LaplacesDemon: An R Package for Bayesian Inference

Byron Hall
STATISTICAT, LLC

Abstract

LaplacesDemon, usually referred to as Laplace's Demon, is a contributed R package for Bayesian inference, and is freely available on the Comprehensive R Archive Network (CRAN). Laplace's Demon allows Laplace Approximation and the choice of numerous MCMC algorithms to update a Bayesian model according to a user-specified model function. The user-specified model function enables Bayesian inference for any model form, provided the user specifies, or approximates, the likelihood. Laplace's Demon also attempts to assist the user by creating and offering R code, based on a previous model update, that can be copy/pasted and executed. Posterior predictive checks and many other features are included as well. Laplace's Demon seeks to be generalizable and user-friendly to Bayesians, especially Laplacians.

Keywords:~Adaptive, AM, Bayesian, Delayed Rejection, DR, DRAM, DRM, Gradient Ascent, Hamiltonian, High Performance Computing, HMC, HPC, Laplace Approximation, LaplacesDemon, Laplace's Demon, Markov chain Monte Carlo, MCMC, Metropolis, Metropolis-within-Gibbs, Optimization, Parallel, R, Random Walk, Random-Walk, Resilient Backpropagation, STATISTICAT, t-walk.

Bayesian inference is named after Reverend Thomas Bayes (1702-1761) 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

[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.

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 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 Development Core Team 2012) package called **LaplacesDemon** (Hall 2012), 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 Laplace's Demon. This article guides the user through installation, data, specifying a model, initial values, updating Laplace's Demon, summarizing and plotting output, posterior predictive checks, general suggestions, discusses independence and observability, high performance computing, covers details of the algorithm, software comparisons, and discusses large data sets and speed.

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, Gelman, Carlin, Stern, and Rubin (2004), and Crawley (2007).

1. Installation

To obtain Laplace's Demon, simply open R and install the **LaplacesDemon** package from a CRAN mirror:

```
> install.packages("LaplacesDemon")
```

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

[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.

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

```
> library(LaplacesDemon)
```

2. Data

Laplace's Demon 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 some of the remaining variables. Since an intercept will be included, a vector of 1's is inserted into design matrix $\mathbf{X}$.

```
> data(demonsnacks)
> N <- nrow(demonsnacks)
> y <- log(demonsnacks$Calories)
> X <- cbind(1, as.matrix(demonsnacks[,c(7,8,10)]))
> J <- ncol(X)
> for (j in 2:J) {X[,j] <- CenterScale(X[,j])}
> mon.names <- c("LP","sigma")
> parm.names <- as.parm.names(list(beta=rep(0,J), log.sigma=0))
> MyData <- list(J=J, X=X, mon.names=mon.names, parm.names=parm.names, y=y)
```

There are J=4 independent variables (including the intercept), one for each column in design matrix $\mathbf{X}$. However, there are 5 parameters, since the residual variance, σ^2, must be included as well. The reason why it is called `log.sigma` will be explained later. 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). Laplace's Demon provides a `CenterScale` function to center and scale predictors[4].

Laplace's Demon will consider using Laplace Approximation, and part of this consideration includes determining the sample size. The user must specify the number of observations in the data as either a scalar `n` or `N`. If these are not found by the `LaplaceApproximation` or `LaplacesDemon` functions, then it will attempt to determine sample size as the number of rows in `y` or `Y`.

3. Specifying a Model

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

[3]Though most R functions use data in the form of a data frame, Laplace's Demon 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))`.

[5]Examples of more than 80 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).

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

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[1:Data$J]
+      sigma <- exp(parm[Data$J+1])
+      ### 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))
+      ### Log-Posterior
+      LP <- LL + sum(beta.prior) + sigma.prior
+      Modelout <- list(LP=LP, Dev=-2*LL, Monitor=c(LP,sigma), yhat=mu,
+           parm=parm)
+      return(Modelout)
+      }
```

[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.

Laplace's Demon 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 Laplace's Demon is maximizing the logarithm of the unnormalized joint posterior density, Laplace's Demon passes two arguments 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 to Laplace's Demon 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=c(LP,sigma),
    yhat=mu, 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 Laplace's Demon passes a vector of parameters called `parm` 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`. The reason why `sigma` is exponentiated will, again, be explained later.

```
beta <- parm[1:Data$J]
sigma <- exp(parm[Data$J+1])
```

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)
```

It is important to reparameterize all parameters to be real-valued. For example, a positive-only parameter such as variance should be allowed to range from $-\infty$ to ∞, and be trans-

[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'.

formed in the `Model` function with the `exp` function, which will force it to positive values. A parameter θ that needs to be bounded in the model, such as in the interval [1,5], can be transformed to that range with a logistic function, such as $1+4[\exp(\theta)/(\exp(\theta)+1)]$. Alternatively, each parameter may be constrained in the `Model` function, such as with the `interval` function. Laplace's Demon will attempt to increase or decrease the value of each parameter to maximize `LP`, without consideration for the distributional form of the parameter. In the above example, the residual standard deviation `sigma` receives a half-Cauchy distributed prior of the form:

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

In this specification, `sigma` cannot be negative. By reparameterizing `sigma` as

```
sigma <- exp(parm[Data$J+1])
```

Laplace's Demon will increase or decrease `parm[Data$J+1]`, which is effectively `log(sigma)`. Now it is possible for Laplace's Demon to decrease `log(sigma)` below zero without causing an error or violating its half-Cauchy distributed specification.

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
```

Specifying the model in the `Model` function is the most involved aspect for the user of Laplace's Demon. But it has been designed so it is also incredibly flexible, allowing a wide variety of Bayesian models to be specified. Missing values are also easy to estimate (see the "Examples" vignette).

4. Initial Values

Laplace's Demon requires a vector of initial values for the parameters. Each initial value is a starting point for the estimation of a parameter. When all initial values are set to zero, Laplace's Demon will optimize initial values using a resilient backpropagation 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. Laplace's Demon will not use Laplace Approximation when the sample size is not at least five times the number of parameters. Otherwise, the user may prefer to optimize initial values in the `LaplaceApproximation` function before using the `LaplacesDemon` function. When

Laplace's Demon receives initial values that are not all set to zero, it will begin to update each parameter.

In this example, there are 5 parameters. With no prior knowledge, it is a good idea either to randomize each initial value, such as with the `GIV` function (which stands for "generate initial values"), or set all of them equal to zero and let the `LaplaceApproximation` function optimize the initial values, provided there is sufficient sample size. Here, the `LaplaceApproximation` function will be introduced in the `LaplacesDemon` function, so the first 4 parameters, the `beta` parameters, have been set equal to zero, and the remaining parameter, `log.sigma`, has been set equal to `log(1)`, which is equal to zero. This visually reminds me that I am working with the log of `sigma`, rather than `sigma`, and is merely a personal preference. 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.

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

5. Laplace's Demon

Compared to specifying the model in the `Model` function, the actual use of Laplace's Demon is very easy. Since Laplace's Demon is stochastic, or involves pseudo-random numbers, 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)
```

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=200000, Status=50000, Thinning=200,
+      Algorithm="twalk", Specs=list(SIV=Initial.Values+1, n1=4, at=6, aw=1.5))
```

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. Laplace's Demon offers Laplace Approximation and numerous MCMC algorithms (which are explained in section 12). The above example specifies the t-walk algorithm for updating.

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.

[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'.

- 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, so the algorithm will begin with its own estimate.
- The `Iterations=200000` argument indicates that the `LaplacesDemon` function will update 200,000 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=200` argument indicates that only ever Kth iteration will be retained in the output, and in this case, every 200th 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. In this case, t-walk is abbreviated to `twalk`.
- Finally, the `Specs` argument contains specifications for each algorithm named in the `Algorithm` argument. The `twalk` algorithm accepts four specifications, the first (`SIV`) is recommended to be specified, and the remaining specifications are recommended to remain as defaults: `SIV`, `n1`, `at`, and `aw`. Details on algorithms and specifications are given later.

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

```
> Fit <- LaplacesDemon(Model, Data=MyData, Initial.Values,
+      Covar=NULL, Iterations=200000, Status=50000, Thinning=200,
+      Algorithm="twalk", Specs=list(SIV=Initial.Values+1, n1=4, at=6, aw=1.5))

Laplace's Demon was called on Mon Jul  2 11:11:25 2012

Performing initial checks...

Laplace Approximation will be used on initial values.
Sample Size:  39
Laplace Approximation begins...
Iteration:  10  of  100
Iteration:  20  of  100
Iteration:  30  of  100
Iteration:  40  of  100
Iteration:  50  of  100
Iteration:  60  of  100
Iteration:  70  of  100
Iteration:  80  of  100
Iteration:  90  of  100
Iteration:  100  of  100
Creating Summary from Point-Estimates
Laplace Approximation is finished.
```

```
The covariance matrix from Laplace Approximation has been scaled
for Laplace's Demon, and the posterior modes are now the initial
values for Laplace's Demon.

Algorithm: t-walk

Laplace's Demon is beginning to update...
Iteration: 50000,   Proposal: Multivariate Subset
Iteration: 100000,   Proposal: Multivariate Subset
Iteration: 150000,   Proposal: Multivariate Subset
Iteration: 200000,   Proposal: Multivariate Subset

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

Laplace's Demon has finished.
```

Laplace's Demon 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] 0.250095
```

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 = 2e+05, Status = 50000, Thinning = 200,
    Algorithm = "twalk", Specs = list(SIV = Initial.Values +
        1, n1 = 4, at = 6, aw = 1.5))

Acceptance Rate: 0.2501
Adaptive: 200001
Algorithm: t-walk
Covariance Matrix: (NOT SHOWN HERE; diagonal shown instead)
[1] 2.518045e-16 2.518045e-16 2.518045e-16 2.518045e-16 2.518045e-16

Covariance (Diagonal) History: (NOT SHOWN HERE)
Deviance Information Criterion (DIC):
       All Stationary
Dbar 82.48     82.308
pD    8.46      5.452
DIC  90.94     87.760

Delayed Rejection (DR): 0
Initial Values:
    beta[1]     beta[2]     beta[3]     beta[4]   log.sigma
1.431484579 0.042252862 0.005882033 0.022319770 5.163186831

Iterations: 2e+05
Log(Marginal Likelihood): -42.9888
Minutes of run-time: 0.94
Model: (NOT SHOWN HERE)
Monitor: (NOT SHOWN HERE)
Parameters (Number of): 5
Periodicity: 1
Posterior1: (NOT SHOWN HERE)
Posterior2: (NOT SHOWN HERE)
Recommended Burn-In of Thinned Samples: 101
Recommended Burn-In of Un-thinned Samples: 20200
Recommended Thinning: 600
Status is displayed every 50000 iterations
Summary1: (SHOWN BELOW)
Summary2: (SHOWN BELOW)
Thinned Samples: 1000
Thinning: 200
```

```
Summary of All Samples
                Mean        SD        MCSE       ESS          LB      Median
beta[1]    5.0423935 0.1137634 0.004732943  687.1895   4.8278198   5.0401542
beta[2]    1.4774269 0.2303922 0.007468725 1000.0000   1.0585381   1.4705194
beta[3]    0.5269607 0.2765857 0.012628882  630.4903   0.0299373   0.5155616
beta[4]    0.9874657 0.2587317 0.010370946  754.1476   0.4948897   0.9869138
log.sigma  -0.3621324 0.1266120 0.006815453  502.4538  -0.5889436  -0.3702670
Deviance   82.4797906 4.1135035 0.296982949  335.4500  77.8420893  81.6655942
LP        -62.4169592 2.0569348 0.148516210  335.4334 -66.7580009 -62.0096498
sigma      0.7013199 0.0930383 0.005558376  419.2609   0.5549132   0.6891981
                  UB
beta[1]    5.2683750
beta[2]    1.9579202
beta[3]    1.0519756
beta[4]    1.5001514
log.sigma  -0.1186703
Deviance   91.1640392
LP        -60.0977381
sigma      0.8990128


Summary of Stationary Samples
                Mean         SD        MCSE      ESS           LB      Median
beta[1]    5.0404876 0.10891465 0.004123040 794.1390   4.82816192   5.0367385
beta[2]    1.4772455 0.23168922 0.007583019 900.0000   1.05589391   1.4693777
beta[3]    0.5268889 0.26330708 0.009208656 900.0000   0.02250511   0.5199781
beta[4]    0.9856159 0.25584060 0.009157865 814.2606   0.50567837   0.9822276
log.sigma  -0.3688615 0.11854461 0.004429594 807.3218  -0.58676627  -0.3751553
Deviance   82.3080579 3.30212502 0.124487829 888.3963  77.84236748  81.7031713
LP        -62.3310632 1.65115963 0.148516210 888.4879 -66.46463439 -62.0288283
sigma      0.6958158 0.08328142 0.005558376 809.7844   0.55709619   0.6862527
                  UB
beta[1]    5.2581337
beta[2]    1.9608400
beta[3]    1.0338978
beta[4]    1.5007908
log.sigma  -0.1390347
Deviance   90.5761081
LP        -60.0981061
sigma      0.8748019

Demonic Suggestion

Due to the combination of the following conditions,

1. t-walk
```

```
2. The acceptance rate (0.250095) 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
   101 iterations.

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 2e+05 iterations of 5 variables in 0.94 minutes. Of course this was fast, since there were only 39 records, and the form of the specified model was simple. As discussed later, Laplace's Demon 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 Laplace's Demon 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). The acceptance rate of most algorithms is considered satisfactory if it is within the interval [15%,50%][9], and LMC or MALA must be in the interval [50%, 65%]. 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 are sufficiently large, and all parameters were estimated to be stationary. Since the algorithm was the non-adaptive t-walk (twalk), the Markov property holds, so let's look at some plots.

7. Plotting Output

Laplace's Demon 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, so that we don't see the initial values, it is set to 100.

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.

```
> BurnIn <- 100

> plot(Fit, BurnIn=100, 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.

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:

[9]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 must be in the interval [60%, 70%]

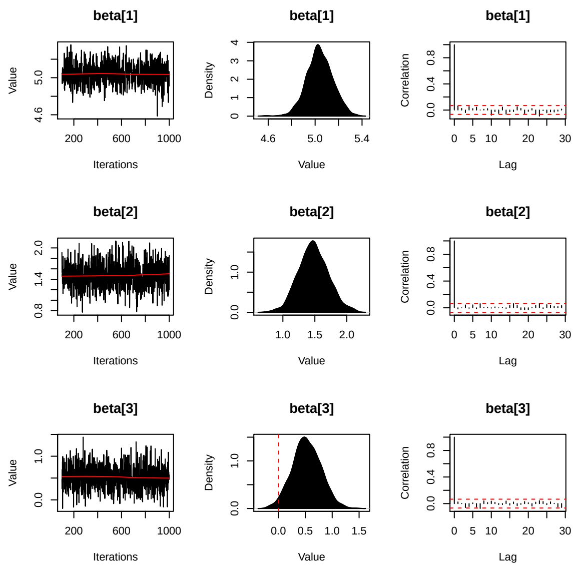

Figure 1: Plots of Marginal Posterior Samples

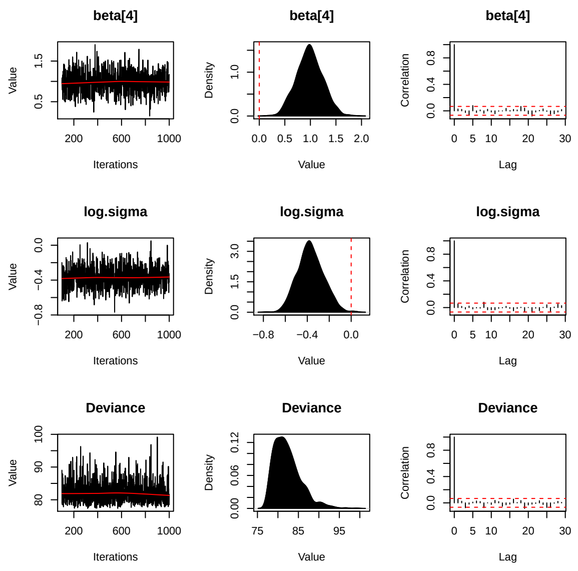

Figure 2: Plots of Marginal Posterior Samples

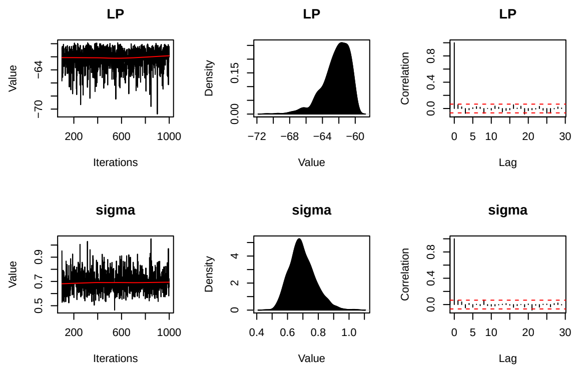

Figure 3: Plots of Marginal Posterior Samples

```
> caterpillar.plot(Fit, Parms=1:4)
```

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, introduced in the **LaplacesDemon** package), 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 Laplace's Demon, simply use the `predict` function:

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

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 two components: `y` and `yhat`. 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`. Each set of samples is used to calculate `mu`, which is the expectation of `y`, and `mu` is reported here as `yhat`. When there are few discrepancies between `y` and $\mathbf{y}^{rep}$, the model is considered to fit well to the data.

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

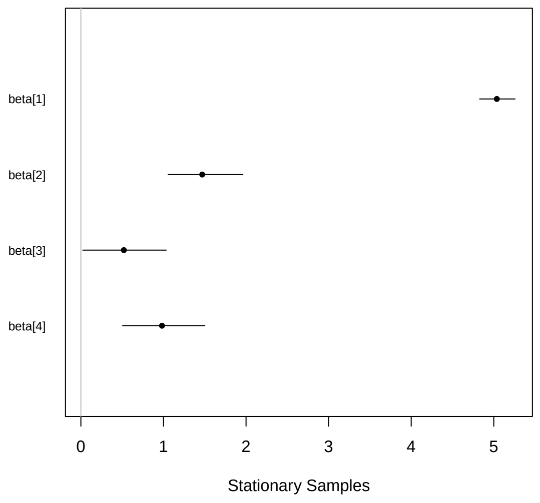

Figure 4: Caterpillar Plot

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

Concordance:  0.5384615
Discrepancy Statistic:  623.386
L-criterion: 21.104, S.L: 0.43
Records:
          y  Mean    SD    LB Median    UB    PQ Discrep
1  4.174387 4.308 0.152 4.001  4.312 4.584 0.816   0.766
2  5.361292 5.407 0.204 5.002  5.407 5.806 0.592   0.050
3  6.089045 5.267 0.211 4.855  5.275 5.680 0.000  15.162
4  5.298317 5.214 0.185 4.852  5.216 5.575 0.312   0.209
5  4.406719 4.216 0.167 3.916  4.209 4.554 0.142   1.302
6  2.197225 3.803 0.167 3.469  3.801 4.138 1.000  92.256
7  5.010635 4.409 0.148 4.119  4.409 4.711 0.000  16.533
8  1.609438 3.820 0.166 3.488  3.818 4.152 1.000 177.234
9  4.343805 4.380 0.130 4.125  4.382 4.632 0.614   0.077
10 4.812184 4.513 0.139 4.240  4.515 4.781 0.010   4.603
11 4.189655 4.254 0.144 3.976  4.263 4.526 0.661   0.203
12 4.919981 4.380 0.130 4.125  4.382 4.632 0.000  17.266
13 4.753590 4.240 0.136 3.973  4.243 4.508 0.000  14.165
14 4.127134 4.204 0.140 3.928  4.207 4.480 0.707   0.298
15 3.713572 3.976 0.158 3.656  3.976 4.287 0.951   2.759
16 4.672829 4.551 0.138 4.272  4.556 4.811 0.186   0.776
17 6.930495 7.265 0.310 6.659  7.265 7.880 0.849   1.160
18 5.068904 4.564 0.152 4.256  4.564 4.863 0.000  10.986
19 6.775366 6.534 0.370 5.843  6.530 7.255 0.252   0.423
```

```
20 6.553933 6.711 0.376 5.987  6.708 7.454 0.648   0.175
21 4.890349 4.926 0.143 4.650  4.921 5.211 0.591   0.062
22 4.442651 4.501 0.156 4.185  4.505 4.789 0.649   0.141
23 2.833213 4.067 0.151 3.756  4.073 4.353 1.000  66.395
24 4.787492 4.663 0.136 4.395  4.662 4.923 0.182   0.836
25 6.933423 7.312 0.308 6.684  7.320 7.927 0.898   1.514
26 6.180017 6.300 0.215 5.869  6.308 6.710 0.714   0.311
27 5.652489 5.038 0.181 4.702  5.038 5.389 0.000  11.484
28 5.429346 4.480 0.145 4.195  4.479 4.777 0.000  42.895
29 5.634790 5.912 0.528 4.922  5.911 6.932 0.700   0.276
30 4.262680 4.012 0.155 3.702  4.010 4.323 0.059   2.614
31 3.891820 4.276 0.149 3.967  4.280 4.546 0.996   6.597
32 6.613384 6.535 0.350 5.852  6.528 7.228 0.397   0.050
33 4.919981 4.292 0.134 4.032  4.296 4.553 0.000  21.892
34 6.541030 6.094 0.287 5.524  6.101 6.636 0.058   2.424
35 6.345636 5.876 0.227 5.401  5.875 6.321 0.014   4.265
36 3.737670 4.307 0.179 3.993  4.302 4.677 0.997  10.106
37 7.356280 8.326 0.460 7.443  8.320 9.285 0.984   4.454
38 5.739793 4.749 0.113 4.540  4.749 4.969 0.000  76.389
39 5.517453 4.894 0.165 4.559  4.900 5.225 0.000  14.278
```

The `summary.demonoid.ppc` function returns a list with 4 components when `y` is continuous (different output occurs for categorical dependent variables when given the argument `Categorical=TRUE`):

- `Concordance` is the predictive concordance of Gelfand (1996), that indicates the percentage 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 1% 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 *et˜al.* (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 4.308. Most importantly, `PQ[1]` is 0.816, indicating that 81.6% 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]`=1. Therefore, `yhat[6,]` was not a good replication of `y[6]`, because the distribution of `yhat[6,]`

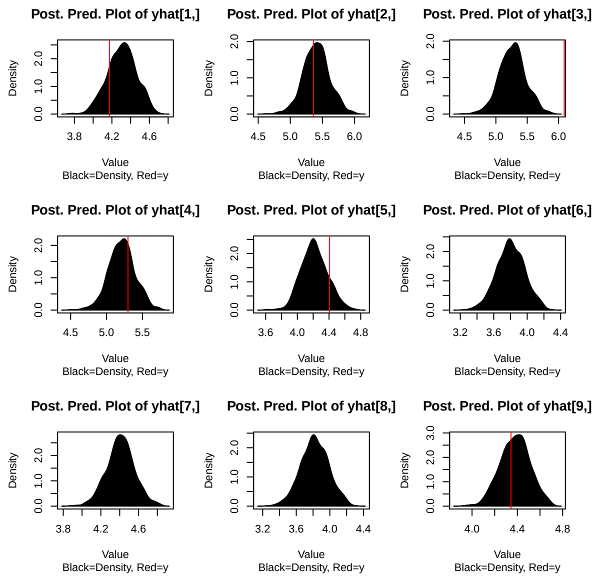

Figure 5: Posterior Predictive Plots

is almost always greater than `y[6]`. While `y[1]` is within the 95% probability interval of `yhat[1,]`, the 95% probability interval of `yhat[6,]` is above `y[6]` 99.8% 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)
```

The `Importance` function is not presented here in detail, but is 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 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.

9. General Suggestions

Following are general suggestions on how best to use Laplace's Demon:

- 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.
- MCMC is a stochastic method 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. Also, during this phase, 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 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 t-walk 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 Laplace's Demon uses more RAM than the computer has available, then the computer will crash. This can be used to estimate the maximum number of iterations 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.
- If adaptation does not seem to improve estimation (if adaptation is used) or the initial movement in the chains is worse than expected, then consider optimizing the initial values with the `LaplaceApproximation` function, changing the initial values, or setting all initial values equal to zero so the `LaplacesDemon` function will use the `LaplaceApproximation` function. In MCMC, initial values are most effective when the starting points are close to the target distributions (though, if the target distributions were known *a priori*, then there would be little point in much of this). When

initial values are far enough away from the target distributions to be in low-probability regions, the algorithms (both Laplace Approximation and MCMC) may take longer than usual. Some MCMC algorithms use a proposal covariance matrix, and these algorithms will struggle more as the proposal covariance matrix approaches near-singularity (though some algorithms, such as the t-walk or AMWG do not use a proposal covariance matrix). In extreme examples, it is possible for the proposal covariance matrix to become singular, which will stop Laplace's Demon. If there is no information available to make a better selection, then randomize the initial values with the `GIV` function and use `LaplaceApproximation`. Centered and scaled predictors also help by essentially standardizing the possible range of the target distributions.

- 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 Laplace's Demon 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 `LaplacesDemon` function. Simply change the parameter of interest as appropriate and place the constrained value back in the `parm` vector.

- `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 `Geweke.Diagnostic` may indicate stationarity (lack of a trend) when it does not exist, and this most likely occurs when too few thinned samples remain. 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.

- Use a two-phase approach with Laplace's Demon (unless using a non-adaptive but self-adjusting algorithm such as the t-walk algorithm), where the first phase consists of using an adaptive algorithm (usually AHMC, AMWG, AMM, AM, DRAM, RAM, SAMWG, or USAMWG) 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, such as DRM, HMC, MWG, RWM, SMWG, THMC, or USMWG. 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.

10. Independence and Observability

Laplace's Demon was designed with independence and observability in mind. By independence, it is meant that a goal was to minimize dependence on other software. Laplace's Demon requires only base R. 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 base R, Laplace's Demon 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 **LaplacesDemon** 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 Laplace's Demon 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. Laplace's Demon 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 Laplace's Demon is often significantly faster than other MCMC software that was 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 Laplace's Demon. To access any function, simply enter the function name and press enter. For example, to print the code for **LaplacesDemon** to the R console, simply enter:

```
> LaplacesDemon
```

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

```
> LaplacesDemon:::RWM
```

Laplace's Demon 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 may expand into other HPC areas in the future. For the moment, HPC refers to parallel processing.

The `LaplacesDemon` function is extended with the `LaplacesDemon.hpc` function to 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 any MCMC algorithm in the `LaplacesDemon` function. The `LaplacesDemon.hpc` extension uses the **parallel** package of base R.

An example of using `LaplacesDemon.hpc` is to simultaneously update three 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 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 two current limitations with `LaplacesDemon.hpc`. First, multiple chains must begin with the same, rather than dispersed, initial values. This is only a limitation until it can be programmed to accept dispersed initial values. Second, during parallel processing, `Status` messages do not appear. Once submitted, the user must wait until it finishes without knowing its status.

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 first chain 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.

One of many attractive possible future HPC extensions is to include the inter-chain adaptive (INCA) approach for adaptive MCMC (Craiu, Rosenthal, and Yang 2009; Solonen, Ollinaho, Laine, Haario, Tamminen, and Jarvinen 2012). INCA uses parallel chains that are independent, except that they share the adaptive component, and this sharing speeds convergence. Recent attempts at coding this have been unsuccessful due to unfamiliarity with how best to utilize package **parallel**. If you know how to extend the MCMC algorithms in `LaplacesDemon` with INCA, or how to incorporate dispersed initial values, then please email `statisticat@gmail.com`.

12. Details

The **LaplacesDemon** package uses two broad types of numerical approximation algorithms: Laplace Approximation and Markov chain Monte Carlo (MCMC), and Approximate Bayesian Computation (ABC) may be estimated within each. Each is described below, but MCMC is emphasized.

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. 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), 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 the AMM MCMC algorithm evaluates once per iteration), making most of the MCMC algorithms faster per iteration. The attractiveness of Laplace Approximation is that it typically improves the objective function better than MCMC when the parameters are in low-probability regions (in which MCMC algorithms may suffer unreasonably low acceptance rates) until an adaptive MCMC has "learned" how to move better. Laplace Approximation is also typically faster 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. Bernardo and Smith (2000) note that Laplace Approximation is an attractive numerical approximation algorithm, 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, estimate covariance, and save time for the user.

Most optimization algorithms assume that the logarithm of the unnormalized joint posterior density is defined and differentiable. An approximate gradient is taken for each initial value as the difference in the logarithm of the unnormalized joint posterior density due to a slight increase versus decrease in the parameter.

Adaptive Gradient Ascent

With adaptive gradient ascent, at 10 evenly-space times, `LaplaceApproximation` attempts several step sizes, which are also called rate parameters in other literature, and selects the best step size from a set of 10 fixed options. Thereafter, each iteration in which an improvement does not occur, the step size shrinks, being multiplied by 0.999.

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.

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. This is the default algorithm (`method="LBFGS"`) for `LaplaceApproximation`, which calls `method="L-BFGS-B"` in the `optim` function of base R.

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).

Afterward

After `LaplaceApproximation` finishes, due either to early convergence or completing the number of specified iterations, it approximates the Hessian matrix of second derivatives, 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 `LaplacesDemon`, 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.

12.3. Markov Chain Monte Carlo

Although the `LaplacesDemon` function may be assisted by Laplace Approximation, Laplace's Demon mainly accomplishes numerical approximation with Markov chain Monte Carlo (MCMC) algorithms, 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, Random-Walk Metropolis (RWM), slice sampling, and many others, including hybrid algorithms such as Hamiltonian Monte Carlo, Metropolis-within-Gibbs (Tierney 1994), and algorithms for specific methods, such as Updating Adaptive Metropolis-within-Gibbs for state-space models (SSMs). RWM was developed first (Metropolis, Rosenbluth, M.N., and Teller 1953), and Metropolis-Hastings was a generalization of RWM (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.

While designing Laplace's Demon, the primary goal in numerical approximation was generalization. The most generalizable MCMC algorithm is the Metropolis-Hastings (MH) generalization of the RWM algorithm. The MH algorithm extended RWM to include asymmetric proposal distributions. Having no need of asymmetric proposals, Laplace's Demon uses variations of the original RWM algorithm, which use symmetric proposal distributions, specifically Gaussian proposals (and sometimes others, such as in the RAM algorithm of Vihola (2011)). For years, the main disadvantage of the RWM and MH 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. Slice sampling 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.

There are valid ways to tune the RWM algorithm as it updates. This is known by many names, including adaptive Metropolis and adaptive MCMC, among others. A brief discussion follows of MCMC algorithms in **LaplacesDemon**.

Block Updating

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, so each block may be treated differently. A block may contain one or more variables. Advantages of block updating are that a different MCMC algorithm may be used for each block (or variable, for that matter), creating a more specialized approach, and the acceptance of a newly proposed state is likely to be higher than sampling from all target distributions at once in high dimensions. Disadvantages of block updating are that correlations probably exist between variables between blocks, and each block is updated while holding the other blocks constant, ignoring these correlations of variables 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, block updating allows a more specialized approach at the expense of accuracy, generalization, and speed. Laplace's Demon generally avoids block updating, though this increases the importance that the initial values are not in low-probability regions, and may cause Laplace's Demon to have chains that are slow to begin moving.

Random-Walk Metropolis

In MCMC algorithms, each iterative estimate of a parameter is part of a changing state. The succession of states or iterations constitutes a Markov chain when the current state is influenced only by the previous state. In random-walk Metropolis (RWM), a proposed future estimate, called a proposal[10] or candidate, of the joint posterior density is calculated, and a ratio of the proposed to the current joint posterior density, called α, is compared to a random number drawn uniformly from the interval (0,1). In practice, the logarithm of the unnormalized joint posterior density is used, so $\log(\alpha)$ is the proposal density minus the current density. The proposed state is accepted, replacing the current state with probability 1 when the proposed state is an improvement over the current state, and may still be accepted if the logarithm of a random draw from a uniform distribution is less than $\log(\alpha)$. Otherwise, the proposed state is rejected, and the current state is repeated so that another proposal may be estimated at the next iteration. By comparing $\log(\alpha)$ to the log of a random number when $\log(\alpha)$ is not an improvement, random-walk behavior is included in the algorithm, and it is possible for the algorithm to backtrack while it explores.

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, RWM 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. Nonetheless, multimodal target distributions are often problematic.

Multiple parameters usually exist, and therefore correlations may occur between the parameters. Most MCMC algorithms in Laplace's Demon are modified to attempt to estimate multivariate proposals, thereby taking correlations into account through a covariance matrix. If a failure is experienced in attempting to estimate multivariate proposals in the Adaptive Metropolis (AM) of Haario, Saksman, and Tamminen (2001) here, or if the acceptance rate is less than 5%, then Laplace's Demon temporarily resorts to single-component proposals by updating one randomly-selected parameter, and will continue to attempt to return to

[10]Laplace's Demon allows the user to constrain proposals in the `Model` function. Laplace's Demon generates a proposal vector, which is passed to the `Model` function in the `parm` vector. In the `Model` function, the user may constrain the proposal to prevent the sampler from exploring certain areas of the state space by altering the proposed values and placing them back into the `parm` vector, which will be passed back to Laplace's Demon.

multivariate proposals at each iteration.

Throughout the RWM algorithm, the proposal covariance or variance remains fixed. The user may enter a vector of proposal variances or a proposal covariance matrix, and if neither is supplied, then Laplace's Demon estimates both before it begins, based on the number of variables.

The acceptance or rejection of each proposal should be observed at the completion of the RWM algorithm as the acceptance rate, which is the number of acceptances divided by the total number of iterations. If the acceptance rate is too high, then the proposal variance or covariance 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 or covariance 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%.

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 Laplace's Demon. 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 remaining MCMC algorithms in the **LaplacesDemon** package are now presented alphabetically.

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 the HMC section below. 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).

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. If AHMC is used for adaptation, then the final, non-adaptive algorithm should be HMC.

Adaptive Metropolis

The Adaptive Metropolis (AM) algorithm of Haario *et˜al.* (2001) adapts based on the observed covariance matrix from the history of the chains[11]. Laplace's Demon uses a variation of the Adaptive Metropolis (AM) algorithm of Haario *et˜al.* (2001).

Given the number of dimensions (K) or parameters, the optimal scale of the proposal variance, also called the jumping kernel, has been reported as $2.4^2/K$[12] 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. Laplace's Demon uses a scale that is accurate to more decimals: $2.381204^2/K$. There are algorithms in statistical literature that attempt to optimize this scale, such as the `RAM` algorithm.

Haario *et˜al.* (2001) tested their algorithm with up to 200 dimensions or parameters. It has been tested in Laplace's Demon with as many as 2,600 parameters, so it is capable of large-scale Bayesian inference. To effectively finish adapting, AM must solve the proposal covariance matrix, and this can be slow in high dimensions.

The version of AM in Laplace's Demon should be capable of more dimensions than the AM algorithm as it was presented, because when Laplace's Demon experiences an error in multivariate AM, or when the acceptance rate is less than 5%, it defaults to random-scan single-component adaptive proposals (Haario, Saksman, and Tamminen 2005). Although single-component adaptive proposals should take more iterations to converge, the algorithm is limited in dimension only by the random-access memory (RAM) of the computer.

In both the multivariate and single-component cases, the AM algorithm begins with a fixed proposal variance or covariance that is either estimated internally or supplied by the user. Next, the algorithm begins, and it does not adapt until the iteration is reached that is specified by the user in the `Adaptive` argument of the algorithm specification list. Then, the

[11] 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).

[12] The optimal proposal standard deviation in this case is approximately $2.4/\sqrt{K}$.

algorithm will adapt with every `n` iterations according to the `Periodicity` argument, also in the algorithm specification list. Therefore, the user has control over when the AM algorithm begins to adapt, and how often it adapts. The value of the `Adaptive` argument in Laplace's Demon is chosen subjectively by the user according to their confidence in the accuracy of the initial proposal covariance or variance. The value of the `Periodicity` argument is chosen by the user according to their patience: when the value is 1, the algorithm will adapt continuously, which will be slower to calculate. The AM algorithm adapts the proposal covariance or variance according to the observed covariance or variance in the entire history of all parameter chains, as well as the scale factor.

As recommended by Haario *et˜al.* (2001), there are two tricks that may be used to assist the AM algorithm in the beginning. Although Laplace's Demon does not use the suggested "greedy start" method (and will instead use Laplace Approximation when sample size permits), it uses the second suggested trick of shrinking 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, Laplace's Demon 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.

The AM algorithm performs very well in practice, though each adaptation is time-consuming after numerous iterations. The Adaptive-Mixture Metropolis (AMM) of Roberts and Rosenthal (2009) and Robust Adaptive Metropolis (Vihola 2011) are extensions of the AM algorithm.

Adaptive Metropolis-within-Gibbs

The Adaptive Metropolis-within-Gibbs (AMWG) algorithm is presented in (Roberts and Rosenthal 2009; Rosenthal 2007). 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 in **LaplacesDemon**, 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 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.

The advantages of AMM over AMWG are that it takes correlation into account 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 AMWG for more). Disadvantages of AMM compared to 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 RWM.

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, Laplace's Demon will attempt one alternate proposal when using the DRAM (see below) 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 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 Laplace's Demon, 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. Considered as a non-adaptive algorithm, it is acceptable to conclude model updates with this algorithm, rather than following up with RWM.

Laplace's Demon 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.

Each problem is different, and this can be a useful algorithm. In general, however, it is more likely that other algorithms are used.

Delayed Rejection Adaptive Metropolis

The Delayed Rejection Adaptive Metropolis (DRAM) algorithm is merely the combination of both DRM (or DR) and AM (Haario, Laine, Mira, and Saksman 2006). DRAM has been demonstrated as 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.

The DRAM algorithm is useful to assist the AM algorithm 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, non-adaptive algorithm should be RWM.

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). 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 [50%, 65%], 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. It is easy to experience problems when inexperienced, but HMC is a rewarding algorithm once proficiency is acquired. As can be expected, the adaptive extension, AHMC, will also be easier, since ϵ is adapted and does not require tuning.

Partial derivatives are required, and hence the parameters must be differentiable 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 $L + 2KL$ evaluations of the model specification function per iteration.

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) is an adaptive version of HML in which ϵ is adapted based on recent history of acceptance and rejection.

Metropolis-within-Gibbs

Metropolis-within-Gibbs (MWG) is a hybrid algorithm, combining Metropolis-Hastings and Gibbs sampling, and was suggested in Tierney (1994). Also referred to as Metropolis within Gibbs or Metropolis-in-Gibbs, it is a componentwise algorithm in which 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 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. MWG runs most efficiently when the acceptance rate of each parameter is 0.44, which is the optimal acceptance rate of a target distribution that is univariate and Gaussian.

The advantage of MWG over RWM is that it is more efficient with information per iteration, so convergence is faster in iterations. The disadvantage of MWG is that 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 decreases. Since fewer iterations are completed in a given time-interval, the possible amount of thinning is also at a disadvantage. MWG may be used for simple models with few parameters, but is not recommended here for large and complex models.

Robust Adaptive Metropolis

The AM and 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. The RAM algorithm does not require a burn-in period prior to adaptation. The RAM algorithm 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.

This version of RAM does not force positive-definiteness of the variance-covariance matrix, and adapts only when it is positive-definite. Alternative versions exist elsewhere that force positive-definiteness, but in testing here, it seems better to allow it to adapt only when it is positive-definite without coercion.

RAM is slow when `Periodicity=1` (where it performs best), and does not seem to perform well when `Periodicity` is ≥ 100. A general recommendation is `Periodicity=10`.

In testing the RAM algorithm, it has not been observed to obtain its acceptance rate goal and some wild fluctuations have been observed in the proposal variance after many iterations in some cases. In some models it does well, nonetheless it cannot be recommended as a first choice for a generalized algorithm.

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 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 AMWG for more), and frequent adaptation may be desirable, but slow. If RAM is used for adaptation, then the final, non-adaptive algorithm should be 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 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 SMWG.

Sequential Metropolis-within-Gibbs

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

Tempered Hamiltonian Monte Carlo

The Tempered Hamiltonian Monte Carlo (THMC) algorithm is an extension of the HMC algorithm to include another algorithm specification: `Temperature`. The `Temperature` 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.

In this implementation, the user specifies initial values for two chains, `Initial.Values` (as per usual) and `SIV`, which stands for secondary initial values. The secondary vector of initial values may be left to its default, `NULL`, in which case it is generated with the `GIV` function. However, it is recommended that the user specify `SIV` to be similar to, but unique from, `Initial.Values`, especially in complex models where `GIV` may have difficulty finding suitable values, or may arrive at very distant 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. In the "Examples" vignette, there are three examples for which the t-walk algorithm cannot function, no matter how initial values are specified: Discrete Choice Multivariate Probit, Multivariate Probit, and Multivariate Binary Probit. The reason is because a `Z` matrix with binary elements is calculated as parameters, and it will never be achieved that `Initial.Values` and `SIV` remain unique after passing through the `Model` specification function. The t-walk algorithm is applicable only with continuous parameters.

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 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).

The t-walk has four specification arguments, three of which are tuning parameters. The authors recommend using the default values. The first specification argument is `SIV`, and was explained previously. The n_1 specification argument affects the size of the subset of each set of points to adjust, and 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 40% 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.

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 each 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 MWG family, over multivariate adaptive algorithms such as AMM and 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 truly 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 single-component 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).

The t-walk algorithm seems best suited for high-dimensional problems, especially initial exploration. With enough iterations and thinning, the t-walk has produced excellent results in testing, and it has been subjected to an extreme test here on a model with 8,000 parameters. Due to limitations in computer memory (RAM), the model was updated several times for 30,000 iterations, and the thinned results were appended together. Convergence was not pursued.

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 SAMWG and 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 information. 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, in general, is with the t-walk algorithm. It is "self-adjusting" but non-adaptive, so it does not need to be followed up with a non-adaptive algorithm, and therefore, diminishing adaptation is not a concern. Although the t-walk algorithm requires more iterations than other algorithms, it iterates quickly despite its complexity. The t-walk algorithm improves more in the beginning than other samplers (and so does AMWG, and its relatives), which is a practical concern in applied settings. When updated with parallel chains, the t-walk is also more consistent than many other algorithms such as AMM or RAM; parallel t-walk chains tend to lock onto the correct solution quickly and in unison, while AMM or RAM (or many more algorithms) tend to wander separately until enough samples have been taken, since correct adaptation requires solving a covariance matrix. The t-walk algorithm does not need to dedicate an initial burn-in period to non-adaptivity (and this attractive property is shared by RAM and the AMWG family). A strong advantage of t-walk is that it is invariant to all but the most extreme correlation structures, and that a proposal covariance matrix is not used. Single-component algorithms such as the AMWG family do not use a proposal covariance matrix either, but these single-component algorithms are not invariant to correlation structures. A disadvantage of the t-walk is that, since it updates a randomly-selected subset at each iteration, the mixing of the chains appears worse as the number of parameters increases, until a large number of iterations and thinning is used. It is also difficult to consider the t-walk acceptance rate, because only a subset is selected for updating at each iteration, and only one of two chains. Considering everything above, as well as all other algorithms, the t-walk is the best general place to start, usually having a higher number of Indepedent (thinned) Samples per Minute (`ISM`), as indicated in the `Juxtapose` function. Now, special cases are considered.

In models with small dimensions, arbitrarily less than a couple hundred, and in general cases, the AMM algorithm performs best. In all tests to date, AMM is an improvement over AM and DRAM. The reason that AMM is less applicable in larger dimensions, say with thousands of parameters, is because it must solve the proposal covariance matrix, and the number of roughly half of its elements increases faster than the number of parameters. In models with small dimensions, AMM converges faster in minutes than other algorithms, though it requires more iterations than the AMWG family, and less than t-walk.

In models with large dimensions, from hundreds to thousands, there are two contenders: the AMWG family, and t-walk. The AMWG algorithm often has faster improvement and convergence in iterations, though this comes at the cost of time per iteration. The t-walk algorithm iterates much faster, but only a subset of parameters is considered. Consequently, many chains do not move for numerous iterations, and more iterations and thinning are required. A disadvantage of t-walk in large dimensions is that significantly more random-access memory (RAM) is required to complete more iterations. However, an advantage of t-walk is that, provided the necessary RAM is available, it is invariant to all but the most correlated structures, where the AMWG family is not.

In models with highly-correlated parameters, algorithms with multivariate proposals such as AMM or RAM are probably best, though t-walk also performs well in all but the most extreme cases. Single-component algorithms such as the AMWG family do not explicitly take correlated parameters into account, but try to use a random-scan ordering of parameters to improve. In models with small dimensions, as above, AMM is recommended. In models with large dimensions, t-walk is recommended.

State-space models (SSMs), or dynamic linear models (DLMs), are a special consideration.

The **LaplacesDemon** package has algorithms in the AMWG 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 t-walk 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. The recommended strategy here is to use parallel[13] RAM chains specified with the t-distribution. If the dimension is too large for a proposal covariance matrix to be practical, then parallel t-walk chains are recommended. Another alternative algorithm is THMC, though tuning is difficult. Parallel chains increase the chances that different chains may settle on different modes, and it is hoped that t-distributed proposals assist a chain in mode-switching behavior, rather than becoming confined only to one mode. Although parallel chains may be helpful in finding multiple modes, when the chains are combined with the `Combine` function for inference, each mode probably is not represented in a proportion correct for the distribution.

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.

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 `Geweke.Diagnostic` function of Geweke (1992) is iteratively applied to successively smaller tail-sections 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 number of thinned samples is divided into cumulative 10% groups, and the `Geweke.Diagnostic` function is applied by beginning with each cumulative group.

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). Laplace's Demon 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 a Cumulative Sample Function (`CSF`), Effective Sample Size (`ESS`), `Gelman.Diagnostic`, `Geweke.Diagnostic`, Integrated Autocorrelation Time (`IAT`), the Kolmogorov-Smirnov test (`KS.Diagnostic`), Monte Carlo Standard Error (`MCSE`), and both the `plot` and `PosteriorChecks` functions include multiple diagnostics.

[13]Parallel chains are enabled with the `LaplacesDemon.hpc` function.

13. Software Comparisons

To the best of my knowledge, there currently is no other software that 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, 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). 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, Gibbs sampling is not as generalizable, because it can function only when certain conjugate distributional forms are known *a priori* (Gilks and Roberts 1996). Moreover, Gibbs sampling was avoided for Laplace's Demon because it doesn't perform well with correlated variables or parameters, which usually exist, and I have been bitten by that *bug* many times.

The BUGS and JAGS families of MCMC software are excellent. BUGS is capable of several things that Laplace's Demon is not. BUGS allows the user to specify the model graphically as a directed acyclic graph (DAG) in Doodle BUGS. BUGS has other algorithms not yet in Laplace's Demon, such as reversible-jump. Many journal articles and textbooks in several

[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".

fields have been published that use BUGS, and many include example code[15].

Advantages of **LaplacesDemon** over JAGS and WinBUGS (not much experience with OpenBUGS) are: Bayes factors, comparison of algorithmic inefficiency, confidence in results (correlations do not cause trouble like in Gibbs), elicitation, enivornment is part of R for data manipulation and posterior analysis, estimation of random-access memory (RAM) usage, examples in documentation are more plentiful, faster with large data sets (when model specification avoids loops), Importance (Variable and Parameter), Laplace Approximation, log-posterior is available, likelihood-free estimation, marginal likelihood calculated automatically, modes (functions for multimodality), missing values do not require initial values (unless predicting predictors with missing predictors), 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, posterior predictive checks and discrepancy statistics, `predict` function for posterior predictive checks or scoring new data sets, suggested code is provided at the end of each run, trap errors do not exist or occur, and weights can be applied easily (such as weighting records in the likelihood).

The MCMC algorithms in Laplace's Demon are generalizable, and generally robust to correlation between variables or parameters. With larger data sets, there is no comparison: Laplace's Demon will deliver a converged model long before BUGS or JAGS. When correlations are high, almost any algorithm in Laplace's Demon will perform much better than Gibbs sampling.

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 2010) 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 2012) 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 absurd price.

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

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

14. Large Data Sets and Speed

An advantage of Laplace's Demon compared to other MCMC software is that the model is specified in a way that takes advantage of R's vectorization. BUGS and JAGS, 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, Laplace's Demon 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 AMM algorithm (`Adaptive=500` and `Periodicity=100`), and the initial value for each β set to 0.1 to bypass Laplace Approximation. This took 0.35 minutes with Laplace's Demon, according to a simple, linear regression[18]. It was nowhere near convergence, but updating the same model with the same data for 1,000 iterations took 15.02 minutes in JAGS 3.1.0.

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

For example, a 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 Laplace's Demon. Depending on circumstances, Laplace's Demon 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 such as C 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, and SAS. 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, Laplace's Demon currently leads the way in general-purpose Bayesian inference for users not specialized in vectorization with lower-level languages.

15. Conclusion

The **LaplacesDemon** package is a significant contribution toward Bayesian inference in R. In

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

[18]These updates were performed on a 2010 System76 Pangolin Performance laptop with 64-bit Debian Linux and 8GB RAM.

[19]To continue this example, JAGS may be *guessed* to take 20,000 iterations or 5.01 hours, and `LaplacesDemon` may take 400,000 iterations or 2.33 hours, and also have less autocorrelation in the chains due to more thinning. The slowest adaptive algorithm in Laplace's Demon is AMWG, which updated in 5.04 minutes, and should finish around 50,000 iterations or 4.2 hours. As sample size increases and when `for` loops are controlled, Laplace's Demon 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 Laplace's Demon completes 1,000 iterations in around 15 minutes, while JAGS is estimated to take over 10 days! This is what is meant by absurdity. So much for C or lower-level languages in that style of programming model specification functions!

turn, contributions toward the development of Laplace's Demon are welcome. Please send an email to statisticat@gmail.com with constructive criticism, reports of software bugs, or offers to contribute to Laplace's Demon.

References

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.

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.

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.

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.

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.

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.

Geweke J (1992). "Evaluating the Accuracy of Sampling-Based Approaches to the Calculation of Posterior Moments." *Bayesian Statistics*, **4**, 1–31.

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

Geyer C (2010). ***mcmc**: Markov Chain Monte Carlo*. R package version 0.8, URL `http://cran.r-project.org/web/packages/mcmc/index.html`.

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.

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.

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.

Haario H, Saksman E, Tamminen J (2005). "Componentwise Adaptation for High Dimensional MCMC." *Computational Statistics*, **20**(2), 265–274.

Hall B (2012). ***LaplacesDemon**: Software for Bayesian Inference*. R package version 12.07.02, URL `http://cran.r-project.org/web/packages/LaplacesDemon/index.html`.

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

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.

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 (2012). ***MCMCpack**: Markov chain Monte Carlo (MCMC) Package*. R package version 1.2-4, 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.

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.

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 Development Core Team (2012). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL `http://www.R-project.org`.

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

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.

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.

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

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.

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.

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

Affiliation:

Byron Hall
STATISTICAT, LLC
Farmington, CT
E-mail: statisticat@gmail.com
URL: http://www.statisticat.com/laplacesdemon.html