finite
- The square root of the sum of the squared changes in μ is less than or equal to the `Stop.Tolerance`

Tolerance includes two metrics: change in mean integration error and change in parameters. Including the change in parameters for tolerance was not mentioned in Naylor and Smith (1982).

Naylor and Smith (1982) consider a transformation due to correlation. This is not included here.

The AGH algorithm does not currently handle constrained parameters, such as with the `interval` function. If a parameter is constrained and changes during a model evaluation, this changes the node and the multivariate weight. This is currently not corrected.

An advantage of AGH over componentwise adaptive quadrature is that AGH estimates covariance, where a componentwise algorithm ignores it. A disadvantage of AGH over a componentwise algorithm is that the number of nodes increases so quickly with dimension, that AGH is limited to small-dimensional models.

Adaptive Gauss-Hermite Sparse Grid

The Adaptive Gauss-Hermite Sparse Grid (AGHSG) algorithm is the Naylor and Smith (1982) algorithm applied to a sparse grid, rather than a traditional multivariate quadrature rule. This is identical to the AGH algorithm above, except that a sparse grid replaces the multivariate quadrature rule.

The sparse grid reduces the number of nodes. The cost of reducing the number of nodes is that the user must consider the accuracy, K.

There are four algorithm specifications: `K` is the accuracy (as a positive integer), `Kmax` is the maximum accuracy, `Packages` accepts any package required for the model function when parallelized, and `Dyn.libs` accepts dynamic libraries for parallelization, if required. These arguments represent accuracy rather than the number of univariate nodes, but otherwise are similar to the AGH algorithm.

Componentwise Adaptive Gauss-Hermite

The Componentwise Adaptive Gauss-Hermite (CAGH) algorithm is a componentwise version of the adaptive Gauss-Hermite quadrature of Naylor and Smith (1982). Each iteration, each marginal posterior distribution is approximated sequentially, in a random order, with univariate quadrature. The conditional mean and conditional variance are also approximated each iteration, making it an adaptive algorithm.

There are four algorithm specifications: `N` is the number of nodes, `Nmax` is the maximum number of nodes, `Packages` accepts any package required for the model function when parallelized, and `Dyn.libs` accepts dynamic libraries for parallelization, if required. The number of nodes begins at N. All parameters have the same number of nodes. Any of the following events will cause N to increase by 1 when N is less than `Nmax`, and these conditions refer to all parameters (not individually):

- Any LP weights are not finite
- All LP weights are zero
- μ does not result in an increase in LP
- The square root of the sum of the squared changes in μ is less than or equal to the `Stop.Tolerance`

It is recommended to begin with `N=3` and set `Nmax` between 10 and 100. As long as CAGH does not experience problematic weights, and as long as CAGH is improving LP with μ, the number of nodes does not increase. When CAGH becomes either universally problematic or universally stable, then N slowly increases until the sum of both the mean integration error and the sum of the squared changes in μ is less than the `Stop.Tolerance` for two consecutive iterations.

If the highest LP occurs at the lowest or highest node, then the value at that node becomes the conditional mean, rather than calculating it from all weighted samples; this facilitates movement when the current integral is poorly centered toward a well-centered integral. If all weights are zero, then a random proposal is generated with a small variance.

Tolerance includes two metrics: change in mean integration error and change in parameters, as the square root of the sum of the squared differences.

When a parameter constraint is encountered, the node and weight of the quadrature rule is recalculated.

An advantage of CAGH over multidimensional adaptive quadrature is that CAGH may be applied in large dimensions. Disadvantages of CAGH are that only variance, not covariance, is estimated, and ignoring covariance may be problematic.

12.4. Laplace Approximation

The Laplace Approximation or Laplace Method is a family of asymptotic techniques used to approximate integrals. Laplace's method seems to accurately approximate unimodal posterior moments and marginal posterior distributions in many cases. Since it is not applicable in all cases, it is recommended here that Laplace Approximation is used cautiously in its own right, or preferably, it is used before MCMC.

After introducing the Laplace Approximation (Laplace 1774, p. 366–367), a proof was published later (Laplace 1814) as part of a mathematical system of inductive reasoning based on probability. Laplace used this method to approximate posterior moments.

Since its introduction, the Laplace Approximation has been applied successfully in many disciplines. In the 1980s, the Laplace Approximation experienced renewed interest, especially in statistics, and some improvements in its implementation were introduced (Tierney

and Kadane 1986; Tierney, Kass, and Kadane 1989). Only since the 1980s has the Laplace Approximation been seriously considered by statisticians in practical applications.

There are many variations of Laplace Approximation, with an effort toward replacing Markov chain Monte Carlo (MCMC) algorithms as the dominant form of numerical approximation in Bayesian inference. The run-time of Laplace Approximation is a little longer than Maximum Likelihood Estimation (MLE), usually shorter than variational Bayes, and much shorter than MCMC (Azevedo-Filho and Shachter 1994).

The speed of Laplace Approximation depends on the optimization algorithm selected, and typically involves many evaluations of the objective function per iteration (where an MCMC algorithm with a multivariate proposal usually evaluates once per iteration), making many MCMC algorithms faster per iteration. The attractiveness of Laplace Approximation is that it typically improves the objective function better than iterative quadrature, MCMC, and PMC when the parameters are in low-probability regions. Laplace Approximation is also typically faster than MCMC and PMC because it is seeking point-estimates, rather than attempting to represent the target distribution with enough simulation draws. Laplace Approximation extends MLE, but shares similar limitations, such as its asymptotic nature with respect to sample size and that marginal posterior distributions are Gaussian. Bernardo and Smith (2000) note that Laplace Approximation is an attractive family of numerical approximation algorithms, and will continue to develop.

`LaplaceApproximation` seeks a global maximum of the logarithm of the unnormalized joint posterior density. The approach differs by `Method`. The `LaplacesDemon` function uses the `LaplaceApproximation` algorithm to optimize initial values and save time for the user.

Most optimization algorithms assume that the logarithm of the unnormalized joint posterior density is defined and differentiable[11]. Some methods calculate an approximate gradient for each initial value as the difference in the logarithm of the unnormalized joint posterior density due to a slight increase in the parameter.

The user may select from numerous optimization algorithms:

Adaptive Gradient Ascent

With adaptive gradient ascent, the direction and distance for each parameter is proposed based on an approximate truncated graident and an adaptive step size. The step size parameter, which is often plural and called rate parameters in other literature, is adapted each iteration with the univariate version of the Robbins-Monro stochastic approximation in Garthwaite, Fan, and Sisson (2010). The step size shrinks when a proposal is rejected and expands when a proposal is accepted.

Gradient ascent is criticized for sometimes being relatively slow when close to the maximum, and its asymptotic rate of convergence is inferior to other methods. However, compared to other popular optimization algorithms such as Newton-Raphson, an advantage of the gradient ascent is that it works in infinite dimensions, requiring only sufficient computer memory. Although Newton-Raphson converges in fewer iterations, calculating the inverse of the negative Hessian matrix of second-derivatives is more computationally expensive and subject to singularities. Therefore, gradient ascent takes longer to converge, but is more generalizable.

[11] When the joint posterior is not differentiable, and should be, it has probably encountered an area of flat density. It is recommended that WIPs are used for regularization. For more information on WIPs, see the accompanying vignette entitled "Bayesian Inference".

BFGS

The Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm was proposed independently by Broyden (1970), **?**, Goldfarb (1970), and Shanno (1970). BFGS may be the most efficient and popular quasi-Newton optimiziation algorithm. As a quasi-Newton algorithm, the Hessian matrix is approximated using rank-one updates specified by (approximate) gradient evaluations. Since BFGS is very popular, there are many variations of it. This is a version by Nash that has been adapted from the Rvmmin package, and is used in the `optim` function of base R. The approximate Hessian is not guaranteed to converge to the Hessian. When BFGS is used, the approximate Hessian is not used to calculate the final covariance matrix.

BHHH

The BHHH algorithm of Berndt, Hall, Hall, and Hausman (1974) is a quasi-Newton method that includes a step-size parameter, partial derivatives, and an approximation of a covariance matrix that is calculated as the inverse of the sum of the outer product of the gradient (OPG), calculated from each record. The OPG method becomes more costly with data sets with more records. Since partial derivatives must be calculated per record of data, the list of data has special requirements with this method, and must include design matrix $\mathbf{X}$, and dependent variable $\mathbf{y}$ or $\mathbf{Y}$. Records must be row-wise. An advantage of BHHH over NR (see below) is that the covariance matrix is necessarily positive definite, and gauranteed to provide an increase in LP each iteration (given a small enough step-size), even in convex areas. The covariance matrix is better approximated with larger data sample sizes, and when closer to the maximum of LP. Disadvantages of BHHH include that it can give small increases in LP, especially when far from the maximum or when LP is highly non-quadratic.

Conjugate Gradient

Conjugate gradient (CG) is a family of algorithms that uses partial derivatives, but does not use the Hessian matrix or any approximation of it. CG usually requires more iterations to reach convergence than other algorithms that use the Hessian or an approximation. However, since the Hessian becomes computationally expensive as the dimension of the model grows, CG is applicable to large dimensional models. CG was originally developed by Hestenes and Stiefel (1952). The version here is a nonlinear CG method.

Davidon-Fletcher-Powell

The Davidon-Fletcher-Powell (DFP) algorithm was the first popular, multidimensional, quasi-Newton optimization algorithm. The DFP update of an approximate Hessian matrix maintains symmetry and positive-definiteness. The approximate Hessian is not guaranteed to converge to the Hessian. When DFP is used, the approximate Hessian is not used to calculate the final covariance matrix. Although DFP is very effective, it was superseded by the BFGS algorithm.

Hit-And-Run

This version of the Hit-And-Run (HAR) algorithm makes multivariate proposals and uses an adpative length parameter. The length parameter is adapted each iteration with the univariate version of the Robbins-Monro stochastic approximation in Garthwaite *et al.* (2010). The

length shrinks when a proposal is rejected and expands when a proposal is accepted. This is the same algorithm as the HARM or Hit-And-Run Metropolis MCMC algorithm with adaptive length, except that a Metropolis step is not used.

Hooke-Jeeves

The Hooke-Jeeves algorithm (Hooke and Jeeves 1961) is a derivative-free, direct search method. Each iteration involves two steps: an exploratory move and a pattern move. The exploratory move explores local behavior, and the pattern move takes advantage of pattern direction. It is sometimes described as a hill-climbing algorithm. If the solution improves, it accepts the move, and otherwise rejects it. Step size decreases with each iteration. The decreasing step size can trap it in local maxima, where it gets stuck and convergences erroneously. Users are encouraged to attempt again after what seems to be convergence, starting from the latest point. Although getting stuck at local maxima can be problematic, the Hooke-Jeeves algorithm is also attractive because it is simple, fast, does not depend on derivatives, and is otherwise relatively robust.

Levenberg-Marquardt

Also known as the Levenberg-Marquardt Algorithm (LMA) or the Damped Least-Squares (DLS) method, Levenberg-Marquardt (LM) is a trust region (not to be confused with TR below) optimization algorithm that minimizes nonlinear least squares, and has been adapted here to maximize LP. LM uses partial derivatives and approximates the Hessian with outer-products. It is suitable for nonlinear optimization up to a few hundred parameters, but loses its efficiency in larger problems due to matrix inversion. LM is considered between the Gauss-Newton algorithm and gradient descent. When far from the solution, LM moves slowly like gradient descent, but is guaranteed to converge. When LM is close to the solution, LM becomes a damped Gauss-Newton method.

Limited-Memory BFGS

The limited-memory BFGS (Broyden-Fletcher-Goldfarb-Shanno) algorithm is a quasi-Newton optimization algorithm that compactly approximates the Hessian matrix. Rather than storing the dense Hessian matrix, L-BFGS stores only a few vectors that represent the approximation. This algorithm is better suited for large-scale models than the BFGS algorithm. When (`method="LBFGS"`) for `LaplaceApproximation`, `method="L-BFGS-B"` is called in the `optim` function of base R.

Nelder-Mead

The Nelder-Mead algorithm (Nelder and Mead 1965) is a derivative-free, direct search method that is known to become inefficient in large-dimensional problems. As the dimension increases, the search direction becomes increasingly orthogonal to the steepest ascent (usually descent) direction. However, in smaller dimensions it is a popular algorithm. At each iteration, three steps are taken to improve a simplex: reflection, extension, and contraction.

Newton-Raphson

The Newton-Raphson optimization algorithm, also known as Newton's Method, uses deriva-

tives and a Hessian matrix. The algorithm is included for its historical significance, but is known to be problematic when starting values are far from the targets, and calculating and inverting the Hessian matrix can be computationally expensive. As programmed here, when the Hessian is problematic, it tries to use only the derivatives, and when that fails, a jitter is applied. Newton-Raphson should not be the first choice of the user, and Levenberg-Marquardt should always be preferred.

Particle Swarm Optimization

Of numerous Particle Swarm Optimization (PSO) algorithms, the Standard Particle Swarm Optimization 2007 (SPSO 07) algorithm is used here. A swarm of particles is moved according to velocity, neighborhood, and the best previous solution. The neighborhood for each particle is a set of informing particles. PSO is derivative-free.

Resilient Backpropagation

"Rprop" stands for resilient backpropagation. In Rprop, the approximate gradient is taken for each parameter in each iteration, and its sign is compared to the approximate gradient in the previous iteration. A weight element in a weight vector is associated with each approximate gradient. A weight element is multiplied by 1.2 when the sign does not change, or by 0.5 if the sign changes. The weight vector is the step size, and is constrained to the interval [0.001, 50], and initial weights are 0.0125. This is the resilient backpropagation algorithm, which is often denoted as the "Rprop-" algorithm of Riedmiller (1994).

Self-Organizing Migration Algorithm

The Self-Organizing Migration Algorithm (SOMA) of Zelinka (2004), as used here, moves a population of ten particles or individuals in the direction of the best particle, the leader. The leader does not move in each iteration, and a line-search is used for each non-leader, up to three times the difference in parameter values between each non-leader and leader. This algorithm is derivative-free and often considered in the family of evolution algorithms. Numerous model evaluations are performed per non-leader per iteration.

Spectral Projected Gradient

The Spectral Projected Gradient (SPG) algorithm is a non-monotone algorithm that is suitable for high-dimensional models. The approximate gradient is used, but the Hessian matrix is not. SPG is the default algorithm for the `LaplaceApproximation` function.

Stochastic Gradient Descent

The stochastic gradient descent (SGD) algorithm, here, is designed only for big data. Traditional optimization algorithms require the entire data set to be included in the model evaluation each iteration. In contrast, SGD reads and processes only a small, randomly selected batch of records each iteration. In addition to saving computation time, the entire data set does not need to be loaded into memory at once.

In this version of SGD, a multivariate proposal is used, and it is merely the vector of current values plus a step size times the gradient.

SGD requires five objects in the `Data` list: `epsilon` or ϵ is the step size as a scalar, `file` is a quoted name of a .csv file that is the big data set, `Nr` is the number of rows in the big data set, `Nc` is the number of columns in the big data set, and `size` is the number of rows to be read and processed each iteration.

Since SGD, as implemented here, is designed for big data, the entire data set is not included in the `Data` list, but one small batch must be included and named `X`. All data must be included. For example, both the dependent variable `y` and design matrix **X** in linear regression are included. The requirement for the small batch to be in `Data` is so that numerous checks may be passed after `LaplaceApproximation` is called and before the SGD algorithm begins. Each iteration, SGD uses the `scan` function, without headers, to read a random block of rows from, say, `X.csv`, stores it in `Data$X`, and passes it to the `Model` specification function. The `Model` function must differ from the other examples found in this package in that multiple objects, such as `X` and `y` must be read from `Data$X`, where usually there is both `Data$X` and `Data$y`.

The user tunes SGD with step size ϵ via `Data$epsilon`. The step size must be scalar and remain in the interval (0,1). When $\epsilon = 0$, SGD is reduced to zero, the algorithm will not move, and false convergence occurs. When ϵ is too large, degenerate results occur. A good recommendation seems to be to begin with ϵ set to `1/Nr`. The user may perform several short runs, and experimenting with adjusting `Data$epsilon`. At least `Nr / size` iterations are suggested.

Symmetric Rank-One

The Symmetric Rank-One (SR1) algorithm is a quasi-Newton optimization algorithm, and the Hessian matrix is approximated, often without being positive-definite. At the posterior modes, the true Hessian is usually positive-definite, but this is often not the case during optimization when the parameters have not yet reached the posterior modes. Other restrictions, including constraints, often result in the true Hessian being indefinite at the solution. For these reasons, SR1 often outperforms BFGS. The approximate Hessian is not guaranteed to converge to the Hessian. When SR1 is used, the approximate Hessian is not used to calculate the final covariance matrix.

Trust Region

The Trust Region (TR) algorithm of Nocedal and Wright (1999) attempts to reach its objective in the fewest number of iterations, is therefore very efficient, as well as safe. The efficiency of TR is attractive when model evaluations are expensive.The Hessian is approximated each iteration, making TR best suited to models with small to medium dimensions, say up to a few hundred parameters.

Afterward

After `LaplaceApproximation` finishes, due either to early convergence or completing the number of specified iterations, it approximates the Hessian matrix of second derivatives (by default, but the user has other options), and attempts to calculate the covariance matrix by taking the inverse of the negative of this matrix. If successful, then this covariance matrix may be passed to `IterativeQuadrature`, `LaplacesDemon`, or `PMC`, and the diagonal of this matrix is the variance of the parameters. If unsuccessful, then a scaled identity matrix is returned, and each parameter's variance will be 1.

12.5. Markov Chain Monte Carlo

Markov chain Monte Carlo (MCMC) algorithms are also called samplers. There are a large number of MCMC algorithms, too many to review here. Popular families (which are often non-distinct) include Gibbs sampling, Metropolis-Hastings, slice sampling, Hamiltonian Monte Carlo, and many others. Though the name is misleading, Metropolis-within-Gibbs (MWG) was developed first (Metropolis, Rosenbluth, M.N., and Teller 1953), and Metropolis-Hastings was a generalization of MWG (Hastings 1970). All MCMC algorithms are known as special cases of the Metropolis-Hastings algorithm. Regardless of the algorithm, the goal in Bayesian inference is to maximize the unnormalized joint posterior distribution and collect samples of the target distributions, which are marginal posterior distributions, later to be used for inference.

The most generalizable MCMC algorithm is the Metropolis-Hastings (MH) generalization of the MWG algorithm. The MH algorithm extended MWG to include asymmetric proposal distributions. For years, the main disadvantage of the MWG algorithms was that the proposal variance (see below) had to be tuned manually, and therefore other MCMC algorithms have become popular because they do not need to be tuned.

Gibbs sampling became popular for Bayesian inference, though it requires conditional sampling of conjugate distributions, so it is precluded from non-conjugate sampling in its purest form. Gibbs sampling also suffers under high correlations (Gilks and Roberts 1996). Due to these limitations, Gibbs sampling is less generalizable than RWM, though RWM and other algorithms are not immune to problems with correlation. The Griddy-Gibbs sampler evaluates a grid of proposals and approximates the conditional distribution, which enables non-conjugate sampling. Componentwise slice sampling is a special case of Gibbs sampling that samples a distribution by sampling uniformly from the region under the plot of its density function, and is more appropriate with bounded distributions that cannot approach infinity, though the improved slice sampler of Neal (2003) is available here (see 12.5.36).

Blockwise Sampling

Usually, there is more than one target distribution, in which case it must be determined whether it is best to sample from target distributions individually, in groups, or all at once. Block updating refers to splitting a multivariate vector into groups called blocks, and each block is sampled separately. A block may contain one or more parameters.

Parameters are usually grouped into blocks such that parameters within a block are as correlated as possible, and parameters between blocks are as independent as possible. This strategy retains as much of the parameter correlation as possible for blockwise sampling, as opposed to componentwise sampling where parameter correlation is ignored. The `PosteriorChecks` function can be used on the output of previous runs to find highly correlated parameters, and the `Blocks` function may be used to create blocks based on posterior correlation.

Advantages of blockwise sampling are that a different MCMC algorithm may be used for each block (or parameter, for that matter), creating a more specialized approach (though different algorithms by block are not supported here), the acceptance of a newly proposed state is likely to be higher than sampling from all target distributions at once in high dimensions, and large proposal covariance matrices can be reduced in size, which is most helpful again in high dimensions.

Disadvantages of blockwise sampling are that correlations probably exist between parameters

between blocks, and each block is updated while holding the other blocks constant, ignoring these correlations of parameters between blocks. Without simultaneously taking everything into account, the algorithm may converge slowly or never arrive at the proper solution. However, there are instances when it may be best when everything is not taken into account at once, such as in state-space models. Also, as the number of blocks increases, more computation is required, which slows the algorithm. In general, blockwise sampling allows a more specialized approach at the expense of accuracy, generalization, and speed. Blockwise sampling is offered in the following algorithms: Adaptive-Mixture Metropolis (AMM), Elliptical Slice Sampler (ESS), Hit-And-Run Metropolis (HARM), and Random-Walk Metropolis (RWM).

Markov Chain Properties

This tutorial introduces only briefly the basics of Markov chain properties. A Markov chain is Markovian when the current iteration depends only on the previous iteration. Many (but not all) adaptive algorithms are merely chains but not Markov chains when the adaptation is based on the history of the chains, not just the previous iteration. A Markov chain is said to be aperiodic when it is not repeating a cycle. A Markov chain is considered irreducible when it is possible to go from any state to any other state, though not necessarily in one iteration. A Markov chain is said to be recurrent if it will eventually return to a given state with probability 1, and it is positive recurrent if the expected return time is finite, and null recurrent otherwise. The ergodic theorem states that a Markov chain is ergodic when it is aperiodic, irreducible, and positive recurrent.

The non-Markovian chains of an adaptive algorithm that adapt based on the history of the chains should have two conditions: containment and diminishing adaptation. Containment is difficult to implement and is not currently programmed into **LaplacesDemon**. The condition of diminishing adaptation is fulfilled when the amount of adaptation diminishes with the length of the chain. Diminishing adaptation can be achieved when the proposal variances become smaller or by decreasing the probability of performing adaptations with more iterations (Roberts and Rosenthal 2007). Trace-plots of the output of the `LaplacesDemon` function automatically include plots of the absolute differences in proposal variance with each adaptation for adaptive algorithms, and the `Consort` function will try to suggest a different adaptive algorithm when these absolute differences are not trending downward.

The MCMC algorithms in the **LaplacesDemon** package are now presented alphabetically.

Adaptive Directional Metropolis-within-Gibbs

The Adaptive Directional Metropolis-within-Gibbs (ADMG) algorithm was proposed by Bai (2009) as an extension of the Adaptive Metropolis-within-Gibbs (AMWG) algorithm in which the direction and jumping distance of the componentwise proposal is affected by an estimate of the historical covariance matrix. For more information on AMWG, see section 12.5.7.

ADMG has one algorithm specification: `Periodicity`. The `Periodicity` argument indicates the frequency in iterations at which the algorithm adapts.

Although ADMG is a componentwise algorithm, it does not ignore correlation of parameters, as is typical with componentwise algorithms. ADMG makes each componentwise proposal based on an estimate of the historical covariance matrix. Adaptation is based on the singular value decomposition (SVD) of the estimate of the historical covariance matrix. In large

dimensions, SVD is computationally expensive, and larger integers are recommended for the algorithm specifications. If SVD results in a singularity, then the algorithm defaults to a simple MWG proposal. Otherwise, when SVD is successful, then Metropolis-within-Gibbs (MWG) is performed under the rotated coordinates.

The author notes that the Metropolis-within-Gibbs part of the algorithm may proceed by deterministic-scan (which he refers to as system-scan) and random-scan. The algorithm implemented here uses random-scan, and is denoted by the author as ADRSMG. Random-scan updates means that the order of the update of the parameters is randomized each iteration.

Compared to other adaptive algorithms with multivariate proposals, a disadvantage is the time to complete each iteration increases as a function of parameters and model complexity, as noted in MWG. For example, in a 100-parameter model, ADMG completes its first iteration as the AMM algorithm completes its 100th. However, ADMG is more efficient with information per iteration.

The advantage of ADMG to other componentwise algorithms (such as AMWG, MWG, RDMH, and Slice) is that ADMG makes each proposal while accounting for parameter correlation, while other componentwise algorithms traditionally ignore parameter correlation. The author asserts that SVD-based adaptation is more efficient than adapting directly to the historical covariance matrix or an estimate thereof. The disadvantage of ADMG to other componentwise algorithms is that SVD becomes computationally expensive in large dimensions. Since ADMG is adaptive, MWG should be used as the final algorithm.

Adaptive Griddy-Gibbs

The Adaptive Griddy-Gibbs (AGG) sampler is an extension of the Griddy-Gibbs (GG) sampler (see 12.5.16) in which the scale of the centered grid is tuned each iteration for continuous parameters. The scale is calculated as the standard deviation of the conditional distribution. If discrete parameters are used, then the grid of discrete parameters is not tuned, since all discrete values are evaluated.

AGG has six algorithm specifications: `Grid` is a vector or list of points in the grid, `dparm` is a vector that indicates discrete parameters and defaults to `NULL`, `smax` is the maximum allowable conditional standard deviation, `CPUs` is the number of available central processing units (CPUs), `Packages` is a vector of package names, and `Dyn.lib` is a vector of shared libraries. The default for `Grid` is `GaussHermiteQuadRule(3)$nodes`. For each continuous parameter in each iteration, the scaled grid values are added to the latest value of the parameter, and the model is evaluated with the parameter at each point on the grid and a sample is taken. For each discrete parameter, the model is evaluated at each grid point and a sample is taken.

The points in the grid for continuous parameters are selected by the user as the nodes of a Gauss-Hermite quadrature rule (with the `GaussHermiteQuadRule` function), and these points are not evenly-spaced. Many problems require as few as 3 nodes, while others require perhaps as many as 99 nodes. When the number of nodes is larger than necessary, time is wasted in computation, fewer points occur within most of the density, and occasional extreme values of LP are observed, but the chains reach the target distributions in fewer iterations due to the wider grid. When the number of nodes is too small, it may not converge on the correct distribution. It is therefore suggested to begin with a small, odd number of grid points, such as 3, to reduce computation time. An odd number of grid points is preferred. If occasional extreme values of LP are observed, set `smax` to something reasonable. If `smax` is too small,

then higher autocorrelation will occur and more thinning is necessary.

Advantages of parallelized AGG over most componentwise algorithms is that it yields a 100% acceptance rate, and draws from the approximated distribution should be less autocorrelated. A disadvantage is that more model evaluations are required, and even if a parallel environment had zero overhead, AGG would be twice as slow per iteration as other componentwise algorithms. AGG is adaptive in the sense of self-adjusting, but not in the sense of being non-Markovian, so it is suitable as a final algorithm.

Adaptive Hamiltonian Monte Carlo

This is an adaptive form of Hamiltonian Monte Carlo (HMC) called Adaptive Hamiltonian Monte Carlo (AHMC). For more information on HMC, see section 12.5.17. In AHMC, an additional algorithm specification is included called `Periodicity`, which specifies how often the algorithm adapts, and it can only begin to adapt after the tenth iteration. Of the remaining algorithm specifications, the vector `epsilon` (ϵ) is adapted, and `L` (L) is not. When adapting, and considering K parameters, AHMC multiplies ϵ_k by 0.8 when a proposal for parameter k has not been accepted in the last 10 iterations, or multiplies it by 1.2 when a proposal has been accepted at least 8 of the last 10 iterations, as suggested by Neal (2011).

AHMC has three algorithm specifications: `epsilon` or ϵ is the step-size, `L` or L is the number of leapfrog steps, and `Periodicity` is the frequency in iterations for adaptation.

As with HMC, the `Demonic Suggestion` section of the output of `Consort` treats AHMC differently when $L > 1$ than most other algorithms by potentially suggesting a new value for L to achieve independent samples, without altering the latest specification of the user for `Iterations` and `Thinning`. The suggested value of L may be close to correct or wildly incorrect, so bear in mind that it is not an adaptive parameter here.

As with HMC, the AHMC algorithm is slower than many other algorithms, but often produces chains with good mixing. Alternatives to AHMC that should perform better are HMCDA and MALA, presented below. AHMC is more consistent with respect to time per iteration, because L remains constant, than HMCDA and NUTS, which may have some iterations that are much slower than others. If AHMC is used for adaptation, then the final, non-adaptive algorithm should be HMC.

Adaptive Metropolis

The Adaptive Metropolis (AM) algorithm of Haario, Saksman, and Tamminen (2001) is an extension of Random-Walk Metropolis (RWM) that adapts based on the observed covariance matrix from the history of the chains[12]. AM is historically significant as the first adaptive MCMC algorithm.

AM has two algorithm specifications: `Adaptive` is the iteration at which adaptation begins, and `Periodicity` is the frequency in iterations of adaptation. The user should not allow AM to adapt immediately, since AM adapts based on the observed covariance matrix of historical and accepted samples. Since enough samples are needed to obtain a valid covariance matrix before adaptation, a small covariance matrix is often used initially to encourage a high

[12] Haario *et al.* (2001) assert that the chains remain ergodic in the limit as the amount of change in the adaptations should decrease to zero as the chains approach the target distributions, now referred to as the diminishing adaptation condition of Roberts and Rosenthal (2007).

acceptance rate.

As recommended by Haario *et al.* (2001), there are two tricks that may be used to assist the AM algorithm in the beginning. Although the suggested "greedy start" method is not used here, the second suggested trick is used, which is to shrink the proposal as long as the acceptance rate is less than 5%, and there have been at least five acceptances. Haario *et al.* (2001) suggest loosely that if "it has not moved enough during some number of iterations, the proposal could be shrunk by a constant factor". For each iteration that the acceptance rate is less than 5% and that the AM algorithm is used but the current iteration is prior to adaptation, `LaplacesDemon` multiplies the proposal covariance or variance by (1 - 1/`Iterations`). Over pre-adaptive time, this encourages a smaller proposal covariance or variance to increase the acceptance rate so that when adaptation begins, the observed covariance or variance of the chains will not be constant, and then shrinkage will cease and adaptation will take it from there.

AM is best suited for a model with small or medium dimensions (number of parameters). The Adaptive-Mixture Metropolis (AMM) of Roberts and Rosenthal (2009) and Robust Adaptive Metropolis (Vihola 2011) are extensions of the AM algorithm. If AM is used for adaptation, then the final, non-adaptive algorithm should be RWM.

Adaptive Metropolis-within-Gibbs

The Adaptive Metropolis-within-Gibbs (AMWG) algorithm is presented in (Roberts and Rosenthal 2009; Rosenthal 2007). It is an adaptive version of Metropolis-within-Gibbs (MWG). For more information on MWG, see section 12.5.24.

AMWG has one algorithm specification: `Periodicity`. The `Periodicity` argument indicates the frequency in iterations at which the algorithm adapts.

In AMWG, the standard deviation of the proposal of each parameter is manipulated to optimize the associated acceptance rate toward 0.44. This is much simpler than other adaptive methods that adapt based on sample covariance in large dimensions. Large covariance matrices require a large number of elements to adapt, which takes exponentially longer to adapt as the dimension increases. Regardless of dimension, the AMWG optimizes each parameter to a univariate acceptance rate, and a sample covariance matrix does not need to be estimated for adaptation, which consumes time and memory. The order of the parameters for updating is randomized each iteration (random-scan AMWG), as opposed to sequential updating (deterministic-scan AMWG).

Compared to other adaptive algorithms with multivariate proposals, a disadvantage is the time to complete each iteration increases as a function of parameters and model complexity, as noted in MWG. For example, in a 100-parameter model, AMWG completes its first iteration as the AMM algorithm completes its 100th. However, to adapt accurately, the AMM algorithm must correctly estimate 5,050 elements of a sample covariance matrix, while AMWG must correctly estimate only 100 proposal standard deviations. Roberts and Rosenthal (2009) have shown an example model with 500 parameters that had a burn-in of around 25,000 iterations.

The advantages of AMWG over AMM are that AMWG does not require a burn-in period before it can begin to adapt, and that AMWG does not need to estimate a covariance matrix to adapt properly. The disadvantages of AMWG compared to AMM are that correlation can be problematic since it is not taken into account with a proposal covariance matrix, and AMWG solves the model function once per parameter per iteration, which can be unacceptably slow

with large or complicated models. The advantage of AMWG over RAM is that AMWG does not need to estimate a covariance matrix to adapt properly. The disadvantages of AMWG compared to RAM are AMWG is less likely to handle multimodal or heavy-tailed targets, and AMWG solves the model function once per parameter per iteration, which can be unacceptably slow with large or complicated models. If AMWG is used for adaptation, then the final, non-adaptive algorithm should be MWG.

Adaptive-Mixture Metropolis

The Adaptive-Mixture Metropolis (AMM) algorithm is an extension by Roberts and Rosenthal (2009) of the Adaptive-Metropolis (AM) algorithm of Haario *et al.* (2001). AMM differs from the AM algorithm in two respects. First, AMM updates a scatter matrix based on the cumulative current parameters and the cumulative associated outer-products, and these are used to generate a multivariate normal proposal. This is more efficient with large numbers of parameters adapting over many iterations, especially with frequent adaptations, and results in a much faster algorithm. The second (and main) difference, is that the proposal is a mixture. The two mixture components are adaptive multivariate and static/symmetric univariate proposals. The mixture is determined at each iteration with a mixture weight. The mixture weight must be in the interval (0,1], and it defaults to 0.05, as in Roberts and Rosenthal (2009). A higher value of the mixture weight is associated with more static/symmetric univariate proposals, and a lower weight is associated with more adaptive multivariate proposals. The algorithm will be unable to include the multivariate mixture component until it has accumulated some history, and models with more parameters will take longer to be able to use adaptive multivariate proposals.

AMM has four algorithm specifications: `Adaptive` is the iteration in which adaptation begins, `B` accepts a list of blocked parameters, `Periodicity` is the frequency in iterations of adaptation, and `w` is the weight of the small-variance mixture component. `B` allows the user to organize parameters into blocks. `B` accepts a list, in which each component is a block and accepts a vector that consists of numbers that point to the associated parameters in parm.names. `B` defaults to `NULL`, in which case blocking does not occur. When blocking does occur, the proposal covariance matrix may be either `NULL` or a list in which each component is the covariance matrix for a block. As more blocks are added, the algorithm becomes closer to Adaptive Metropolis-within-Gibbs (AMWG).

The advantages of AMM over AMWG are that it takes correlation into account (when `B=NULL`) as it adapts, and is much faster to update each iteration. The disadvantages are that AMWG does not require a burn-in period before it can begin to adapt, and more information must be learned in the covariance matrix to adapt properly (see section 12.5.7 for more). Disadvantages of AMM compared to Robust Adaptive Metropolis (RAM) are that RAM does not require a burn-in period before it can begin to adapt, RAM is more likely to better handle multimodal or heavy-tailed targets, and RAM also adapts to the shape of the target distributions and coerces the acceptance rate. If AMM is used for adaptation, then the final, non-adaptive algorithm should be Random-Walk Metropolis (RWM).

Affine-Invariant Ensemble Sampler

The Affine-Invariant Ensemble Sampler (AIES) of Goodman and Weare (2010) uses a complmentary ensemble of at least $2J$ walkers for J parameters. Each walker receives J initial

values, and initial values must differ for each walker. At each iteration, AIES makes a multivariate proposal for each walker, given a scaled difference in position by parameter between the current walker and another randomly-selected walker.

AIES has six algorithm specifications: `Nc` is the number of walkers, `Z` is a $N_c \times J$ matrix of initial values, β is a scale parameter, `CPUs` is the number of central processing units (CPUs), `Packages` as a vector of package names, and `Dyn.lib` is a vector of shared libraries. The recommended number of walkers is at least $2J$. If separate sets of initial values are not supplied in `Z`, since `Z` defaults to `NULL`, then the `GIV` function is used to generate initial values. The original article referred to the scale parameter as α, though it has been renamed here to β to avoid a conflict with the acceptance probability α in the Metropolis step. The β parameter may be manipulated to affect the desired acceptance rate, though in practice, the acceptance rate may potentially be better affected by increasing the number of walkers. It is recommended to specify `CPUs=1` and leave the remaining arguments to `NULL`, unless needed.

This version returns the samples from one walker, and the other walkers assist the main walker. A disadvantage of this approach is that all samples from all walkers are not returned. An advantage of this approach is that if a particular walker is an outlier, then it does not affect the main walker, unless of course it is the main walker! Multiple sets of samples are best returned in a list, such as with parallel chains in the `LaplacesDemon.hpc` function, though it is not applicable in `LaplacesDemon`.

AIES has been parallelized by Foreman-Mackey, Hogg, Lang, and Goodman (2012), and this style of parallelization is available here as well. The user is cautioned to prefer `CPUs=1`, because parallelizing may result in a slower algorithm due to communication between the master and slave CPUs each iteration. This communication is costly, and is best overcome with a large number of CPUs available, and when the `Model` function is slow to evaluate in comparison to network communication time.

Both the parallelized (CPUs > 1) and un-parallelized (CPUs=1) versions should be called from `LaplacesDemon`, not `LaplacesDemon.hpc`. When parallelized, the number of walkers must be an even number (odd numbers are not permitted), and the walkers are split into two equal-sized batches. Each iteration, each walker moves in relation to a randomly-selected walker in the other batch. This retains detailed balance.

AIES is attractive for offering affine-invariance, and therefore being generally robust to badly scaled posteriors, such as with highly correlated parameters. It is also attractive for making a multivariate proposal without a proposal covariance matrix. However, since at least $2J$ walkers are recommended, the number of model evaluations per iteration exceeds most componentwise algorithms by at least twice, making AIES a slow algorithm per iteration, and computation scales poorly with model dimension. Large-scale computing environments may overcome this limitation with parallelization, but parallelization is probably not very helpful (and may be detrimental) in small-scale computing environments when evaluating the model function is not slow in comparison with network communication time. AIES is not an adaptive algorithm, and is therefore suitable as a final algorithm.

Componentwise Hit-And-Run Metropolis

The Hit-And-Run algorithm is a variation of RWM that has been around as long as Gibbs sampling. Hit-And-Run randomly samples a direction on the unit sphere, and a proposal is made for each parameter in its randomly-selected direction for a uniformly-distributed

distance Gilks and Roberts (1996). This version of Hit-And-Run, called Componentwise Hit-And-Run Metropolis (CHARM), includes componentwise proposals and a Metropolis step for rejection. Introduced by Turchin (1971) along with Gibbs sampling, and popularized by Smith (1984), Hit-And-Run was given its name later due to its ability to run across the state-space and arrive at a distant "hit-point". It is related to other algorithms with interesting names, such as Hide-And-Seek and Shake-And-Bake.

CHARM has one optional algorithm specification: `alpha.star`. When `Specs=NULL`, CHARM is non-adaptive. Otherwise, `alpha.star` is the target acceptance rate, and is recommended to be 0.44. When a target acceptance rate is declared, CHARM applies the Robbins-Monro stochastic approximation of Garthwaite *et al.* (2010) to the proposal distance to attain the target acceptance rate.

As a componentwise algorithm, the model is evaluated after a proposal is made for each parameter, which results in an algorithm that takes more time per iteration. As with the Metropolis-within-Gibbs (MWG) family, the time to complete each iteration grows with the number of parameters. Compared to other algorithms with multivariate proposals, a disadvantage is the time to complete each iteration increases as a function of parameters and model complexity. For example, in a 100-parameter model, CHARM completes its first iteration as HARM completes its 100th.

CHARM enjoys many of the advantages of HARM, such as having no tuning parameters (unless the adaptive form is used), traversing complex spaces with bounded sets in one iteration, not being adaptive (unless specified as adaptive), handling high correlations well, and having the potential to work well with multimodal distributions. When non-adaptive, CHARM may be used as a final algorithm.

Delayed Rejection Metropolis

The Delayed Rejection Metropolis (DRM or DR) algorithm is a RWM with one, small twist. Whenever a proposal is rejected, the DRM algorithm will try one or more alternate proposals, and correct for the probability of this conditional acceptance. By delaying rejection, autocorrelation in the chains may be decreased, and the algorithm is encouraged to move. Currently, `LaplacesDemon` will attempt one alternate proposal when using the DRAM (see section 12.5.12) or DRM algorithm. The additional calculations may slow each iteration of the algorithm in which the first set of proposals is rejected, but it may also converge faster. For more information on DRM, see Mira (2001).

DRM does not have any algorithm specifications.

DRM may be considered to be an adaptive MCMC algorithm, because it adapts the proposal based on a rejection. However, DRM does not violate the Markov property, because the proposal is based on the current state. For the purposes of `LaplacesDemon`, DRM is not considered to be an adaptive MCMC algorithm, because it is not adapting to the target distribution by considering previous states in the Markov chain, but merely makes more attempts from the current state.

`LaplacesDemon` also temporarily shrinks the proposal covariance arbitrarily by 50% for delayed rejection. A smaller proposal covariance is more likely to be accepted, and the goal of delayed rejection is to increase acceptance. In the long-term, a proposal covariance that is too small is undesirable, and so it is only used in this case to assist acceptance.

Since DRM is non-adaptive, it is suitable as a final algorithm.

Delayed Rejection Adaptive Metropolis

The Delayed Rejection Adaptive Metropolis (DRAM) algorithm is merely the combination of both Delayed Rejection Metropolis (DRM) and Adaptive Metropolis (AM) (Haario, Laine, Mira, and Saksman 2006). DRAM has been demonstrated to be robust in extreme situations where DRM or AM fail separately. Haario *et al.* (2006) present an example involving ordinary differential equations in which least squares could not find a stable solution, and DRAM did well.

DRAM has two algorithm specifications: `Adaptive` is the iteration in which DRAM becomes adaptive, and `Periodicity` is the frequency in iterations of adaptation.

The DRAM algorithm is useful to assist AM when the acceptance rate is low. As an alternative, the Adaptive-Mixture Metropolis (AMM) is an extension of the AM algorithm that includes a mixture of proposals, and one mixture component has a small proposal standard deviation to assist in overcoming initially low acceptance rates. If DRAM is used for adaptation, then the final algorithm should be Random-Walk Metropolis (RWM).

Differential Evolution Markov Chain

The original Differential Evolution Markov Chain (DEMC) (Ter Braak 2006), referred to in the literature as DE-MC, included a Metropolis step on a genetic algorithm called Differential Evolution with multiple chains (per parameter), in which the chains learn from each other. It could be considered as parallel adaptive direction sampling (ADS) with the Gibbs sampling (Gibbs) step replaced by a Metropolis step, or as a non-parametric form of Random-Walk Metropolis (RWM). However, for a model with J parameters, the original DEMC required at least $2J$ chains, and often required as much as $20J$ chains. Hence, from $2J$ to $20J$ model evaluations were required per iteration, whereas a typical multivariate sampler such as Adaptive-Mixture Metropolis (AMM) requires one evaluation, or a componentwise sampler such as Adaptive Metropolis-within-Gibbs (AMWG) requires J evaluations per iteration.

The version included here was presented in Ter Braak and Vrugt (2008), and the required number of chains is drastically reduced by adapting based on historical, thinned samples (the parallel direction move), and periodically using a snooker move instead. In the article, three chains were used to update as many as 50-100 parameters. Larger models may require blocks (as suggested in the article, and blocking is not included here), or more chains (see below). In testing, here, a few 200-dimensional models have been solved with 5-10 chains.

DEMC has four algorithm specifications: `Nc` is the number of chains (at least 3), `Z` is a $T \times J$ matrix or $T \times J \times N_c$ array of T thinned samples for J parameters and N_c chains, `gamma` is a scale parameter, and `w` is the probability of a snooker move for each iteration. When `gamma=NULL`, the scale parameter defaults to the recommended $2.381204/\sqrt{2J}$, though for snooker moves, it is $2.381204/\sqrt{2}$ regardless of the algorithm specification. The default, recommended probability of a snooker move is $w = 0.1$.

The parallel direction move consists of a multivariate proposal for each chain in which two randomly selected previous iterations from two randomly selected other chains are differenced and scaled, and a small jitter, $\mathcal{U} \sim (-0.001, 0.001)^J$, is added. The snooker move differences these with another randomly selected (current) chain in the current iteration, and with a fixed scale. Another variation is to use snooker with past chain time-periods. The snooker move facilitates mode-jumping behavior for multimodal posteriors.

For the first update, `Z` is usually `NULL`. Internally, `LaplacesDemon` populates `Z` with `GIV`, and it is strongly recommended that `PGF` is specified by the user. As the sampler iterates, `Z` is used for adaptation, and elements of `Z` are replaced with thinned samples. A short, first run is recommended to obtain thinned samples, such as in `Fit$Posterior1`. For consecutive updates, this posterior matrix is used as `Z`.

In this implementation, samples are returned of the main chain, for which `Initial.Values` are specified. Samples for other chains (associated with `PCIV`) are not returned, but are used to assist the main chain. The authors note that too many chains can be problematic when an outlier chain exists. Here, samples of other chains are not returned, outlier or not. If an outlier chain exists, it simply does not help the main chain much and wastes computational resources, but does not negatively affect the main chain.

An attractive property is that DEMC does not require a proposal covariance matrix, but adapts instead based on past (thinned) samples. In the output of `LaplacesDemon`, the thinned samples are also stored in `CovarDHis`, though they are thinned samples, not the history of the diagonal of the covariance matrix. This is done so the absolute differences can be observed for diminishing adaptation. Another attractive property is that the chains may be parallelized, such as across CPUs, in the future, though the current version is not parallelized.

DEMC has been considered to perform like a version of Metropolis-within-Gibbs (MWG) that updates by chain, rather than by component. DEMC may be one form of a compromise between a one-evaluation multivariate proposal and J componentwise proposals. Since it is adaptive, DEMC is not recommended as a final algorithm.

Elliptical Slice Sampler

The Elliptical Slice Sampler (ESS) was introduced by Murray, Adams, and KacKay (2010) as a multivariate extension of componentwise slice sampling (see 12.5.36) that utilizes ellipse ν and angle θ. Each iteration, ellipse ν is drawn from $\mathcal{N}(0, \Sigma)$, where Σ is a user-specified prior covariance for the ellipse. The authors recommend using a form of the covariance of the data. Even though other parameters are not discussed, an identity matrix, or a combination thereof, performs well in practice. The prior covariance matrix is accepted as `VarCov` in the `LaplacesDemon` function, rather than the usual proposal covariance matrix.

ESS has one algorithm specification `B` for blocks of parameters, and defaults to `B=NULL`. For large-dimensional problems, block updating may be used. To specify block updating, the user gives the `B` specification a list in which each component is a block, and each component is a vector of integers pointing to the position of the parameters for that block. Block updating also requires the `Covar` argument to receive a list of prior covariance matrices of the appropriate dimension.

Each proposal is an additive, trigonometric function of the current state, ellipse ν, and angle θ. Angle θ is originally in the interval $(0, 2\pi]$ each iteration, and is shrunk until acceptance occurs. This results in a 100% acceptance rate, and usually multiple model evaluations per iteration.

This algorithm is applicable only to models in which the prior mean of all parameters is zero. It is often possible to apply ESS when the variables are centered and scaled.

An advantage of ESS over the (componentwise) Slice algorithm is the computational efficiency of a multivariate proposal. A disadvantage is that the user must specify the prior covariance Σ for optimal ESS performance, and the algorithm can be very sensitive to this prior. Since

ESS is not adaptive, it is suitable as a final algorithm.

Gibbs Sampler

Gibbs sampling was introduced by Turchin (1971), and later by brothers Geman and Geman (1984). The Geman brothers named the algorithm after the physicist J. W. Gibbs, some eight decades after his death, in reference to an analogy between the sampling algorithm and statistical physics. Geman and Geman introduced Gibbs sampling in the context of image restoration.

Gibbs has two algorithm specifications: `FC` accepts a user-specified function to calculate full-conditionals and `MWG` defaults to `NULL` and otherwise accepts a numeric vector that indicates which parameters are updated with Metropolis-within-Gibbs. `FC` accepts two arguments, `parm` and `Data`, just like the Model specification function, and returns the full vector of parameters `parm`. Each iteration, the full-conditional distributions are completed first, then MWG updates, if any.

In its basic version, Gibbs sampling is a special case of the Metropolis-Hastings algorithm. However, in its extended versions, Gibbs sampling can be considered a general framework for sampling from a large set of variables by sampling each variable (or in some cases, each group of variables) in turn, and can incorporate the Metropolis-Hastings algorithm (such as Metropolis-within-Gibbs or similar methods such as slice sampling) to implement one or more of the sampling steps. Currently, Metropolis-within-Gibbs is included.

Gibbs sampling is applicable when the joint distribution is not known explicitly or is difficult to sample from directly, but the conditional distribution of each variable is known and easy to sample from. A Gibbs sampler generates a draw from the distribution of each parameter or variable in turn, conditional on the current values of the other parameters or variables. Therefore, a Gibbs sampler is a componentwise algorithm. As a simplified example, given a joint probability distribution $p(\theta_1, \theta_2|\mathbf{y})$, a Gibbs sampler would draw $p(\theta_1|\theta_2, \mathbf{y})$, then $p(\theta_2|\theta_1, \mathbf{y})$.

For a user to determine each conditional distribution, the joint distribution must be known first, then all parameters are held constant except the current parameter of interest, and the equation is simplified to the conditional distribution.

There are numerous variations of Gibbs sampling, such as blocked Gibbs sampling, collapsed Gibbs sampling, and ordered overrelaxation.

The advantage of Gibbs sampling to other MCMC algorithms is that it is often more efficient when it is appropriate, due to a 100% acceptance rate. The disadvantages are that a Gibbs sampler is appropriate only with conjugate distributions and low correlations between parameters, and therefore Gibbs sampling is less generally applicable than other MCMC algorithms. Since Gibbs is not adaptive, it is suitable as a final algorithm.

Griddy-Gibbs

Introduced by Ritter and Tanner (1992), the Griddy-Gibbs (GG) sampler is a componentwise algorithm that approximates the conditional distribution by evaluating the model at a discrete set of points, the user-specified grid for each parameter. The proposal is a random draw from the approximated distribution. In this implementation, splinetic interpolation is used to approximate the distribution for continuous parameters with 1,000 points, given the evaluated

points. The acceptance rate is 100%.

GG has five algorithm specifications: `Grid` is a vector or list of evenly-spaced points in the grid, `dparm` is a vector that indicates discrete parameters and defaults to `NULL`, `CPUs` is the number of available central processing units (CPUs), `Packages` is a vector of package names, and `Dyn.lib` is a vector of shared libraries. The default for `Grid` is `seq(from=-0.1, to=0.1, len=5)`, which creates a grid with the values -0.1, -0.05, 0, 0.05, and 0.1. For each continuous parameter in each iteration, the grid values are added to the latest value of the parameter, and the model is evaluated with the parameter at each point on the grid. For each discrete parameter, the model is evaluated at each grid point and a sample is taken.

At least five grid points are recommended for a continuous parameter, and a grid with more points will better approximate the distribution, but requires more evaluations. However, the approximation in GG may be parallelized (within `LaplacesDemon`, not `LaplacesDemon.hpc`), so a large computer environment may approach an excellent approximation with little inefficiency. It is natural to desire a grid with a larger range, but in practice this becomes problematic, so it is recommended to keep the range of the grid relatively small, say within [-0.1,0.1] or [-0.2,0.2], and may require experimentation. After observing a Markov chain, the user may adjust the range of the grid to decrease autocorrelation in a future update.

When an odd number of grid points is used for a continuous parameter, the current position is evaluated. If there are too few grid points, then the current point may be the only point with appreciable density, and the parameter may never move. This can be checked afterward with the `AcceptanceRate` function. If the acceptance rate is less than one for any parameter, then there are too few grid points. This failure may be harder to find when there are numerous parameters, an even number of grid points, and too few grid points, because the acceptance rate may be 100% for a parameter, yet it may be oscillating between two values.

GG is one of the slower algorithms per iteration, since the model must be evaluated multiple times per parameter per iteration. This may mostly be alleviated in a parallel environment. GG seems appropriate when the problem must be solved with a componentwise algorithm, and excessive thinning is required. GG may help reduce thinning by making proposals from the approximated conditional distribution, though parameter correlation may increase autocorrelation.

Advantages of parallelized GG over most componentwise algorithms is that it yields a 100% acceptance rate, and draws from the approximated distribution should be less autocorrelated. A disadvantage is that more model evaluations are required, and even if a parallel environment had zero overhead, GG would be twice as slow per iteration as other componentwise algorithms. However, if the user tunes the grid, it may be more efficient than other componentwise algorithms. The Adaptive Griddy-Gibbs (AGG) sampler is available so the user can avoid tuning the grid. Since GG is not adaptive, it is suitable as a final algorithm.

Hamiltonian Monte Carlo

Introduced under the name of hybrid Monte Carlo (Duane, Kennedy, Pendleton, and Roweth 1987), the name Hamiltonian Monte Carlo (HMC) surpasses it in popularity in statistics literature. HMC introduces auxiliary momentum variables with independent, Gaussian proposals. Momentum variables receive alternate updates, from simple updates to Metropolis updates. Metropolis updates result in the proposal of a new state by computing a trajectory according to Hamiltonian dynamics, from physics. Hamiltonian dynamics is discretized with

the leapfrog method. In this way, distant jumps can be proposed and random-walk behavior avoided.

HMC has two algorithm specifications: a vector of the step size of the leapfrog steps, `epsilon` (ϵ), that is equal in length to the number of parameters, and the number of leapfrog steps, `L` (L). When $L = 1$, HMC reduces to Langevin Monte Carlo (LMC), also called the Metropolis-Adjusted Langevin Algorithm (MALA), introduced by Rossky, Doll, and Friedman (1978). An adaptive version is available in section 12.5.22. For HMC, these tuning parameters must be adjusted until the acceptance rate is appropriate. The optimal acceptance rate of HMC is 65%, and Laplace's Demon is appeased when it is within the interval [60%, 70%], or in the case of LMC or MALA, in the interval [40%, 80%], where 57.4% is optimal. Tuning ϵ and L, however, is very difficult. The trajectory length, ϵL must also be considered. The ϵ vector is output in the list component `CovarDHis`, though it is not the diagonal of a covariance matrix.

Suggestions for tuning ϵ and L are found in Neal (2011). When ϵ is too large, the algorithm becomes unstable and suffers from a low acceptance rate. When ϵ is too small, the algorithm takes too many small steps and is inefficient. When L is too large, trajectory lengths (ϵL) result in double-back behavior and become computationally self-defeating. When L is too small, more random-walk behavior occurs and mixing becomes slower.

If a user is new to tuning HMC algorithms, then good advice may be to leave $L = 1$ and begin with small values for ϵ, say 0.1 or smaller. To avoid tuning in this case, the adaptive MALA algorithm may be used instead. It is easy to experience problems when inexperienced, but HMC is a rewarding algorithm once proficiency is acquired. As can be expected, the adaptive extensions (AHMC, HMCDA, MALA, and NUTS), will also be easier, since ϵ is adapted and does not require tuning (and in the case of NUTS, L does not require tuning).

Partial derivatives are required, and hence the parameters must be differentiable[13] everywhere the algorithm explores. Partial derivatives are approximated with the `partial` function. This is computationally intensive, and computational expense increases with the number of parameters. For K parameters and L leapfrog steps, there are $L + KL$ evaluations of the model specification function per iteration. In practice, starting any of the algorithms in the HMC family (AHMC, HMC, HMCDA, MALA, or THMC) in a region that is distant from density may result in failure due to differentiation, unless manipulated with priors.

The `Demonic Suggestion` section of the output of `Consort` treats HMC (when $L > 1$) differently than most other algorithms. For example, after updating a model with HMC, Laplace's Demon will not suggest a different number of iterations and thinning, but instead may suggest a new value of L after taking autocorrelation in the chains into account. As L increases, the speed per iteration decreases due to more calculations, and a higher value of L is not necessarily desirable. Laplace's Demon attempts suggestions in an effort to give independent samples consistent with the latest specification of the user for iterations.

Since HMC requires the approximation of partial derivatives, it is slower per iteration than most algorithms, and much slower in higher dimensions. Tuned well, HMC is an excellent algorithm, but tuning can be very difficult. The AHMC algorithm (described above) and HMCDA (below) are adaptive versions of HMC in which ϵ is adapted based on recent history of acceptance and rejection. The MALA and NUTS algorithms (below) are fully adaptive

[13] When the joint posterior is not differentiable, and should be, it has probably encountered an area of flat density. It is recommended that WIPs are used for regularization. For more information on WIPs, see the accompanying vignette entitled "Bayesian Inference".

versions that does not require tuning of ϵ or L. Since HMC is not adaptive, it is suitable as a final algorithm.

Hamiltonian Monte Carlo with Dual-Averaging

The Hamiltonian Monte Carlo with Dual-Averaging (HMCDA) algorithm is an extension of Hamiltonian Monte Carlo (HMC) that adapts the scalar step-size parameter, ϵ, according to dual-averaging. This is algorithm #5 in Hoffman and Gelman (2012), with the addition of an optional constraint, `Lmax`. For more information on HMC, see section 12.5.17.

HMCDA has five algorithm specifications: the number of adaptive iterations `A`, the target acceptance rate `delta` or δ (and 0.65 is recommended), a scalar step-size `epsilon` or ϵ, a maximum number of leapfrog steps `Lmax`, and the trajectory length `lambda` or λ. When `epsilon=NULL`, a reasonable initial value is found. The trajectory length is scalar $\lambda = \epsilon L$, where L is the unspecified number of leapfrog steps that is determined from ϵ and λ. The `Consort` function requires the acceptance rate to be within 5% of δ.

Each iteration $i \leq A$, HMCDA adapts ϵ and coerces the target acceptance rate δ. The number of leapfrog steps, L, is printed with each `Status` message. In **LaplacesDemon**, all thinned samples are returned, including adaptive samples. This allows the user to examine adaptation. To obtain strictly non-adaptive samples after a previous update that included adaptation, simply update again with `A=0` and the last value of ϵ. `epsilon=NULL` is recommended.

As with HMC, the HMCDA algorithm is slower per iteration than many other algorithms, but often produces chains with good mixing. HMCDA should outperform Adaptive Hamiltonian Monte Carlo (AHMC), and iterates faster as well, unless L becomes large. When mixing is inadequate, consider switching to the MALA or NUTS algorithm. When parameters are highly correlated, another algorithm should be preferred in which correlation is taken into account, such as MALA, or in which the algorithm is generally invariant to correlation, such as twalk. When adaptive, it is not suitable as a final algorithm.

Hit-And-Run Metropolis

The Hit-And-Run algorithm is a variation of Random-Walk Metropolis (RWM) that has been around at least as long as Gibbs sampling (Gibbs). Hit-And-Run randomly samples a direction on the unit sphere as in Gilks and Roberts (1996), and a proposal is made for each parameter in its randomly-selected direction for a uniformly-distributed distance. This version of Hit-And-Run, called Hit-And-Run Metropolis (HARM), includes multivariate proposals and a Metropolis step for rejection. Introduced by Turchin (1971) along with Gibbs sampling, and popularized by Smith (1984), Hit-And-Run was given its name later due to its ability to run across the state-space and arrive at a distant "hit-point". It is related to other algorithms with interesting names, such as Hide-And-Seek and Shake-And-Bake.

HARM has two optional algorithm specifications: `alpha.star` is the target acceptance rate, and `B` is a list of blocked parameters. When `alpha.star=NULL`, non-adaptive HARM is used. Otherwise, the Robbins-Monro stochastic approximation of Garthwaite *et al.* (2010) is applied to the proposal distance. `alpha.star=0.234` is recommended. For blockwise sampling, each component of the list is a block and consists of a vector indicating the position of the parameters per block.

HARM is the fastest MCMC algorithm per iteration in this package, because it is very simple.

For example, HARM does not use a proposal covariance matrix, and there are no tuning parameters, with one optional exception discussed below. The proposal is multivariate in the sense that all parameters are proposed at once, though from univariate draws. HARM often mixes better than the Gibbs sampler (Gilks and Roberts 1996), especially with correlated parameters (Chen and Schmeiser 1992).

Adaptive HAR (without the Metropolis step) with a multivariate proposal is available in the `LaplaceApproximation` function.

The HARM algorithm is able to traverse complex spaces with bounded sets in one iteration. As such, HARM may work well with multimodal posteriors due to potentially good mode-switching behavior. However, HARM may have difficulty sampling regions of high probability that are spike-shaped or near constraints, and this difficulty is likely to be more problematic in high dimensions. When HARM is non-adaptive, it may be used as a final algorithm.

Independence Metropolis

Proposed by Hastings (1970) and popularized by Tierney (1994), the Independence Metropolis (IM) algorithm (also called the independence sampler) is an algorithm in which the proposal distribution does not depend on the previous state or iteration. The proposal distribution must be a good approximation of the target distribution for IM to peform well, and the proposal distribution should have slightly heavier tails than the target distribution. IM is used most often to obtain additional posterior samples given an algorithm that has already converged.

IM has one algorithm specification: `mu`. The usual approach to IM is to update the model with Laplace Approximation, and then supply the posterior mode and covariance to IM. The posterior mode vector of Laplace Approximation becomes the `mu` argument in the algorithm specifications for IM. The covariance matrix from Laplace Approximation is expanded by multiplying it by 1.1 so that it has heavier tails. Each iteration, IM draws from a multivariate normal distribution as the proposal distribution. Alternatively, posterior means and covariances may be used from other algorithms, such as other MCMC algorithms.

Since IM is non-adaptive and uses a proposal distribution that remains fixed for all iterations, it may be used as a final algorithm.

Interchain Adaptation

The Interchain Adaptation (INCA) algorithm of Craiu *et al.* (2009) is an extension of the Adaptive Metropolis (AM) algorithm of Haario *et al.* (2001). Craiu *et al.* (2009) refer to INCA as inter-chain adaptation and inter-chain adaptive MCMC. INCA uses parallel chains that are independent, except that they share the adaptive component, and this sharing speeds convergence. Since parallel chains are a defining feature of INCA, this algorithm works only with `LaplacesDemon.hpc`, not `LaplacesDemon`.

INCA has two algorithm specifications: `Adaptive` is the iteration in which it becomes adaptive, and `Periodicity` is the frequency in iterations of adaptation.

As with AM, the proposal covariance matrix is set equal to the historical (or sample) covariance matrix. Ample pre-adaptive iterations are recommended (Craiu *et al.* 2009), and initial values should be dispersed to aid in discovering multimodal marginal posterior distributions. After adaptation begins, INCA combines the historical covariance matrices from all parallel

chains during each adaptation. Each chain learns from experience as in AM, and in INCA, each chain also learns from the other parallel chains.

Solonen *et al.* (2012) found a dramatic reduction in the number of iterations to convergence when INCA used 10 parallel chains, compared against a single-chain AM algorithm. Similar improvements have been noted in the **LaplacesDemon** package, though more time is required per iteration.

The `Gelman.Diagnostic` is recommended by Craiu *et al.* (2009) to determine when the parallel chains have stopped sharing different information about the target distributions. The exchange of information between chains decreases as the number of iterations increases.

This implementation of INCA uses a server function that is built into `LaplacesDemon.hpc`. If the connection to this server fails, then the user must kill the process and then close all open connections with the `closeAllConnections` function.

Since INCA is an adaptive algorithm, the final algorithm should be Random-Walk Metropolis (RWM).

Metropolis-Adjusted Langevin Algorithm

Also called Langevin Monte Carlo (LMC), the Metropolis-Adjusted Langevin Algorithm (MALA) was proposed in Roberts and Tweedie (1996), and an adaptive version in Atchade (2006), and an alternative adaptive version in Shaby and Wells (2010). MALA was inspired by stochastic models of molecular dynamics. MALA is an extension of the multivariate Random-Walk Metropolis (RWM) algorithm that includes partial derivatives to improve mixing. Roberts and Tweedie (1996) presented ULA, MALA, and MALTA, and recommended MALTA. MALTA is a refinement of MALA that uses a truncated drift, where the drift parameter is a step-size parameter for the partial derivatives. The non-adaptive version of MALA is equivalent to the Hamiltonian Monte Carlo (HMC) algorithm with `L=1` leapfrog steps. For more information on HMC, see section 12.5.17.

The original, non-adaptive MALA family is difficult to tune, and optimal tuning differs between transience and stationarity. For this reason, the MALA presented here is the adaptive version presented in Atchade (2006). This adaptive MALA uses stochastic approximation to update an expectation, scale, and covariance every iteration. The scale of the proposal is adapted to obtain a target acceptance rate.

This version of MALA has five algorithm specifications: `A` defaults to 1e7 as the maximum acceptable value of the Euclidean norm of the adaptive parameters mu and sigma, and the Frobenius norm of the covariance matrix, `alpha.star` is the target acceptance rate and defaults to 57.4%, `gamma` accepts a constant in $[1, T]$ where T is iterations and controls decay in adaptation, `delta` defaults to 1 and is a constant in the bounded drift function, and `epsilon` is a vector of length two that defaults to `c(1e-6,1e-7)`, in which the first element is the acceptable minimum of adaptive scale sigma, and the second element is added to the diagonal of the covariance matrix for regularization. The expectation, scale, and covariance adapt each iteration, and the amount of adaptation is a decreasing function $1/t$ for T iterations. Larger values of `gamma` result in less decay in adaptation. The optimal acceptance rate is 57.4%, and is acceptable in the interval [40%, 80%].

MALA approximates partial derivatives for multivariate proposals. Approximating partial derivatives is computationally expensive, and requires $J + 1$ model evaluations per J parameters per iteration. This results in a multivariate algorithm that is slightly slower per

iteration than a traditional componentwise algorithm, though the higher acceptance rate and additional information from partial derivatives may allow it to mix better and approach convergence faster. Unlike most componentwise algorithms, this version of MALA accounts for parameter correlation.

Since this version of MALA is adaptive, it is recommended that the non-adaptive version in HMC (with `L=1`) is used as a final algorithm.

Metropolis-Coupled Markov Chain Monte Carlo

The Metropolis-Coupled Markov Chain Monte Carlo (MCMCMC) algorithm – also referred to as MC^3, $(MC)^3$, Metropolis-Coupled MCMC, Parallel Tempering, or Replica Exchange – was introduced by Geyer (1991) for multimodal distributions. The name 'parallel tempering' was introduced in Earl and Deem (2005).

MCMCMC consists of parallel chains that use different temperatures and combine to form a mixture distribution, where each chain is a mixture component. The chains or mixture components are updated in parallel, each with a Metropolis step that is adjusted by temperature. After this first set of Metropolis steps, two parallel chains are selected at random, and another adjusted Metropolis step is used to accept or reject a swap between chains. The act of swapping is referred to as coupling. Each swap may be a jump between modes. MCMCMC has both within-chain and between-chain proposals each iteration, but only the coolest chain is retained.

MCMCMC has four algorithm specifications: `lambda` is either a positive-only scalar that controls the temperature spacing or a vector of temperature spacings, `CPUs` is the number of central processing units in the computer, `Packages` is a vector of any package names that are required, and `Dyn.libs` are any dynamic libaries that are required. A larger scalar value of λ results in greater differences in temperature between chains, and often a lower swap acceptance rate (see below). Given a scalar λ and $m = 1, ..., M$ CPUs, the current or proposal distribution is exponentiated to this power: $1/[1 + \lambda(m - 1)]$. The number of CPUs must be at least two; as programmed, MCMCMC will not function on a single-core computer.

MCMCMC must be called with `LaplacesDemon`, not `LaplacesDemon.hpc`, even though MCMCMC requires parallel chains. Each iteration, LaplacesDemon communicates with all CPUs, collects the latest for all the chains, and then the master process calculates the Metropolis steps for each chain, as well as the swap.

As with Random-Walk Metropolis (RWM), the proposal covariance matrix must be tuned. When nothing is known about this matrix, as is often the case, it is suggested to begin with an identity matrix with small-scale diagonal values, such as 1e-3. After a short run that hopefully has some acceptances, another short run can begin with the observed historical covariance matrix. Eventually, the proposal covariance matrix is usually satisfactory.

Atchade, Roberts, and Rosenthal (2011) demonstrate that MCMCMC performance improves as the acceptance rate of proposed swaps approaches 0.234. In <code>LaplacesDemon</code>, the swap acceptance rate is printed to the console at the end of each update. The swap acceptance rate is affected by the temperature spacing between parallel chains. If the default temperature spacing that results from a scalar λ is unacceptable, then a different scalar value may be attempted, or the user may enter their own temperature spacing directly as a vector for λ.

Coupling induces dependence among the chains, and the chains are no longer Markovian. The

whole stochastic process of m chains together does form a Markov chain.

Along with the J-walking algorithm of Frantz, Freeman, and Doll (1990), MCMCMC was one of the first extensions of Metropolis-Hastings for multimodal distributions. Many additional MCMC algorithms, such as serial tempering or simulated tempering, were influenced by or variations of MCMCMC.

The advantage of MCMCMC over RWM is that MCMCMC is better able to approximate multimodal distributions, and that successful coupling (swaps) improves mixing. A disadvantage is that most of the information in the warmer chains is lost, speed per iteration is slower than RWM due to communication with CPUs, and distributions with many modes require at least as many CPUs. Since MCMCMC is not adaptive, it is suitable as a final algorithm.

Metropolis-within-Gibbs

Metropolis-within-Gibbs (MWG) is the original MCMC algorithm, introduced in Metropolis *et al.* (1953). Since it was the original MCMC algorithm, it pre-dated Gibbs sampling, and was not known as Metropolis-within-Gibbs. MWG was later proposed as a hybrid algorithm that combines Metropolis-Hastings and Gibbs sampling, and was suggested in Tierney (1994). The idea was to substitute a Metropolis step when Gibbs sampling fails. However, Gibbs sampling is not included in this version in `LaplacesDemon` (or most versions), making it an algorithm with a misleading name. Without Gibbs sampling, the more honest name would be componentwise random-walk Metropolis, but the more common name is MWG. MWG is also referred to as Metropolis within Gibbs, Metropolis-in-Gibbs, Random-Walk Metropolis-within-Gibbs, single-site Metropolis, the Metropolis algorithm, or Variable-at-a-Time Metropolis.

MWG is a componentwise algorithm, meaning that each parameter is updated indvidually each iteration. This implies that the model specification function is evalulated a number of times equal to the number of parameters, per iteration. The order of the parameters for updating is randomized each iteration (random-scan MWG), as opposed to sequential updating (deterministic-scan MWG). MWG often uses blocks, but in **LaplacesDemon**, all blocks have dimension 1, meaning that each parameter is updated in turn. If parameters were grouped into blocks, then they would undesirably share a proposal standard deviation. A componentwise algorithm typically ignores correlated parameters, updating each parameter individually.

The update of each parameter occurs in a Metropolis step, otherwise called an accept/reject step or a Metropolis accept/reject step. A componentwise proposal is generated randomly and the model is evaluated with the proposed parameter. If the proposal is an improvement in the logarithm of the unnormalized joint posterior density, then the proposal is accepted. If the proposal is not an improvement, then the proposal is accepted or rejected according to a probability.

The acceptance rate is the total number of proposals accepted in the Metropolis step divided by the total number of iterations. If the acceptance rate is too high, then the proposal variance is too small. In this case, the algorithm will take longer than necessary to find the target distribution and the samples will be highly autocorrelated. If the acceptance rate is too low, then the proposal variance is too large, and the algorithm is ineffective at exploration. In the worst case scenario, no proposals are accepted and the algorithm fails to move. Under theoretical conditions, the optimal acceptance rate for a sole, independent and identically

distributed (IID), Gaussian, marginal posterior distribution is 0.44 or 44%. The optimal acceptance rate for an infinite number of distributions that are IID and Gaussian is 0.234 or 23.4%. Since MWG is a componentwise algorithm, it is most efficient when the acceptance rate of each parameter is 0.44.

MWG is a componentwise Random-Walk Metropolis (RWM) algorithm. Random-walk behavior is desirable because it allows the algorithm to explore, and hopefully avoid getting trapped in undesirable regions. On the other hand, random-walk behavior is undesirable because it takes longer to converge to the target distribution while the algorithm explores. The algorithm generally progresses in the right direction, but may periodically wander away. Such exploration may uncover multimodal target distributions, which other algorithms may fail to recognize, and then converge incorrectly. With enough iterations, MWG is guaranteed theoretically to converge to the correct target distribution, regardless of the starting point of each parameter, provided the proposal variance for each proposal of a target distribution is sensible.

Historically, MWG first ran on an early computer that was built specifically for it, called MANIAC I. Metropolis discarded the first 16 iterations as burn-in, and updated an additional 48-64 iterations, which required 4-5 hours on MANIAC I.

The advantage of MWG over its multivariate version, RWM, is that it is more efficient with information per iteration, so convergence is faster in iterations. The disadvantages of MWG are that covariance is not included in proposals, and it is more time-consuming due to the evaluation of the model specification function for each parameter per iteration. As the number of parameters increases, and especially as model complexity increases, the run-time per iteration increases. Since fewer iterations are completed in a given time-interval, the possible amount of thinning is also at a disadvantage. MWG is non-adaptive, and is suitable as a final algorithm.

Multiple-Try Metropolis

The Multiple-Try Metropolis (MTM) algorithm was introduced in Liu *et al.* (2000) as a componentwise Metropolis algorithm with an improved acceptance rate due to an increased proposal variance and number of proposals. The user specifies that K proposals will be made. For each parameter at each iteration, K normally-distributed proposals are made around the current parameter, scaled according to the user-specified proposal variance. Each proposal is weighted according to its unnormalized joint posterior density. One proposal is selected randomly, with probability proportional to the weights. A reference set of length K is created, in which the first $K-1$ elements are draws from a normal distribution centered around the selected proposal, again according to proposal variance. The last element, K, is the selected proposal itself. A Metropolis step is performed for the weighted reference set. If the weighted reference set is accepted, then the selected proposal becomes the new value for the parameter.

MTM has four algorithm specifications: `K` is the number of proposals, `CPUs` is the number of CPUs, `Packages` accepts any package required for the model function, and `Dyn.libs` accepts dynamic libraries for parallelization, if required.

(Liu *et al.* 2000) demonstrate the combinination of MTM with the Snooker algorithm from Adaptive Directional Sampling (ADS), Conjugate Gradient Monte Carlo (CGMC), Hit-And-Run modified as Random-Ray Monte Carlo (RRMC), and Griddy-Gibbs (GG). MTM has since been extended to multivariate proposals, proposals with different scales, and more.

Advantages of MTM over Metropolis-within-Gibbs (MWG) is that the acceptance rate is higher, and multiple evaluations of the model specification function are parallelized each iteration. The advantage of MTM over Griddy-Gibbs (GG) is exact rather than approximate estimation and that an equilibrium distribution cannot be guaranteed from an approximation of the conditional such as in GG. A disadvantage of MTM compared to MWG is that MTM must evaluate the model specification function multiple times per parameter per iteration, resulting in an algorithm that is slower per iteration. Since MTM is not adaptive, it is suitable as a final algorithm.

No-U-Turn Sampler

The No-U-Turn Sampler (NUTS) is an extension of Hamiltonian Monte Carlo (HMC) that adapts both the scalar step-size ϵ and scalar number of leapfrog steps L. This is algorithm #6 in Hoffman and Gelman (2012). For more information on HMC, see section 12.5.17.

NUTS has three algorithm specifications: the number of adaptive iterations `A`, the target acceptance rate `delta` or δ (and 0.6 is recommended), and a scalar step-size `epsilon` or ϵ. When `epsilon=NULL`, a reasonable initial value is found. The `Consort` function requires the acceptance rate to be within 5% of δ.

Each iteration $i \leq A$, NUTS adapts both ϵ and L, and coerces the target acceptance rate δ. L continues to change after adaptation ends, but is not an adaptive parameter in the sense of destroying ergodicity. The adaptive samples are discarded and only the thinned non-adaptive samples are returned.

If NUTS begins (or begins again) with a value of ϵ that is too small, then early iterations may take a very long time, since L may be very large before a u-turn is found. `epsilon=NULL` is recommended. It is also recommended, in complex models, to set `Status=1`. The time per iteration varies greatly by iteration as NUTS searches for L. If NUTS seems to hang at an iteration, then most likely L is becoming large, and the search for it is becoming time-consuming. It may be recommended to cancel the update and try HMCDA, which will print L to the screen with each `Status` message. Though these are different algorithms, this may help the user decide whether or not to try a different algorithm. The AHMC algorithm will not perform as well, but offers consistent time per iteration.

The main advantage of NUTS over other HMC algorithms is that NUTS is the algorithm most likely to produce approximately independent samples, in the sense of low autocorrelation. Due to computational complexity, NUTS is slower per iteration than HMC, and the HMC family is among the slowest. Despite this, NUTS often produces chains with excellent mixing, and should outperform other adaptive versions of HMC, such as AHMC and HMCDA. Per iteration, NUTS should generally perform better than any other HMC algortihm. Per minute, however, is another story.

NUTS has been extended elsewhere to allow for a non-diagonal mass matrix (proposal covariance matrix for momentum), and allowing the user to limit L. These extensions are not yet included here.

In complex and high-dimensional models, NUTS may produce approximately independent samples much more slowly in minutes than other MCMC algorithms, such as Adaptive Metropolis-within-Gibbs (AMWG). This is because the combination of calculating partial derivatives and the search each iteration for L is computationally intensive.

Oblique Hyperrectangle Slice Sampler

Oblique Hyperrectangle Slice Sampler (OHSS) is an adaptive algorithm that approximates the sample covariance matrix. Introduced by Thompson (2011), OHSS is an extension of the hyperrectangle method in Neal (2003), which in turn is a multivariate extension of the slice sampler (Slice). The initial hyperrectangle is oriented to have edges that are parallel to the eigenvectors of the sample covariance matrix. The initial hyperrectangle is also scaled to have lengths proportional to the square roots of the eigenvalues of the sample covariance matrix. Rejection sampling is performed, and when a sample is rejected, the slice approximation is shrunk. Aside from specifying the initial hyperrectangle, this is the hyperrectangle method in Neal (2003).

OHSS does not have any algorithm specifications.

Each iteration, there is a 5% probability that a non-adaptive step is taken, rather than an adaptive step. The first ten iterations or so are non-adaptive. Eigenvectors are estimated no more than every tenth iteration. When adaptive, the sample covariance matrix is updated with each thinned iteration.

Once the slice interval is estimated, a sample is drawn uniformly with rejection sampling from within this interval. If rejected, then the interval is shrunk until an acceptable sample is drawn. OHSS has a 100% acceptance rate.

The time per iteration varies, since rejection sampling often requires more than one evaluation. Since OHSS is adaptive, it is not suitable as a final algorithm.

Random Dive Metropolis-Hastings

Random Dive Metropolis-Hastings (RDMH) was introduced by Dutta (2012). RDMH is a componentwise Metropolis-Hastings algorithm in which the proposal is the product or ratio of a current parameter and a random quantity that does not require any tuning parameters. The random quantity is in the interval $(-1, 1)$. Although RDMH is a very simple algorithm, it has excellent convergence and mixing properties. However, if it is reasonable that the origin (0) has positive probability, then the model should be reparameterized, because RDMH fails in the obscure case when the origin has positive probability (which can arise if the state-space is the set of integers).

RDMH does not have any algorithm specifications.

RDMH allows the proposed state to be far away from the current state, and yet has a good acceptance rate. Therefore, RDMH can explore the state space faster than many other MCMC algorithms.

RDMH is geometrically ergodic for a class of densities that is much larger than most other MCMC algorithms. RDMH excels at representing target densities that are multimodal or fat-tailed, and has been compared with Gibbs sampling, Metropolis-Adjusted Langevin Algorithm (MALA), and Metropolis-within-Gibbs (MWG). Each of these other algorithms became stuck at local modes in multimodal target densities with large distances between the modes. RDMH explored the multimodal densities, or the fat tails, with ease.

As a componentwise algorithm, the model specification function is evaluated a number of times equal to the number of parameters, per iteration. The order of the parameters for updating is randomized each iteration (random-scan), as opposed to sequential updating (deterministic-scan).

The advantages of RDMH over MWG are that RDMH does not require tuning, better explores multimodal and fat-tailed target distributions, is better able to find acceptable proposals that are distant, and that it may therefore explore the state-space faster. Compared to multivariate MCMC algorithms, RDMH shares common disadvantages that it is slower per iteration, and correlated parameters are not taken into account. Since RDMH is not adaptive, it is suitable as a final algorithm.

Random-Walk Metropolis

Random-Walk Metropolis (RWM) is a multivariate extension of Metropolis-within-Gibbs (MWG). Multiple parameters usually exist, and therefore correlations may occur between the parameters. Many MCMC algorithms attempt to estimate multivariate proposals, usually taking correlations into account through a covariance matrix. Each iteration, a multivariate proposal is made by taking a draw from a multivariate normal distribution, given a proposal covariance matrix.

RWM does not have any required algorithm specifications, though blockwise sampling may be specified with `B`, which accepts a list of proposal covariance matrices equal in length to the number of blocks. By default, blockwise sampling is not performed, so all parameters are updated with one multivariate proposal.

Given the number of dimensions (K) or parameters, the optimal scale of the proposal covariance, also called the jumping kernel, has been reported as $2.4^2/K$ based on the asymptotic limit of infinite-dimensional Gaussian target distributions that are independent and identically-distributed (Gelman, Roberts, and Gilks 1996b). In applied settings, each problem is different, so the amount of correlation varies between variables, target distributions may be non-Gaussian, the target distributions may be non-IID, and the scale should be optimized. There are algorithms in statistical literature that attempt to optimize this scale, such as the Robust Adaptive Metropolis (RAM) algorithm.

There have been numerous methods introduced for tuning the proposal covariance matrix. Many adaptive MCMC algorithms such as Adaptive Metropolis (AM), Adaptive-Mixture Metropolis (AMM), and RAM will tune the proposal covariance matrix. Alternatively, initially specify an identity matrix with small-scale diagonal values such as 1e-3 as the proposal covariance matrix, update with RWM, and then update again but this time with the observed covariance of the historical samples. Done repeatedly, this may arrive at an acceptable proposal covariance matrix, suitable for longer and more serious updates.

Since RWM is not adaptive, it is suitable as a final algorithm.

Reflective Slice Sampler

Introduced in Neal (1997), the Reflective Slice Sampler (RSS) is a multivariate slice sampler that uses partial derivatives and reflects into a slice interval. As described in Neal (2003), each iteration, direction and momentum within the slice interval are determined randomly, partial derivatives are taken once, and a number of steps is taken according to a step size. The distance of each step is the product of the step size and momentum. Direction is changed by reflecting off the boundaries of the slice interval. The reflected momentum direction is a function of the incident momentum and the partial derivatives. Random-walk behavior is suppressed, because a large number of steps may be taken in the same direction. The acceptance rate is 100%.

RSS has two algorithm specifications: m and w. Each iteration, m steps are taken with step size w. While m must be a scalar, w may be a scalar or vector equal in length to the number of parameters. A step size that is too small is inefficient, but too small is better than too large, which can entirely avoid the target distribution.

RSS is difficult to tune, but is consistent in time per iteration. citetneal03 states that Hamiltonian Monte Carlo (HMC) performs better than RSS when both HMC and RSS are optimally tuned. Since RSS is not adaptive, it may be used as a final algorithm.

Refractive Sampler

Refractive sampling was introduced by Boyles and Welling (2012) as an alternative to Hamiltonian Monte Carlo (HMC) and the No-U-Turn Sampler (NUTS). For more information on HMC or NUTS, see sections 12.5.17 or 12.5.26, respectively. While the refractive sampler, HMC, and NUTS all use partial derivatives, the refractive sampler uses partial derivatives only for direction, not magnitude. The partial derivatives always point toward the side with the higher index of refraction. Like HMC and unlike NUTS, the refractive sampler requires tuning.

This version of the Refractive algorithm has four algorithm specifications: `Adaptive` indicates the first adaptive iteration, the number `m` of steps to take per iteration, step size `w`, and `r` is the ratio between indices of refraction. The `Adaptive` argument does not appear in Boyles and Welling (2012). When `Adaptive` is less than the number of iterations, an optional Robbins-Monro stochastic approximation of Garthwaite *et al.* (2010) is applied to step size `w`. All specifications must be scalar, and it is recommended that `r=1.3`. The number of steps `m` is similar to the number of leapfrog steps `L` in HMC, and step size `w` is similar to `epsilon`. Typically, as the number of parameters increases, it is often better for `m` to be larger and `w` smaller.

Compared to other MCMC algorithms, a higher posterior density is often found. An advantage over NUTS is that iterative speed is consistent. An advantage over HMC is that Refractive is easier to tune. When `Adaptive` is greater than the number of iterations, the Refractive sampler is not adaptive and is suitable as a final algorithm.

Reversible-Jump

Reversible-jump Markov chain Monte Carlo (RJMCMC) was introduced by Green (1995) as an extension to MCMC in which the dimension of the model is uncertain and to be estimated. Reversible-jump belongs to a family of trans-dimensional algorithms, and it has many applications, including variable selection, model selection, mixture component selection, and more. The RJ algorithm, here, is one of the simplest possible implementations and is intended for variable selection and Bayesian Model Averaging (BMA).

The Componentwise Hit-And-Run Metropolis (CHARM) algorithm (see section 12.5.10) was selected, here, to be extended with reversible-jump. CHARM was selected because it does not have tuning parameters, it is not adaptive (which simplifies things with RJ), and it performs well. Even though it is a componentwise algorithm, it does not evaluate all potential predictors each iteration, but only those that are included. This novel combination is Reversible-Jump (RJ) with Componentwise Hit-And-Run Metropolis (CHARM). The optimal acceptance rate, and a good suggested acceptance rate range, are unknowns, so the `Consort` function will be little help making suggestions here.

RJ proceeds by making two proposals during each iteration. First, a within-model move is proposed. This means that the model dimension does not change, and the algorithm proceeds like a traditional CHARM algorithm. Next, a between-models move is proposed, where a selectable parameter is sampled, and its status in the model is changed. If this selected parameter is in the current model, then RJ proposes a model that excludes it. If this selected parameter is not in the current model, then RJ proposes a model that includes it. RJ also includes a user-specified binomial prior distribution for the expected size of the model (the number of included, selectable parameters), as well as user-specified prior probabilities for the inclusion of each of the selectable parameters.

Behind the scenes, RJ keeps track of the most recent non-zero value for each selectable parameter. If a parameter is currently excluded, then its value is currently set to zero. When this parameter is proposed to be included, the most recent non-zero value is used as the basis of the proposal, rather than zero. In this way, an excluded parameter does not have to travel back toward its previously included density, which may be far from zero. However, if RJ begins updating after a previous run had ended, then it will not begin again with this most recent non-zero value. Please keep this in mind with this implementation of RJ.

RJ has five algorithm specifications. `bin.n` is the scalar size parameter of the binomial prior distribution for model size, and is the maximum size that RJ will explore. `bin.p` is the scalar probability parameter of the binomial prior distribution for model size, and the mean or median expected model size is `bin.n` $\times$ `bin.p`. `parm.p` is a vector of parameter-inclusion probabilities that must be equal in length to the number of initial values. `selectable` is a vector of indicators (0 or 1) that indicate whether or not a parameter is selectable by reversible-jump. When an element is zero, it is always in the model. Finally, the `selected` vector contains indicators of whether or not each parameter is in the model when RJ begins to update. All of these specifications must receive an argument with exactly that name (such as `bin.n=bin.n`, for the `Consort` function to recognize it, with the exception of the `selected` specification.

RJ provides a sampler-based alternative to variable selection, compared with the Bayesian Adaptive Lasso (BAL) or Stochastic Search Variable Selection (SSVS), as two of many other approaches. Examples of BAL and SSVS are in the Examples vignette. Advantages of RJ are that it is easier for the user to apply to a given model than writing the variable-selection code into the model, and RJ requires fewer parameters, because variable-inclusion is handled by the specialized sampler, rather than the model specification function. A disdvantage of RJ is that other methods allow the user to freely change to other MCMC algorithms, if the current algorithm is unsatisfactory.

Robust Adaptive Metropolis

The Adaptive Metropolis (AM) and Adaptive-Mixture Metropolis (AMM) algorithms adapt the scale of the proposal distribution to attain a theoretical acceptance rate. However, these algorithms are unable to adapt to the shape of the target distribution. The Robust Adaptive Metropolis (RAM) algorithm estimates the shape of the target distribution and simultaneously coerces the acceptance rate (Vihola 2011). If the acceptance probability, α, is less (or greater) than an acceptance rate target, $\alpha*$, then the proposal distribution is shrunk (or expanded). Matrix $\mathbf{S}$ is computed as a rank one Cholesky update. Therefore, the algorithm is computationally efficient up to a relatively high dimension. The AM and AMM algorithms

require a burn-in period prior to adaptation, so that these algorithms can adapt to the sample covariance. RAM does not require a burn-in period prior to adaptation. RAM allows the user the option of using the traditional normally-distributed proposals, or t-distributed proposals for heavier-tailed target densities. Unlike AM and AMM, RAM can cope with targets having arbitrarily heavy tails, and handles multimodal targets better than AM. The user is still assumed to know and specify the target acceptance rate.

RAM has four algorithm specifications: `alpha.star` is the target acceptance rate, `Dist` is the target distribution as either `"N"` for normal or `"t"` for the Student t with 5 degrees of freedom, `gamma` accepts a scalar in the interval (0.5,1] and controls the decay of adaptation (0.66 is recommended), and `Periodicity` specifies the frequency of adaptation in iterations. RAM adapts only when the variance-covariance matrix is positive-definite. The algorithm performs best when `Periodicity=1`, though each iteration is slower due to more computation.

The advantages of RAM over AMM are that RAM does not require a burn-in period before it can begin to adapt, RAM is more likely to better handle multimodal or heavy-tailed targets, RAM also adapts to the shape of the target distributions, and attempts to coerce the acceptance rate. The advantages of RAM over Adaptive Metropolis-within-Gibbs (AMWG) are that RAM takes correlations into account, and is much faster to update each iteration. The disadvantage of RAM compared to AMWG is that more information must be learned in the covariance matrix to adapt properly (see section 12.5.7 for more), and frequent adaptation may be desirable, but slow. If RAM is used for adaptation, then the final, non-adaptive algorithm should be Random-Walk Metropolis (RWM).

Sequential Adaptive Metropolis-within-Gibbs

The Sequential Adaptive Metropolis-within-Gibbs (SAMWG) algorithm is for state-space models (SSMs), including dynamic linear models (DLMs). It is identical to the Adaptive Metropolis-within-Gibbs (AMWG) algorithm, except with regard to the order of updating parameters (and here, sequential does not refer to deterministic-scan). Parameters are grouped into two blocks: static and dynamic. At each iteration, static parameters are updated first, followed by dynamic parameters, which are updated sequentially through the time-periods of the model. The order of the static parameters is randomly selected at each iteration, and if there are multiple dynamic parameters for each time-period, then the order of the dynamic parameters is also randomly selected. The argument `Dyn` receives a $T \times K$ matrix of T time-periods and K dynamic parameters. The SAMWG algorithm is adapted from Geweke and Tanizaki (2001) for `LaplacesDemon`. The SAMWG is a single-site update algorithm that is more efficient in terms of iterations, though convergence can be slow with high intercorrelations in the state vector (Fearnhead 2011). If SAMWG is used for adaptation, then the final, non-adaptive algorithm should be Sequential Metropolis-within-Gibbs (SMWG).

Sequential Metropolis-within-Gibbs

The Sequential Metropolis-within-Gibbs (SMWG) algorithm is the non-adaptive version of the Sequential Adaptive Metropolis-within-Gibbs (SAMWG) algorithm, and is used for final sampling of state-space models (SSMs).

Slice Sampler

The origin of slice sampling dates back to Besag and Green (1993), and the current algorithm

was introduced in Neal (1997) and improved in Neal (2003). In slice sampling, a distribution is sampled by sampling uniformly from the region under the plot of its density function. Here, slice sampling uses two phases as follows. First, an interval is created for the slice, and second, rejection sampling is performed within this interval.

Slice has two algorithm specifications: `m` is the number of steps (an integer, and defaults to `Inf`), and `w` is the step size (and defaults to 1). Both `m` and `w` may be specified with either a scalar or vector, so that each parameter receives different settings. The step size `w` may be in the interval $(0, \infty)$. Ideally, `w` is 3 standard deviations of the target distribution.

The Slice algorithm implemented here is componentwise and based on figures 3 and 5 in Neal (2003), in which the slice is replaced with an interval I that contains most of the slice. For each parameter, an interval I is created and expanded by the "stepping out" procedure with step size w until the interval is larger than the slice, and the number of steps m may be limited by the user. The original slice sampler is inappropriate for the unbounded interval $(-\infty, \infty)$, and this improved version allows this unbounded interval by replacing the slice with the created interval I. This algorithm adaptively changes the scale, though it is not an adaptive algorithm in the sense that it retains the Markov property.

The lower and upper bounds of interval I default to $(-\infty, \infty)$, and adjust automatically to constrained parameters. Once the interval is constructed, a proposal is drawn randomly from within the interval until the proposal is accepted, and the interval is otherwise shrunk. The acceptance rate of this Slice algorithm is 1, though multiple model evalutations occur per iteration.

When this Slice sampler uses the default specifications and begins far from the target distributions, the time per iteration should decrease as the algorithm approaches the target distributions. Considerable time may be spent in the first iterations. One strategy may be to limit m.

This componentwise Slice algorithm has been noted to be more efficient than Metropolis updates, may be easier to implement than Gibbs sampling (Gibbs), and is attractive for routine and automated use (Neal 2003). As a componentwise algorithm, it is slower per iteration than algorithms that use multivariate proposals. Multivariate slice samplers have been proposed, such as ESS (see 12.5.14), OHSS (see 12.5.27), RSS (see 12.5.30), and UESS (see 12.5.40). Since Slice is not an adaptive algorithm, it is acceptable as a final algorithm.

Stochastic Gradient Langevin Dynamics

The Stochastic Gradient Langevin Dynamics (SGLD) algorithm of Welling and Teh (2011) is the first MCMC algorithm designed specifically for big data. Traditional MCMC algorithms require the entire data set to be included in the model evaluation each iteration. In contrast, SGLD reads and processes only a small, randomly selected batch of records each iteration. In addition to saving computation time, the entire data set does not need to be loaded into memory at once.

SGLD expands the stochastic gradient descent optimization algorithm to include Gaussian noise with Langevin dynamics. The multivariate proposal is merely the vector of current values plus a step size times the gradient, plus Gaussian noise. The scale of the Gaussian noise is determined by the step size, ϵ.

SGLD has five algorithm specifications: `epsilon` or ϵ is the step size, `file` accepts the quoted name of a .csv file that is the big data set, `Nr` is the number of rows in the big data set, `Nc`

is the number of columns in the big data set, and `size` is the number of rows to be read and processed each iteration.

Since SGLD is designed for big data, the entire data set is not included in the `Data` list, but one small batch must be included and named `X`. All data must be included. For example, both the dependent variable **y** and design matrix **X** in linear regression are included. The requirement for the small batch to be in `Data` is so that numerous checks may be passed after `LaplacesDemon` is called and before the SGLD algorithm begins. Each iteration, SGLD uses the `scan` function, without headers, to read a random block of rows from, say, `X.csv`, stores it in `Data$X`, and passes it to the `Model` specification function. The `Model` function must differ from the other examples found in this package in that multiple objects, such as `X` and `y` must be read from `Data$X`, where usually there is both `Data$X` and `Data$y`.

The user tunes SGLD with step size ϵ via the `epsilon` argument, which accepts either a scalar for a constant step size or a vector equal in length to the number of iterations for an annealing schedule. The step size must remain in the interval (0,1). The user may define an annealing schedule with any function desired. Examples are given in Welling and Teh (2011) of decreasing schedules, as well as for using a constant. When $\epsilon = 0$, both the stochastic gradient and Langevin dynamics components of the equation are reduced to zero and the algorithm will not move. When ϵ is too large, degenerate results occur. A good recommendation seems to be to begin with ϵ set to `1/Nr`. From testing on numerous examples, it seems preferable to perform several short runs and experiment with a constant, rather than using a decreasing schedule, but your mileage may vary.

The SGLD algorithm presented here is the simplest one, which is equation 4 in Welling and Teh (2011). Other components were also proposed such as a preconditioning matrix and covariance matrix, though these are not currently included here.

SGLD does not include a Metropolis step for acceptance and rejection, so the acceptance rate is 100%. SGLD is slower than a componentwise algorithm for two reasons: first it must read data from an external file each iteration, and second, gradients with numerical differencing are computationally expensive, requiring many model evaluations per iteration. At least `Nr / size` iterations are suggested. SGLD has great promise for the application of MCMC to massive data sets.

Tempered Hamiltonian Monte Carlo

The Tempered Hamiltonian Monte Carlo (THMC) algorithm is an extension of Hamiltonian Monte Carlo (HMC) to include another algorithm specification: `Temperature`, which must be positive. When greater than 1, the algorithm should explore more diffuse distributions, and may be helpful with multimodal distributions.

There are a variety of ways to include tempering in HMC, and this algorithm, named here as THMC, uses "tempered trajectory", as described by Neal (2011). When $L > 1$ and during the first half of the leapfrog steps, the momentum is increased (heated) by multiplying it by $\sqrt{T}$, where T is `Temperature`, each leapfrog step. In the last half of the leapfrog steps, the momentum decreases (is cooled down) by dividing it by $\sqrt{T}$. The momentum is largest in the middle of the leapfrog steps, where mode-switching behavior becomes most likely to occur. This preserves the trajectory, ϵL.

As with HMC, THMC is a difficult algorithm to tune. Since THMC is non-adaptive, it is sufficient as a final algorithm.

t-walk

The t-walk (twalk) algorithm of Christen and Fox (2010) is a general-purpose algorithm that requires no tuning, is scale-invariant, is technically non-adaptive (but self-adjusting), and can sample from target distributions with arbitrary scale and correlation structures. A random subset of one of two vectors is moved around the state-space to influence one of two chains, per iteration.

twalk has four algorithm specifications: `SIV` is a vector secondary initial values and the default is `NULL`, `n1` affects the size of the subset of each set of points to adjust and the default is 4, `at` affects the traverse move and the default is 6, and `aw` affects the walk move and the default is 1.5. The vector of secondary initial values may be left to its default, `NULL`, in which case it is generated with the `GIV` function and it is recommended that `PGF` is specified. `SIV` should be similar to, but unique from, `Initial.Values`. The secondary initial values are used for a second chain, which is merely used here to help the first chain, and its results are not reported. The n_1 specification relates to the number of parameters. For example, if $n_1 = 4$ and a model has $J = 100$ parameters, then there is a $p(0.04) = 4/100$ probability that a point is moved that affects each parameter, though this affects only one of two chains per iteration. Put another way, there is a 4% chance that each parameter changes each iteration, and a 50% chance each iteration that the observed chain is selected. The traverse specification argument, a_t, affects the traverse move, which helps when some parameters are highly correlated, and the correlation structure may change through the state-space. The traverse move is associated with an acceptance rate that decreases as the number of parameters increases, and is the reason that n_1 is used to select a subset of parameters each iteration. Finally, the walk specification argument, a_w, affects the walk move. The authors recommend keeping these specification arguments in $n_1 \in [2, 20]$, $a_t \in [2, 10]$, and $a_w \in [0.3, 2]$. The hop and blow moves do not have specifications, but help with multimodality, ensure irreducibility, and prevent the two chains from collapsing together. The hop move is centered on the primary chain, and the blow move is centered on the secondary chain.

The authors have provided the t-walk algorithm in R code as well as other languages. It is called the "t-walk" for "traverse" or "thoughtful" walk, as opposed to Random-Walk Metropolis (RWM). Where adaptive algorithms are designed to adapt to the scale and correlation structure of target distributions, the t-walk is invariant to this structure. The step-size and direction continuously "adjust" to the local structure. Since it is technically non-adaptive, it may also be used as a final algorithm. The t-walk uses one of four proposal distributions or 'moves' per iteration, with the following probabilities: traverse (p=0.4918), walk (p=0.4918), hop (p=0.0082), and blow (p=0.0082).

Testing in **LaplacesDemon** with the default specifications suggests the t-walk is very promising, but due to the subset of proposals, it is important to note that the reported acceptance rate indicates the proportion of iterations in which moves were accepted, but that only a subset of parameters changed, and only one chain is selected each iteration. Therefore, a user who updates a high-dimensional model should find that parameter values change much less frequently, and this requires more iterations.

The main advantage of t-walk, like the Hit-And-Run Metropolis (HARM) and Metropolis-within-Gibbs (MWG) families, over multivariate adaptive algorithms such as Adaptive-Mixture Metropolis (AMM) and Robust Adaptive Metropolis (RAM) is that t-walk does not adapt to a proposal covariance matrix, which can be limiting in random-access memory (RAM) and

other respects in large dimensions, making t-walk suitable for high-dimensional exploration. Other advantages are that t-walk is invariant to all but the most extreme correlation structures, does not need to burn-in before adapting since it technically is non-adaptive (though it 'adjusts' continuously), and continuous adjustment is an advantage, so `Periodicity` does not need to be specified. The advantage of t-walk over componentwise algorithms such as the MWG family, is that the model specification does not have to be evaluated a number of times equal to the number of parameters in each iteration, allowing the t-walk algorithm to iterate significantly faster in high dimension. The disadvantage of t-walk, compared to these algorithms, is that more iterations are required because only a subset of parameters can change at each iteration (though it still updates twice the number of parameters per iteration, on average, than the MWG family).

Since twalk is technically non-adaptive, it is suitable as a final algorithm.

Univariate Eigenvector Slice Sampler

The Univariate Eigenvector Slice Sampler (UESS) is an adaptive algorithm that approximates the sample covariance matrix in the same manner as Adaptive-Mixture Metropolis (AMM). For more information on AMM, see section 12.5.8. Described as "Univar Eigen" in Thompson (2011), UESS is a multivariate variation of slice sampling (Slice) that makes univariate updates along the eigenvectors of the sample covariance matrix. For more information on the componentwise slice sampler, see section 12.5.36.

UESS has one algorithm specification: `m`. Each iteration, a slice interval is estimated with up to m steps. The default is $m = 100$ steps. The step size is affected by the eigenvectors of the sample covariance matrix.

The contours of the target distribution are approximately ellipsoidal and the eigenvectors of its covariance coincide roughly with the axes of these ellipsoids, provided that the shape of the target distribution is approximately convex. Steps taken along these eigenvectors include steps along the long axes of a slice.

Each iteration, there is a 5% probability that a non-adaptive step is taken, rather than an adaptive step. The first ten iterations or so are non-adaptive. Eigenvectors are estimated no more than every tenth iteration. When adaptive, the sample covariance matrix is updated with each thinned iteration.

Once the slice interval is estimated, a sample is drawn uniformly with rejection sampling from within this interval. If rejected, then the interval is shrunk until an acceptable sample is drawn. UESS has a 100% acceptance rate.

The time per iteration varies, since building the slice area requires up to m steps, and rejection sampling often requires more than one evaluation. Although UESS is a combination of the AMM and Slice algorithms, it usually performs as well or better than either.

Updating Sequential Adaptive Metropolis-within-Gibbs

The Updating Sequential Adaptive Metropolis-within-Gibbs (USAMWG) is for state-space models (SSMs), including dynamic linear models (DLMs). After a model is fit with Sequential Adaptive Metropolis-within-Gibbs (SAMWG) and Sequential Metropolis-within-Gibbs (SMWG), and information is later obtained regarding the first future state predicted by the model, the USAMWG algorithm may be applied to update the model given the new informa-

tion. In SSM terminology, updating is filtering and predicting. The `Begin` argument tells the sampler to begin updating at a specified time-period. This is more efficient than re-estimating the entire model each time new information is obtained.

Updating Sequential Metropolis-within-Gibbs

The Updating Sequential Metropolis-within-Gibbs (USMWG) algorithm is the non-adaptive version of the USAMWG algorithm, and is used for final sampling when updating state-space models (SSMs).

Sampler Selection

The optimal sampler differs for each problem, and it is recommended that the `Juxtapose` function is used to help select the least inefficient MCMC algorithm. Nonetheless, some general observations here may be helpful to a user attempting to select the most appropriate sampler for a given model. Suggestions in this section have been reached by attempting to compare all samplers on most models in the accompanying "Examples" vignette. Comparisons consisted of

- diminishing adaptation, if applicable
- how many iterations it took the sampler to seem to converge
- how many minutes it took the sampler to seem to converge
- how quickly the sampler improved in the beginning
- `Juxtapose` results based on integrated autocorrelation time (`IAT`)
- mixing of the chains
- whether or not the sampler arrived at the correct solution

When the user is ready to select a general-purpose sampler, the best place to begin is with the HARM algorithm. This is not to say that HARM is the best sampler and everything else pales by comparison. Instead, HARM is a great sampler with which to start in the general case, and for beginners. HARM does not have any specifications (though optionally it can be adaptive or block updated), making it one of the easiest samplers to use. Non-adaptive HARM does not need to be followed up with a non-adaptive algorithm, pre-adaptive burn-in is not required, and diminishing adaptation is not a concern. HARM does not have to learn a proposal covariance matrix. Other nice properties are that (un-blocked) HARM is the fastest algorithm per iteration, it performs well when started far from the target, HARM does well with correlated parameters, and it has good potential with multimodal parameters.

In models with small dimensions, say less than twenty, most MCMC samplers work well. The current, general recommendation is RAM, since it is multivariate and does not require derivatives.

In models with large dimensions, from hundreds to thousands, blockwise sampling is highly recommended, such as with AMM or HARM (non-adaptive with small blocks, adaptive with larger blocks). The disadvantage of AMM is that it must learn the proposal covariance matrix. Un-blocked multivariate proposals have difficulty in large dimensions. Componentwise

algorithms such as ADMG, AGG, AMWG, CHARM, GG, MWG, RDMH, and Slice often have faster improvement and convergence in iterations, though the time per iteration becomes unacceptable in large dimensions. Like the componentwise algortihms, algorithms that require derivatives, such as the HMC family, take an unacceptably long time per iteration in large dimensions.

In models with highly-correlated parameters, algorithms with multivariate proposals such as AMM or RAM are probably best, though HARM and t-walk also perform well. Componentwise algorithms such as the AGG, CHARM, GG, the MWG family (with the exception of ADMG), RDMH, and Slice do not explicitly take correlated parameters into account. In models with small dimensions, as above, RAM is recommended. In models with large dimensions, blockwise sampling is highly recommended, such as with AMM or HARM.

State-space models (SSMs), or dynamic linear models (DLMs), are a special consideration. The **LaplacesDemon** package has algorithms in the MWG family specifically for SSMs, such as SAMWG, SMWG, USAMWG, and USMWG. Recommendations vary with model dimension and the correlation of parameters. If correlation is not problematic, then SAMWG is recommended. If correlation is problematic, then AMM is recommended for models with small dimensions, and block updating with AMM or adaptive HARM is recommended for models with large dimensions.

Models with multimodal marginal posterior distributions are potentially troublesome for any numerical approximation algorithm, though MCMC may be better suited in general. It is best to begin either with MCMCMC or RDMH. Alternatives include AGG, CHARM, GG, HARM, RAM, Slice, THMC, or t-walk. The MCMCMC and RDMH algorithms have demonstrated remarkable performance with multimodal distributions. The use of parallel chains in MCMCMC increases the chances that different chains may settle on different modes. Parallel chains from other parallelized algorithms may be helpful in finding multiple modes, but when the chains are combined with the `Combine` function for inference, each mode probably is not represented in a proportion correct for the distribution. Consider updating the model with PMC, with multiple mixture components, after MCMC is finished. Unlike MCMC with parallel chains, the proportion of each mode will be correctly represented with PMC.

Models with discrete parameters currently require either the AGG or GG algorithm, or converting the discrete parameters to continuous parameters so that any MCMC algorithm may be used. This is performed via the continuous relaxation of a Markov random field (MRF), as in Zhang, Ghahramani, Storkey, and Sutton (2012). For more information, see the `dcrmrf` and `rcrmrf` functions.

Models with big data sets, too big for memory, may use the SGLD algorithm, the `BigData` function, or opt for alternative methods suggested in the details of the documentation for the `BigData` function.

Regardless of the model or algorithm, parallel chains are recommended in general, provided the user has multiple CPUs and enough random-access memory (RAM). However, it is best to begin with a single chain, until the user is confident in the model specification. Parallel chains produce more posterior samples upon convergence than single chains in roughly the same amount of time, and may facilitate the discovery of multimodal marginal posterior distributions that would otherwise have been overlooked. Although algorithms may update independently in parallel, there are several that learn from other parallel updates, such as AIES and INCA.

The `Demonic Suggestion` section of output from the `Consort` function also attempts to help the user to select a sampler. There are exceptions to each of these suggestions above. In some cases, a particular algorithm will fail to update for a given example. Hopefully this section assists the user in selecting a sampler.

Afterward

Once the model is updated with the `LaplacesDemon` function, the `BMK.Diagnostic` function is applied to 10 batches of the thinned samples to assess stationarity (or lack of trend). When all parameters are estimated as stationary beyond a given iteration, the previous iterations are suggested to be considered as burn-in and discarded.

The importance of Monte Carlo Standard Error (MCSE) is debated (Gelman *et al.* 2004; Jones, Haran, Caffo, and Neath 2006). It is included in posterior summaries of `LaplacesDemon`, and is one of five main criteria as a stopping rule to appease Laplace's Demon. MCSE has been shown to be a better stopping rule than MCMC diagnostics (Jones *et al.* 2006). **LaplacesDemon** provides a `MCSE` function that allows three methods of estimation: sample variance, batch means (Jones *et al.* 2006), and Geyer's method (Geyer 1992).

The user is encouraged to explore MCMC diagnostics (also called convergence diagnostics). The **LaplacesDemon** package offers `AcceptanceRate`, the `BMK.Diagnostic`, a Cumulative Sample Function (`CSF`), Effective Sample Size (`ESS`), `Gelfand.Diagnostic`, `Gelman.Diagnostic`, `Geweke.Diagnostic`, `Heidelberger.Diagnostic`, Integrated Autocorrelation Time (`IAT`), the Kolmogorov-Smirnov test (`KS.Diagnostic`), Monte Carlo Standard Error (`MCSE`), `Raftery.Diagnostic`, and both the `plot` and `PosteriorChecks` functions include multiple diagnostics.

12.6. Variational Bayes

Variational Bayes (VB) is a family of numerical approximation algorithms that is a subset of variational inference algorithms, or variational methods. Some examples of variational methods include the mean-field approximation, loopy belief propagation, tree-reweighted belief propagation, and expectation propagation (EP). Variational inference for probabilistic models was introduced in the field of machine learning, influenced by statistical physics literature.

A VB algorithm deterministically estimates the marginal posterior distributions (target distributions) in a Bayesian model with approximated distributions by minimizing the Kullback-Leibler Divergence (`KLD`) between the target and its approximation. The complicated posterior distribution is approximated with a simpler distribution. The simpler, approximated distribution is called the variational approximation, or approximation distribution, of the posterior. The term variational is derived from the calculus of variations, and regards optimization algorithms that select the best function (which is a distribution in VB), rather than merely selecting the best parameters.

VB algorithms often use Gaussian distributions as approximating distributions. In this case, both the mean and variance of the parameters are estimated.

Usually, a VB algorithm is slower to convergence than a Laplace Approximation algorithm, and faster to convergence than a Monte Carlo algorithm such as Markov chain Monte Carlo (MCMC). VB often provides solutions with comparable accuracy to MCMC in less time. Though Monte Carlo algorithms provide a numerical approximation to the exact posterior using a set of samples, VB provides a locally-optimal, exact analytical solution to an ap-

proximation of the posterior. VB is often more applicable than MCMC to big data or large-dimensional models.

Since VB is deterministic, it is asymptotic and subject to the same limitations with respect to sample size as Laplace Approximation. However, VB estimates more parameters than Laplace Approximation, such as when Laplace Approximation optimizes the posterior mode of a Gaussian distribution, while VB optimizes both the Gaussian mean and variance.

Traditionally, VB algorithms required customized equations. The `VariationalBayes` function uses general-purpose algorithms. A general-purpose VB algorithm is less efficient than an algorithm custom designed for the model form. However, a general-purpose algorithm is applied consistently and easily to numerous model forms.

Salimans2

The `Salimans2` algorithm is the second algorithm of Salimans and Knowles (2013) is used. This requires the gradient and Hessian, which is more efficient with a small number of parameters as long as the posterior is twice differentiable. The step size is constant. This algorithm is suitable for marginal posterior distributions that are Gaussian and unimodal. A stochastic approximation algorithm is used in the context of fixed-form VB, inspired by considering fixed-form VB to be equivalent to performing a linear regression with the sufficient statistics of the approximation as independent variables and the unnormalized logarithm of the joint posterior density as the dependent variable. The number of requested iterations should be large, since the step-size decreases for larger requested iterations, and a small step-size will eventually converge. A large number of requested iterations results in a smaller step-size and better convergence properties, so hope for early convergence. However convergence is checked only in the last half of the iterations after the algorithm begins to average the mean and variance from the samples of the stochastic approximation. The history of stochastic samples is returned.

13. Software Comparisons

To the best of my knowledge, no other software currently provides a complete Bayesian environment. However, there is now a wide variety of software to perform MCMC for Bayesian inference. Perhaps the most common is BUGS (Gilks, Thomas, and Spiegelhalter 1994), which is an acronym for Bayesian Using Gibbs Sampling (Lunn, Spiegelhalter, Thomas, and Best 2009). BUGS has several versions. A popular variant is JAGS, which is an acronym for Just Another Gibbs Sampler (Plummer 2003). Stan is a recent addition (Stan Development Team 2012). The only other comparisons made here are with some R packages (**AMCMC**, **mcmc**, **MCMCpack**), and SAS. Many other R packages use MCMC, but are not intended as general-purpose MCMC software. Hopefully, there are not any general-purpose MCMC packages in R have been overlooked here.

WinBUGS has been the most common version of BUGS, though it is no longer developed. BUGS is an intelligent MCMC engine that is capable of numerous MCMC algorithms, but prefers Gibbs sampling. According to its user manual (Spiegelhalter *et al.* 2003), WinBUGS 1.4 uses Gibbs sampling with full conditionals that are continuous, conjugate, and standard. For full conditionals that are log-concave and non-standard, derivative-free Adaptive Rejection Sampling (ARS) is used. Slice sampling is selected for non-log-concave densities on a restricted

range, and tunes itself adaptively for 500 iterations. Seemingly as a last resort, an adaptive MCMC algorithm is used for non-conjugate, continuous, full conditionals with an unrestricted range. The standard deviation of the Gaussian proposal distribution is tuned over the first 4,000 iterations to obtain an acceptance rate between 20% and 40%. Samples from the tuning phases of both Slice sampling and adaptive MCMC are ignored in the calculation of all summary statistics, although they appear in trace-plots.

The current version of BUGS, OpenBUGS, allows the user to specify an MCMC algorithm from a long list for each parameter (Lunn *et al.* 2009). This is a step forward, overcoming what is perceived here as an over-reliance on Gibbs sampling[14]. However, if the user does not customize the selection of the MCMC sampler, then Gibbs sampling will be selected for full conditionals that are continuous, conjugate, and standard, just as with WinBUGS.

Based on years of almost daily experience with WinBUGS and JAGS, which are excellent software packages for Bayesian inference, Gibbs sampling is selected too often in these automatic, MCMC engines. An advantage of Gibbs sampling is that the proposals are accepted with probability 1, so convergence "may" be faster (or it may not, when considering algorithmic efficiency, such as in the `Juxtapose` function), whereas the RWM algorithm backtracks due to its random-walk behavior. Unfortunately, pure Gibbs sampling is not as generalizable, because it can function only when certain conjugate distributional forms are known *a priori* (Gilks and Roberts 1996).

The BUGS and JAGS families of MCMC software are excellent. BUGS is capable of several things that `LaplacesDemon` is not. BUGS allows the user to specify the model graphically as a directed acyclic graph (DAG) in Doodle BUGS. Many journal articles and textbooks in several fields have been published that use BUGS, and many include example code[15].

Advantages of the **LaplacesDemon** package over JAGS and WinBUGS (not much experience with OpenBUGS) are: Bayes factors, the Bayesian Bootstrap, big data, comparison of algorithmic inefficiency, confidence in results (correlations are less problematic than in Gibbs), disjoint HPD intervals, elicitation, enivornment is part of R for data manipulation and posterior analysis, examples in documentation are more plentiful, faster with large data sets (when model specification avoids loops), Importance (Variable and Parameter), iterative quadrature, Juxtapose to compare MCMC samplers, Laplace Approximation, likelihood-free estimation, log-posterior is available, LPL intervals, marginal likelihood calculated automatically, modes (functions for multimodality), model specification gives the user complete control on how everything is calculated (including the log-likelihood, posterior, etc., and "tricks" do not have to be used), more MCMC algorithms, parallelization, Population Monte Carlo (PMC), posterior predictive checks and discrepancy statistics, `predict` function for posterior predictive checks or scoring new data sets, RAM (random-access memory) estimation, sensitivity analysis, suggested code is provided at the end of each run, trap errors do not exist or occur, validation of holdouts with BPIC, variational Bayes, and weights can be applied easily (such as weighting records in the likelihood).

The MCMC algorithms in **LaplacesDemon** are generalizable, and generally robust to correlation between variables or parameters. With larger data sets, there is no comparison: **LaplacesDemon** will deliver a converged model long before BUGS or JAGS. When correla-

[14]To quote Geyer (2011), "many naive users still have a preference for Gibbs updates that is entirely unwarranted. If I had a nickel for every time someone had asked for help with slowly converging MCMC and the answer had been to stop using Gibbs, I would be rich".

[15]The first published journal article to use **LaplacesDemon** is Gallo, Miller, and Fender (2012).

tions are high, almost any algorithm in **LaplacesDemon** will perform much better than pure Gibbs sampling.

Stan (Stan Development Team 2012) emphasizes the No-U-Turn Sampler (NUTS), is programmed in C++, and is a promising addition to Bayesian software. The model specification function in Stan loops through records, as with BUGS and JAGS. This allows a fast performance on smaller data sets, and larger data sets are very time-consuming. Stan currently permits some vectorization in the model function.

At the time this article was written, the **AMCMC** package in R is unavailable on CRAN, but may be downloaded from the author's website[16]. This download is best suited for a Linux, Mac, or UNIX operating system, because it requires the `gcc` C compiler, which is unavailable in Windows. It performs adaptive Metropolis-within-Gibbs (Roberts and Rosenthal 2009; Rosenthal 2007), and uses C language, which results in significantly faster sampling, but only when the model specification function is also programmed in C. This algorithm is included in `LaplacesDemon`, where it is referred to as AMWG, for Adaptive Metropolis-within-Gibbs. The algorithm is excellent, except it is associated with long run-times per iteration for large and complex models.

Also in R, the **mcmc** package (Geyer 2013) offers RWM with multivariate Gaussian proposals and allows batching, as well as a simulated tempering algorithm, but it does not have any adaptive algorithms.

The **MCMCpack** package (Martin, Quinn, and Park 2013) in R takes a canned-function approach to MCMC, which is convenient if the user needs the specific form provided, but is otherwise not generalizable. Each canned function has a MCMC algorithm that is specialized to it, though details seem not to be documented, so the user does not know exactly how the model is updated. General-purpose RWM is included, but adaptive algorithms are not. It also offers the option of Laplace Approximation to optimize initial values.

In SAS 9.2 (SAS Institute Inc. 2008), an experimental procedure called `PROC MCMC` has been introduced. It is undeniably a rip-off of BUGS (including its syntax), though OpenBUGS is much more powerful, tested, and generalizable. Since SAS is proprietary, the user cannot see or manipulate the source code, and should expect much more from it than OpenBUGS or any open-source software, given the preposterous price.

14. Large Data Sets and Speed

An advantage of **LaplacesDemon** compared to other MCMC software is that the model is specified in a way that takes advantage of R's vectorization. BUGS, JAGS, and Stan (somewhat), for example, require models to be specified so that each record of data is processed one by one inside a '`for` loop', which significantly slows updating with larger data sets. In contrast, **LaplacesDemon** avoids '`for` loops' and `apply` functions wherever possible[17]. For example, a data set of 100,000 rows and 16 columns (the dependent variable, a column vector of 1's for the intercept, and 14 predictors) was updated 1,000 times with the HARM algorithm. This took 0.15 minutes with `LaplacesDemon`, according to a simple, linear regression[18]. It was nowhere near convergence, but updating the same model with the same data for 1,000

[16] **AMCMC** is available from J. S. Rosenthal's website at http://www.probability.ca/amcmc/

[17] However, when 'for loops' or `apply` functions must be used, **LaplacesDemon** is typically slower than BUGS.

[18] These updates were performed on a 2014 laptop with 64-bit Xubuntu Linux and 8GB RAM.

iterations took 6.81 minutes in JAGS 3.4.0.

However, the speed with which an iteration is estimated is not a good, overall criterion of performance. For applied purposes, the best performance is measured in MCMC algorithmic inefficiency with the `Juxtapose` function, using integrated autocorrelation time (`IAT`). For this comparison, however, this would require updating both models to convergence, and so run-time is reported instead.

For example, a pure Gibbs sampling algorithm with uncorrelated target distributions should converge in far fewer iterations than an algorithm based on random-walk behavior, such as many (but not all) algorithms in `LaplacesDemon`. Depending on circumstances, `LaplacesDemon` should handle larger data sets better, and it may estimate each iteration faster, but it may also take more iterations to converge[19].

A lower-level language can be much faster for MCMC, but only when the model specification function is vectorized, which is currently not the case, citing examples such as BUGS, JAGS, SAS, and Stan (somewhat). That style of software is fast only with small sample sizes. Computationally, the future of MCMC algorithms should be in vectorizing model specifications in lower-level languages. And here's the trick: software developers must make it feasible for an ordinary user to specify a model with vast flexibility when unfamiliar with the lower-level language. Until that day arrives, **LaplacesDemon** currently leads the way in general-purpose Bayesian inference for users not specialized in vectorization with lower-level languages.

15. Bayesian-Inference.com

Many additional resources may be found at `http://www.bayesian-inference.com`, as well as other places online:

- A Bayesian forum is available at `http://www.bayesian-inference.com/forum` to discuss all things Bayesian, including **LaplacesDemon**.
- Bayesian information is being compiled under `http://www.bayesian-inference.com/bayesian`.
- Bayesian news is aggregated daily as "The Bayesian Bulletin": `http://www.bayesian-inference.com/newsbayesian`.
- Consulting services are available here: `http://www.bayesian-inference.com/consulting`.
- Merchandise may be found at `http://www.bayesian-inference.com/merchandise`, such as **LaplacesDemon** t-shirts, coffee mugs, and more.
- **LaplacesDemon** development is public and occurs at `https://github.com/Statisticat/LaplacesDemon`.

[19] To continue this example, JAGS may be *guessed* to take 20,000 iterations or 2.25 hours, and `LaplacesDemon` may take 400,000 iterations or 1 hour, and also have less autocorrelation in the chains due to more thinning. One of the slower adaptive algorithms in `LaplacesDemon` is AMWG, which updated in 2.3 minutes, and should finish around 50,000 iterations or 46 minutes. As sample size increases and when `for` loops are controlled, `LaplacesDemon` doesn't just outperform, but embarrasses the looping-style model specification approach of BUGS and its derivatives to the point of absurdity. Increase data size to a million records, and `LaplacesDemon` completes 1,000 iterations in 1.4 minutes, while JAGS is estimated to take over 5 days! This is what is meant by absurdity. So much for `C` or lower-level languages in that style of programming model specification functions!

- **LaplacesDemon** screencasts are available at http://www.bayesian-inference.com/softwarescreencasts.
- **LaplacesDemon** updates are announced at https://plus.google.com/+Bayesian-inference.
- Opinion polls for Bayesian inference and **LaplacesDemon** are here: http://www.bayesian-inference.com/polls.
- Technical support services are available at http://www.bayesian-inference.com/support.
- And, the home of **LaplacesDemon** is http://www.bayesian-inference.com/software.

16. Conclusion

The **LaplacesDemon** package is a significant contribution toward Bayesian inference in R. In turn, contributions toward the development of **LaplacesDemon** are welcome. Please visit https://github.com/Statisticat/LaplacesDemon to contribute to development or report software bugs by opening an issue, or send an email to software@bayesian-inference.com with constructive criticism.

References

Ando T (2007). "Bayesian Predictive Information Criterion for the Evaluation of Hierarchical Bayesian and Empirical Bayes Models." *Biometrika*, **94**(2), 443–458.

Atchade Y (2006). "An Adaptive Version for the Metropolis Adjusted Langevin Algorithm with a Truncated Drift." *Methodology and Computing in Applied Probability*, **8**, 235–254.

Atchade Y, Roberts G, Rosenthal J (2011). "Towards Optimal Scaling of Metropolis-Coupled Markov Chain Monte Carlo." *Statistics and Computing*, **21**(4), 555–568.

Azevedo-Filho A, Shachter R (1994). "Laplace's Method Approximations for Probabilistic Inference in Belief Networks with Continuous Variables." In R Mantaras, D Poole (eds.), *Uncertainty in Artificial Intelligence*, pp. 28–36. Morgan Kauffman, San Francisco, CA.

Bai Y (2009). "An Adaptive Directional Metropolis-within-Gibbs Algorithm." Technical Report in Department of Statistics at the University of Toronto.

Bayes T, Price R (1763). "An Essay Towards Solving a Problem in the Doctrine of Chances. By the late Rev. Mr. Bayes, communicated by Mr. Price, in a letter to John Canton, MA. and F.R.S." *Philosophical Transactions of the Royal Society of London*, **53**, 370–418.

Bernardo J, Smith A (2000). *Bayesian Theory*. John Wiley & Sons, West Sussex, England.

Berndt E, Hall B, Hall R, Hausman J (1974). "Estimation and Inference in Nonlinear Structural Models." *Annals of Economic and Social Measurement*, **3**, 653–665.

Besag J, Green P (1993). "Spatial Statistics and Bayesian Computation." *Journal of the Royal Statistical Society*, **B 55**, 25–37.

Boyles L, Welling M (2012). "Refractive Sampling." http://www.ics.uci.edu/~lboyles/publications.html.

Broyden C (1970). "The Convergence of a Class of Double Rank Minimization Algorithms: 2. The New Algorithm." *Journal of the Institute of Mathematics and its Applications*, **6**, 76–90.

Cappe O, Douc R, Guillin A, Marin J, Robert C (2008). "Adaptive Importance Sampling in General Mixture Classes." *Statistics and Computing*, **18**, 587–600.

Cappe O, Guillin A, Marin J, Robert C (2004). "Population Monte Carlo." *Journal of Computational and Graphical Statistics*, **13**, 907–929.

Chen M, Schmeiser B (1992). "Performance of the Gibbs, Hit-And-Run and Metropolis Samplers." *Journal of Computational and Graphical Statistics*, **2**, 251–272.

Christen J, Fox C (2010). "A General Purpose Sampling Algorithm for Continuous Distributions (the t-walk)." *Bayesian Analysis*, **5(2)**, 263–282.

Craiu R, Rosenthal J, Yang C (2009). "Learn From Thy Neighbor: Parallel-Chain and Regional Adaptive MCMC." *Journal of the American Statistical Association*, **104**(488), 1454–1466.

Crawley M (2007). *The R Book*. John Wiley & Sons Ltd, West Sussex, England.

Duane S, Kennedy AD, Pendleton BJ, Roweth D (1987). "Hybrid Monte Carlo." *Physics Letters*, **B**(195), 216–222.

Dutta S (2012). "Multiplicative Random Walk Metropolis-Hastings on the Real Line." *Sankhya B*, **74**(2), 315–342.

Earl D, Deem M (2005). "Parallel Tempering: Theory, Applications, and New Perspectives." *Journal of Chemistry Chemical Physics*, **7**, 3910–3916.

Fearnhead P (2011). "MCMC for State-Space Models." In S Brooks, A Gelman, G Jones, M Xiao-Li (eds.), *Handbook of Markov Chain Monte Carlo*, pp. 513–530. Chapman & Hall, Boca Raton, FL.

Foreman-Mackey D, Hogg D, Lang D, Goodman J (2012). "emcee: The MCMC Hammer." Upcoming in Publications of the Astronomical Society of the Pacific, http://arxiv.org/abs/1202.3665.

Frantz D, Freeman J, Doll J (1990). "Reducing Quasi-Ergodic Behavior in Monte Carlo Simulations by J-Walking: Applications to Atomic Clusters." *Journal of Chemical Physics*, **93**, 2769–2784.

Gallo E, Miller B, Fender R (2012). "Assessing luminosity correlations via cluster analysis: Evidence for dual tracks in the radio/X-ray domain of black hole X-ray binaries." *Monthly Notices of the Royal Astronomical Society*, **423**, 590–599.

Garthwaite P, Fan Y, Sisson S (2010). "Adaptive Optimal Scaling of Metropolis-Hastings Algorithms Using the Robbins-Monro Process."

Gelfand A (1996). "Model Determination Using Sampling Based Methods." In W Gilks, S Richardson, D Spiegelhalter (eds.), *Markov Chain Monte Carlo in Practice*, pp. 145–161. Chapman & Hall, Boca Raton, FL.

Gelman A (2008). "Scaling Regression Inputs by Dividing by Two Standard Deviations." *Statistics in Medicine*, **27**, 2865–2873.

Gelman A, Carlin J, Stern H, Rubin D (2004). *Bayesian Data Analysis*. 2nd edition. Chapman & Hall, Boca Raton, FL.

Gelman A, Meng X, Stern H (1996a). "Posterior Predictive Assessment of Model Fitness via Realized Discrepancies." *Statistica Sinica*, **6**, 773–807.

Gelman A, Roberts G, Gilks W (1996b). "Efficient Metropolis Jumping Rules." *Bayesian Statistics*, **5**, 599–608.

Gelman A, Rubin D (1992). "Inference from Iterative Simulation Using Multiple Sequences." *Statistical Science*, **7**(4), 457–472.

Geman S, Geman D (1984). "Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images." *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **6**(6), 721–741.

Geweke J, Tanizaki H (2001). "Bayesian Estimation of State-Space Models Using the Metropolis-Hastings Algorithm within Gibbs Sampling." *Computational Statistics and Data Analysis*, **37**, 151–170.

Geyer C (1991). "Markov Chain Monte Carlo Maximum Likelihood." In E Keramidas (ed.), *Computing Science and Statistics: Proceedings of the 23rd Symposium of the Interface*, pp. 156–163. Fairfax Station VA: Interface Foundation.

Geyer C (1992). "Practical Markov Chain Monte Carlo (with Discussion)." *Statistical Science*, **7**(4), 473–511.

Geyer C (2011). "Introduction to Markov Chain Monte Carlo." In S Brooks, A Gelman, G Jones, M Xiao-Li (eds.), *Handbook of Markov Chain Monte Carlo*, pp. 3–48. Chapman & Hall, Boca Raton, FL.

Geyer C (2013). ***mcmc**: Markov Chain Monte Carlo*. R package version 0.9-2, URL http://cran.r-project.org/web/packages/mcmc/index.html.

Gilks W, Roberts G (1996). "Strategies for Improving MCMC." In W Gilks, S Richardson, D Spiegelhalter (eds.), *Markov Chain Monte Carlo in Practice*, pp. 89–114. Chapman & Hall, Boca Raton, FL.

Gilks W, Thomas A, Spiegelhalter D (1994). "A Language and Program for Complex Bayesian Modelling." *The Statistician*, **43**(1), 169–177.

Goldfarb D (1970). "A Family of Variable Metric Methods Derived by Variational Means." *Mathematics of Computation*, **24**(109), 23–26.

Goodman J, Weare J (2010). "Ensemble Samplers with Affine Invariance." *Communications in Applied Mathematics and Computational Science*, **5**(1), 65–80.

Green P (1995). "Reversible Jump Markov Chain Monte Carlo Computation and Bayesian Model Determination." *Biometrika*, **82**, 711–732.

Haario H, Laine M, Mira A, Saksman E (2006). "DRAM: Efficient Adaptive MCMC." *Statistical Computing*, **16**, 339–354.

Haario H, Saksman E, Tamminen J (2001). "An Adaptive Metropolis Algorithm." *Bernoulli*, **7**(2), 223–242.

Hastings W (1970). "Monte Carlo Sampling Methods Using Markov Chains and Their Applications." *Biometrika*, **57**(1), 97–109.

Hestenes M, Stiefel E (1952). "Methods of Conjugate Gradients for Solving Linear Systems." *Journal of Research of the National Bureau of Standards*, **49**(6), 409–436.

Hoffman M, Gelman A (2012). "The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo." *Journal of Machine Learning Research*, pp. 1–30.

Hooke R, Jeeves T (1961). "'Direct Search' Solution of Numerical and Statistical Problems." *Journal of the Association for Computing Machinery*, **8**(3), 212–229.

Irony T, Singpurwalla N (1997). "Noninformative Priors Do Not Exist: a Discussion with Jose M. Bernardo." *Journal of Statistical Inference and Planning*, **65**, 159–189.

Jones G, Haran M, Caffo B, Neath R (2006). "Fixed-Width Output Analysis for Markov chain Monte Carlo." *Journal of the American Statistical Association*, **100**(1), 1537–1547.

Laplace P (1774). "Memoire sur la Probabilite des Causes par les Evenements." *l'Academie Royale des Sciences*, **6**, 621–656. English translation by S.M. Stigler in 1986 as "Memoir on the Probability of the Causes of Events" in *Statistical Science*, **1**(3), 359–378.

Laplace P (1812). *Theorie Analytique des Probabilites*. Courcier, Paris. Reprinted as "Oeuvres Completes de Laplace", **7**, 1878–1912. Paris: Gauthier-Villars.

Laplace P (1814). "Essai Philosophique sur les Probabilites." English translation in Truscott, F.W. and Emory, F.L. (2007) from (1902) as "A Philosophical Essay on Probabilities". ISBN 1602063281, translated from the French 6th ed. (1840).

Laud P, Ibrahim J (1995). "Predictive Model Selection." *Journal of the Royal Statistical Society*, **B 57**, 247–262.

Liu J, Liang F, Wong W (2000). "The Multiple-Try Method and Local Optimization in Metropolis Sampling." *Journal of the American Statistical Association*, **95**, 121–134.

Lunn D, Spiegelhalter D, Thomas A, Best N (2009). "The BUGS Project: Evolution, Critique, and Future Directions." *Statistics in Medicine*, **28**, 3049–3067.

Martin A, Quinn K, Park J (2013). ***MCMCpack**: Markov chain Monte Carlo (MCMC) Package*. R package version 1.3-3, URL http://cran.r-project.org/web/packages/MCMCpack/index.html.

Metropolis N, Rosenbluth A, MN R, Teller E (1953). "Equation of State Calculations by Fast Computing Machines." *Journal of Chemical Physics*, **21**, 1087–1092.

Mira A (2001). "On Metropolis-Hastings Algorithms with Delayed Rejection." *Metron*, **LIX**(3–4), 231–241.

Murray I, Adams R, KacKay D (2010). "Elliptical Slice Sampling." *Journal of Machine Learning Research*, **9**, 541–548.

Naylor J, Smith A (1982). "Applications of a Method for the Efficient Computation of Posterior Distributions." *Applied Statistics*, **31**(3), 214–225.

Neal R (1997). "Markov Chain Monte Carlo Methods Based on Slicing the Density Function." Technical Report, University of Toronto.

Neal R (2003). "Slice Sampling (with Discussion)." *Annals of Statistics*, **31**(3), 705–767.

Neal R (2011). "MCMC for Using Hamiltonian Dynamics." In S Brooks, A Gelman, G Jones, M Xiao-Li (eds.), *Handbook of Markov Chain Monte Carlo*, pp. 113–162. Chapman & Hall, Boca Raton, FL.

Nelder J, Mead R (1965). "A Simplex Method for Function Minimization." *The Computer Journal*, **7**(4), 308–313.

Nocedal J, Wright S (1999). *Numerical Optimization*. Springer-Verlag, New York, New York.

O'Hagan A (1987). "Monte Carlo is Fundamentally Unsound." *The Statistician*, **31**(3), 214–225.

O'Hagan A (1991). "Bayes-Hermite Quadrature." *Statistical Planning and Inference*, **29**, 245–260.

O'Hagan A (1992). "Some Bayesian Numerical Analysis." In J Bernardo, J Berger, A David, A Smith (eds.), *Bayesian Statistics 4*, pp. 356–363. Oxford University Press, England.

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

R Core Team (2013). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL http://www.R-project.org.

Rasmussen C, Ghahramani Z (2003). "Bayesian Monte Carlo." In S Becker, K Obermayer (eds.), *Advances in Neural Information Processing Systems*. MIT Press, Cambridge, MA.

Riedmiller M (1994). "Advanced Supervised Learning in Multi-Layer Perceptrons - From Backpropagation to Adaptive Learning Algorithms." *Computer Standards and Interfaces*, **16**, 265–278.

Ritter C, Tanner M (1992). "Facilitating the Gibbs Sampler: the Gibbs Stopper and the Griddy-Gibbs Sampler." *Journal of the American Statistical Association*, **87**, 861–868.

Robert C (2007). *The Bayesian Choice*. 2nd edition. Springer, Paris, France.

Roberts G, Rosenthal J (2001). "Optimal Scaling for Various Metropolis-Hastings Algorithms." *Statistical Science*, **16**, 351–367.

Roberts G, Rosenthal J (2007). "Coupling and Ergodicity of Adaptive Markov Chain Monte Carlo Algorithms." *Journal of Applied Probability*, **44**, 458–475.

Roberts G, Rosenthal J (2009). "Examples of Adaptive MCMC." *Computational Statistics and Data Analysis*, **18**, 349–367.

Roberts G, Tweedie R (1996). "Exponential Convergence of Langevin Distributions and Their Discrete Approximations." *Bernoulli*, **2**(4), 341–363.

Rosenthal J (2007). "AMCMC: An R Interface for Adaptive MCMC." *Computational Statistics and Data Analysis*, **51**, 5467–5470.

Rossky P, Doll J, Friedman H (1978). "Brownian Dynamics as Smart Monte Carlo Discrete Approximations." *Journal of Chemical Physics*, **69**, 4628–4633.

Salimans T, Knowles D (2013). "Fixed-Form Variational Posterior Approximation through Stochastic Linear Regression." *Bayesian Analysis*, **8**(4), 837–882.

SAS Institute Inc (2008). *SAS/STAT 9.2 User's Guide*. Cary, NC: SAS Institute Inc.

Shaby B, Wells M (2010). "Exploring an Adaptive Metropolis Algorithm." Working Paper in Department of Statistical Science, Duke University.

Shanno D (1970). "Conditioning of quasi-Newton Methods for Function Minimization." *Mathematics of Computation*, **24**, 647–650.

Smith R (1984). "Efficient Monte Carlo Procedures for Generating Points Uniformly Distributed Over Bounded Region." *Operations Research*, **32**, 1296–1308.

Solonen A, Ollinaho P, Laine M, Haario H, Tamminen J, Jarvinen H (2012). "Efficient MCMC for Climate Model Parameter Estimation: Parallel Adaptive Chains and Early Rejection." *Bayesian Analysis*, **7**(2), 1–22.

Spiegelhalter D, Thomas A, Best N, Lunn D (2003). *WinBUGS User Manual, Version 1.4*. MRC Biostatistics Unit, Institute of Public Health and Department of Epidemiology and Public Health, Imperial College School of Medicine, UK.

Stan Development Team (2012). "Stan: A C++ Library for Probability and Sampling." http://mc-stan.org.

Statisticat LLC (2014). ***LaplacesDemon**: Complete Environment for Bayesian Inference*. R package version 14.05.07, URL http://www.bayesian-inference.com/software.

Ter Braak C (2006). "A Markov Chain Monte Carlo Version of the Genetic Algorithm Differential Evolution: Easy Bayesian Computing for Real Parameter Spaces." *Statistics and Computing*, **16**, 239–249.

Ter Braak C, Vrugt J (2008). "Differential Evolution Markov Chain with Snooker Updater and Fewer Chains." *Statistics and Computing*, **18**(4), 435–446.

Thompson M (2011). "Slice Sampling with Multivariate Steps." http://hdl.handle.net/1807/31955.

Tierney L (1994). "Markov Chains for Exploring Posterior Distributions." *The Annals of Statistics*, **22**(4), 1701–1762. With discussion and a rejoinder by the author.

Tierney L, Kadane J (1986). "Accurate Approximations for Posterior Moments and Marginal Densities." *Journal of the American Statistical Association*, **81**(393), 82–86.

Tierney L, Kass R, Kadane J (1989). "Fully Exponential Laplace Approximations to Expectations and Variances of Nonpositive Functions." *Journal of the American Statistical Association*, **84**(407), 710–716.

Turchin V (1971). "On the Computation of Multidimensional Integrals by the Monte Carlo Method." *Theory of Probablility and its Applications*, **16**(4), 720–724.

Vihola M (2011). "Robust Adaptive Metropolis Algorithm with Coerced Acceptance Rate." In Forthcoming (ed.), *Statistics and Computing*, pp. 1–12. Springer, Netherlands.

Welling M, Teh Y (2011). "Bayesian Learning via Stochastic Gradient Langevin Dynamics." *Proceedings of the 28th International Conference on Machine Learning (ICML)*, pp. 681–688.

Wraith D, Kilbinger M, Benabed K, Cappé O, Cardoso J, Fort G, Prunet S, Robert C (2009). "Estimation of Cosmological Parameters Using Adaptive Importance Sampling." *Physical Review D*, **80**(2), 023507.

Zelinka I (2004). "SOMA - Self Organizing Migrating Algorithm." In G Onwubolu, B Babu (eds.), *New Optimization Techniques in Engineering*. Springer, Berlin, Germany.

Zhang Y, Ghahramani Z, Storkey A, Sutton C (2012). "Continuous Relaxations for Discrete Hamiltonian Monte Carlo." *Advances in Neural Information Processing Systems*, **25**, 3203–3211.

Affiliation:

Statisticat, LLC
Farmington, CT
E-mail: software@bayesian-inference.com
URL: http://www.bayesian-inference.com/software