LaplacesDemon: A Complete Environment for Bayesian Inference within R

Statisticat, LLC

Abstract

LaplacesDemon, also referred to as LD, is a contributed R package for Bayesian inference, and is freely available at https://web.archive.org/web/20141224051720/http://www.bayesian-inference.com/indexe. The user may build any kind of probability model with a user-specified model function. The model may be updated with iterative quadrature, Laplace Approximation, MCMC, PMC, or variational Bayes. After updating, a variety of facilities are available, including MCMC diagnostics, posterior predictive checks, and validation. Hopefully, **LaplacesDemon** is generalizable and user-friendly for Bayesians, especially Laplacians.

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

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

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

In 1814, Laplace published his "Essai Philosophique sur les Probabilites", which introduced

[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.

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 Core Team 2014) package called **LaplacesDemon** (Statisticat LLC. 2015), which was designed without consideration for hard determinism, but instead with a lofty goal toward facilitating high-dimensional Bayesian (or Laplacian) inference[2], posing as its own intellect that is capable of impressive analysis. The **LaplacesDemon** R package is often referred to as LD. This article guides the user through installation, data, specifying a model, initial values, updating a numerical approximation algorithm, summarizing and plotting output, posterior predictive checks, general suggestions, discusses independence and observability, high performance computing, covers details of the algorithms, and introduces `https://web.archive.org/web/20141224051720/http://www.bayesian-inference.com/index`.

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

1. Installation

To obtain the **LaplacesDemon** package, simply download the source code from `https://web.archive.org/web/20141224051720/http://www.bayesian-inference.com/softwaredownload`, open R, and install the **LaplacesDemon** package from source:

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

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

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

[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.

the **LaplacesDemon** package does not depend on anything other than base R and its **parallel** package. **LaplacesDemonCpp** is an extension package that uses C++, and imports these packages: **parallel**, **Rcpp**, and **RcppArmadillo**. This tutorial introduces only **LaplacesDemon**, but the use of **LaplacesDemonCpp** is identical. Once installed, simply use the `library` or `require` function in R to activate the **LaplacesDemon** package and load its functions into memory:

```
> library(LaplacesDemon)
```

2. Data

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

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

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

`PGF` is an optional, but highly recommended, user-specified function. PGF stands for Parameter-Generating Function, and is used by the `GIV` function, where GIV stands for Generating Initial

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

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

Values. Although the `PGF` is not technically data, it is most conveniently placed in the list of data. When `PGF` is not specified and `GIV` is used, initial values are generated randomly without respect to prior distributions. To see why `PGF` was specified as it was, consider the following sections on specifying a model and initial values.

3. Specifying a Model

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

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

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

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

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

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

To specify a model, the user must create a function called `Model`. Here is an example for a linear regression model written in R code[7]:

```
> Model <- function(parm, Data)
+      {
+      ### Parameters
```

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

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

[7] A model specification function for the **LaplacesDemon** or **LaplacesDemonCpp** packages may be written and compiled in a faster language, such as in C++ via the **Rcpp** package family.

```
+      beta <- parm[Data$pos.beta]
+      sigma <- interval(parm[Data$pos.sigma], 1e-100, Inf)
+      parm[Data$pos.sigma] <- sigma
+      ### Log-Prior
+      beta.prior <- dnormv(beta, 0, 1000, log=TRUE)
+      sigma.prior <- dhalfcauchy(sigma, 25, log=TRUE)
+      ### Log-Likelihood
+      mu <- tcrossprod(beta, Data$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=LP,
+           yhat=rnorm(length(mu), mu, sigma), parm=parm)
+      return(Modelout)
+      }
```

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

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

where Θ is a set of parameters, $\mathbf{y}$ is the data, $\propto$ means 'proportional to'[8], $p(\Theta|\mathbf{y})$ is the joint posterior density, $p(\mathbf{y}|\Theta)$ is the likelihood, and $p(\Theta)$ is the set of prior densities.

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

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

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

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

The rest of the function specifies the parameters, log of the prior densities, and calculates

[8]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'.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4. Initial Values

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

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

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

5. Numerical Approximation

Compared to specifying the model in the `Model` function, updating a model is easy. Since pseudo-random numbers are involved, it's a good idea to set a 'seed' for pseudo-random number generation, so results can be reproduced. Pick any number you like, but there's only one number appropriate for a demon[9]:

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

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

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

```
> help(LaplacesDemon)
```

Here is one of many possible ways to begin:

[9]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'.

```
> Fit <- LaplacesDemon(Model, Data=MyData, Initial.Values,
+      Covar=NULL, Iterations=1000, Status=100, Thinning=1,
+      Algorithm="AFSS", Specs=list(A=500, B=NULL, m=100, n=0, w=1))
```

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

- 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. AFSS requires proposal covariance, and when not specified, will begin with a scaled identity matrix.
- The `Iterations=1000` argument indicates that the `LaplacesDemon` function will update 1,000 times before completion.
- The `Status=100` argument indicates that a status message will be printed to the R console every 100 iterations.
- The `Thinning=1` argument indicates that only ever Kth iteration will be retained in the output, and in this case, every iteration will be retained. See the `Thin` function for more information on thinning.
- The `Algorithm` argument requires the abbreviated name of the MCMC algorithm in quotes.
- Finally, the `Specs` argument contains specifications for each algorithm named in the `Algorithm` argument. The `AFSS` algorithm has several specifications. The `A` specification indicates at which iteration adaptation will stop, and it is arbitrarily set here so that it adapts for the first half, and is non-adaptive in the second half. The `B` specification is for blockwise sampling, which is not performed here. The `m` specification indicates the maximum number of steps when searching for the slice interval. The `n` specification is set to zero and indicates the number of previous adaptive iterations. The `w` specification is the step-size, which is adapted in this algorithm.

By running[10] the `LaplacesDemon` function, the following output was obtained:

```
> Fit <- LaplacesDemon(Model, Data=MyData, Initial.Values,
+      Covar=NULL, Iterations=1000, Status=100, Thinning=1,
+      Algorithm="AFSS", Specs=list(A=500, B=NULL, m=100, n=0, w=1))
```

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

```
Laplace's Demon was called on Tue Nov 12 04:55:42 2024

Performing initial checks...
Algorithm: Automated Factor Slice Sampler

Laplace's Demon is beginning to update...

Eigendecomposition will occur every 50 iterations.

Iteration: 100,   Proposal: Multivariate,   LP: -59.9
Iteration: 200,   Proposal: Multivariate,   LP: -61.3
Iteration: 300,   Proposal: Multivariate,   LP: -64.5
Iteration: 400,   Proposal: Multivariate,   LP: -64.2
Iteration: 500,   Proposal: Multivariate,   LP: -60.8
Iteration: 600,   Proposal: Multivariate,   LP: -60.7
Iteration: 700,   Proposal: Multivariate,   LP: -60.6
Iteration: 800,   Proposal: Multivariate,   LP: -64.5
Iteration: 900,   Proposal: Multivariate,   LP: -61.8
Iteration: 1000,   Proposal: Multivariate,   LP: -63.5

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

Laplace's Demon has finished.
```

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

```
> str(Fit)
```

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

```
> Fit$Acceptance.Rate

[1] 1
```

5.1. Warnings

During updating in `LaplacesDemon`, warnings are converted to errors, and the proposal is rejected. Warnings may appear due to checks before updating, or summarizing after updating, but not during updating. If chains appear to have numerous rejections after trying a variety

of samplers, then the model specification function may be producing warnings with certain configurations of parameters. If warnings continue to occur, then the priors or parameterization should be considered. An example is when a scale parameter for the posterior predictive distribution is allowed to be too small or large.

6. Summarizing Output

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

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

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

```
> Consort(Fit)

#############################################################
# Consort with Laplace's Demon                              #
#############################################################
Call:
LaplacesDemon(Model = Model, Data = MyData, Initial.Values = Initial.Values,
    Covar = NULL, Iterations = 1000, Status = 100, Thinning = 1,
    Algorithm = "AFSS", Specs = list(A = 500, B = NULL, m = 100,
        n = 0, w = 1))

Acceptance Rate: 1
Algorithm: Automated Factor Slice Sampler
Covariance Matrix: (NOT SHOWN HERE; diagonal shown instead)
  beta[1]   beta[2]   beta[3]   beta[4]     sigma
0.3199880 2.0464020 0.4999857 1.2537933 9.1074196

Covariance (Diagonal) History: (NOT SHOWN HERE)
Deviance Information Criterion (DIC):
         All Stationary
Dbar  84.615     84.615
pD   168.618    168.618
DIC  253.232    253.232
Initial Values:
[1] 0 0 0 0 1

Iterations: 1000
Log(Marginal Likelihood): -42.68294
Minutes of run-time: 0.02
```

```
Model: (NOT SHOWN HERE)
Monitor: (NOT SHOWN HERE)
Parameters (Number of): 5
Posterior1: (NOT SHOWN HERE)
Posterior2: (NOT SHOWN HERE)
Recommended Burn-In of Thinned Samples: 0
Recommended Burn-In of Un-thinned Samples: 0
Recommended Thinning: 5
Specs: (NOT SHOWN HERE)
Status is displayed every 100 iterations
Summary1: (SHOWN BELOW)
Summary2: (SHOWN BELOW)
Thinned Samples: 1000
Thinning: 1


Summary of All Samples
                Mean         SD       MCSE        ESS             LB      Median
beta[1]    5.0701049  0.3899535 0.03211598  260.31587   4.818605924   5.0506630
beta[2]    0.5704438  0.9824330 0.01151589 1000.00000   0.009458456   0.5992680
beta[3]    1.1395967  0.5200623 0.03069628  566.39272   0.502014421   1.1805121
beta[4]    0.9275659  0.7893701 0.04256565  892.31588   0.277554403   0.9106137
sigma      0.8306939  2.0306117 0.14928891  391.76819   0.564691559   0.7045643
Deviance  84.6148050 18.3639621 2.56631966   94.02143  78.227890078  82.0733478
LP       -63.4878465  9.2356894 1.28822084   95.34285 -68.247469775 -62.2126487
                 UB
beta[1]    5.2900910
beta[2]    1.1234251
beta[3]    1.7201589
beta[4]    1.6238980
sigma      0.9905811
Deviance  94.1364250
LP       -60.2898121


Summary of Stationary Samples
                Mean         SD       MCSE        ESS             LB      Median
beta[1]    5.0701049  0.3899535 0.03211598  260.31587   4.818605924   5.0506630
beta[2]    0.5704438  0.9824330 0.01151589 1000.00000   0.009458456   0.5992680
beta[3]    1.1395967  0.5200623 0.03069628  566.39272   0.502014421   1.1805121
beta[4]    0.9275659  0.7893701 0.04256565  892.31588   0.277554403   0.9106137
sigma      0.8306939  2.0306117 0.14928891  391.76819   0.564691559   0.7045643
Deviance  84.6148050 18.3639621 2.56631966   94.02143  78.227890078  82.0733478
LP       -63.4878465  9.2356894 1.28822084   95.34285 -68.247469775 -62.2126487
                 UB
beta[1]    5.2900910
beta[2]    1.1234251
```

```
beta[3]    1.7201589
beta[4]    1.6238980
sigma      0.9905811
Deviance  94.1364250
LP       -60.2898121

Demonic Suggestion

Due to the combination of the following conditions,

1. Automated Factor Slice Sampler
2. The acceptance rate (1) is within the interval [1,1].
3. At least one target MCSE is >= 6.27% of its marginal posterior
   standard deviation.
4. Each target distribution has an effective sample size (ESS)
   of at least 100.
5. Each target distribution became stationary by
   1 iteration.

WARNING: Diminishing adaptation did not occur.
Laplace's Demon has not been appeased, and suggests
copy/pasting the following R code into the R console,
and running it.

Initial.Values <- as.initial.values(Fit)
Fit <- LaplacesDemon(Model, Data=MyData, Initial.Values,
     Covar=Fit$Covar, Iterations=5000, Status=70, Thinning=5,
     Algorithm="AFSS", Specs=list(A=500, B=NULL, m=Fit$Specs$m,
     n=1000, w=Fit$CovarDHis[nrow(Fit$CovarDHis),]))

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 1000 iterations of 5 variables in 0.02 minutes. Of course this was fast, since there were only 39 records, and the form of the specified model was simple.

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

Since this is not a frequentist package, p-values are not part of any table with the `LaplacesDemon` function, and the marginal posterior distributions of the parameters and other variables have

already been summarized in `Fit`, there is no point to have an associated `summary` function. Going one more step toward useability, the `Consort` function of **LaplacesDemon** allows the user to consort with Laplace's Demon about the output object.

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

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

In the above output, Laplace's Demon is appeased. However, if any of these five conditions is unsatisfactory, then Laplace's Demon is not appeased, and suggests it should continue updating, and that the user should copy, paste, and execute its suggested R code. Here are the criteria it measures against. The final algorithm must be non-adaptive, so that the Markov property holds (this is covered in section 12.5.2). The acceptance rate of most algorithms is considered satisfactory if it is within the interval [15%, 50%][11]. LMC or MALA must be in the interval [40%, 80%], and others (AFSS, AGG, ESS, GG, OHSS, SGLD, Slice, and UESS) have an acceptance rate of 100%. For more information on acceptance rates, see the `AcceptanceRate` function. MCSE is considered satisfactory for each target distribution if it is less than 6.27% of the standard deviation of the target distribution. This allows the true mean to be within 5% of the area under a Gaussian distribution around the estimated mean. ESS is considered satisfactory for each target distribution if it is at least 100, which is usually enough to describe 95% probability intervals. And finally, each variable must be estimated as stationary.

In this example, notice that all criteria have been met: MCSEs are sufficiently small, ESSs are sufficiently large, and all parameters were estimated to be stationary. Although the algorithm adapted in the first half, it was non-adaptive in the second half of the run, the Markov property holds, so let's look at some plots.

7. Plotting Output

The **LaplacesDemon** package has a `plot.demonoid` function to enable its own customized plots with `demonoid` objects. The variable `BurnIn` (below) may be left as it is so it will show only the stationary samples (samples that are no longer trending), or set equal to zero so that all samples can be plotted. In this case, only samples are considered that were generated while the algorithm was non-adaptive, so `BurnIn=500`.

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 ex-

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

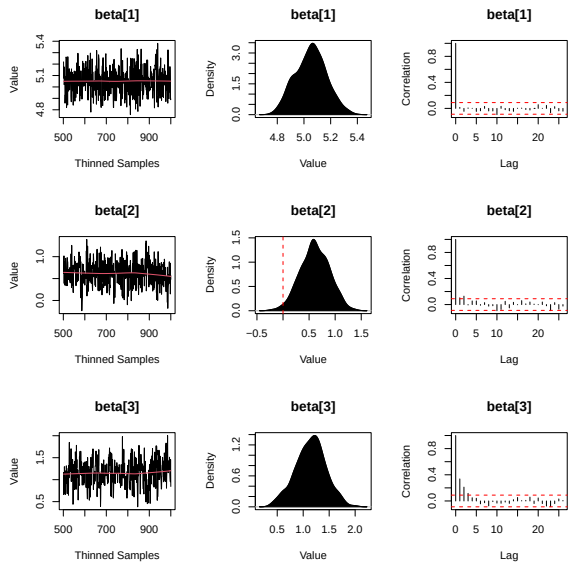

Figure 1: Plots of Marginal Posterior Samples

ample, `Parms=c("beta[1]","beta[2]")` would plot only the first two regression effects, and `Parms=NULL` will plot everything.

```
> plot(Fit, BurnIn=500, MyData, PDF=FALSE, Parms=NULL)
```

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

The Hellinger distances between batches of chains can be plotted with

```
> plot(BMK.Diagnostic(Fit$Posterior1[501:1000,]))
```

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

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

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

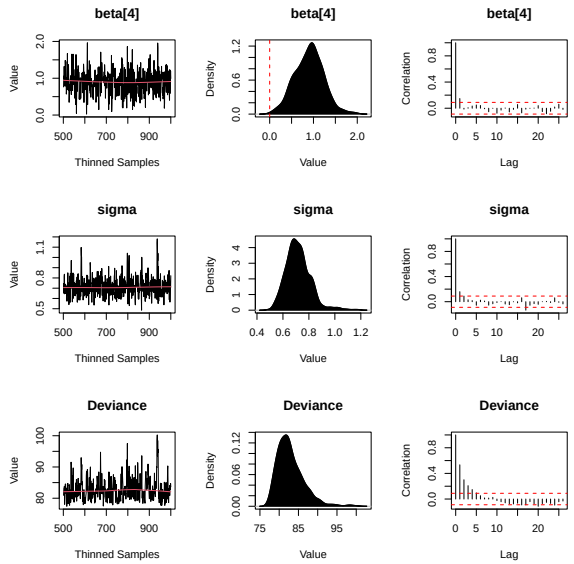

Figure 2: Plots of Marginal Posterior Samples

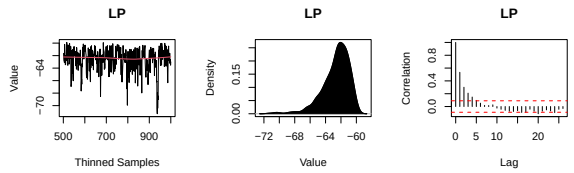

Figure 3: Plots of Marginal Posterior Samples

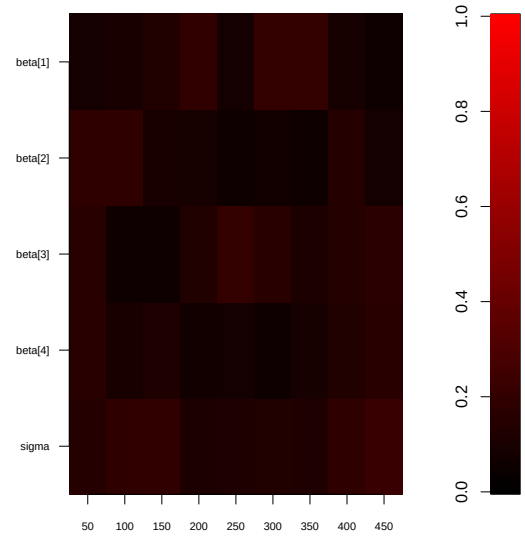

Figure 4: Hellinger Distances

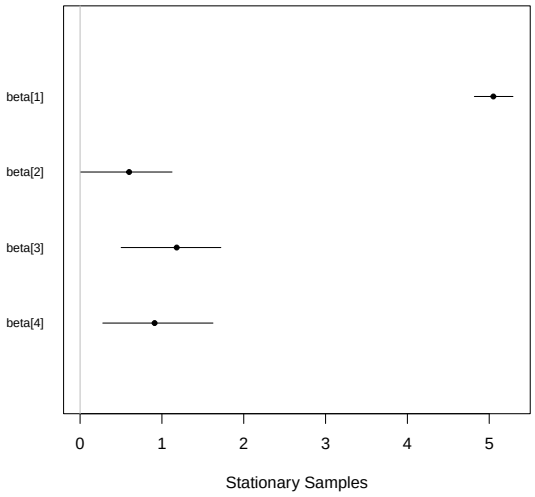

Figure 5: Caterpillar Plot

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

8. Posterior Predictive Checks

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

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

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

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

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

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

Bayesian Predictive Information Criterion:
   Dbar      pD    BPIC
 84.615 168.618 421.851
Concordance:  0.9230769
Discrepancy Statistic:  5.703
L-criterion: 88.337, S.L: 1.099
Records:
          y  Mean    SD    LB Median    UB    PQ Discrep
1  4.174387 4.191 1.075 2.648  4.145 5.873 0.481   0.000
2  5.361292 5.016 1.065 3.569  4.989 6.527 0.307   0.105
3  6.089045 6.102 2.795 4.412  6.168 7.730 0.547   0.000
4  5.298317 4.856 1.438 3.324  4.811 6.436 0.259   0.095
5  4.406719 5.058 1.374 3.483  4.995 6.606 0.797   0.225
6  2.197225 3.569 4.044 2.221  3.694 5.251 0.977   0.115
7  5.010635 4.626 1.791 3.136  4.585 6.073 0.275   0.046
8  1.609438 3.473 1.490 1.803  3.406 4.992 0.987   1.564
```

```
9  4.343805 4.566 2.972 3.065  4.664 6.140 0.666   0.006
10 4.812184 3.726 2.170 2.287  3.755 5.228 0.079   0.251
11 4.189655 3.623 1.869 2.182  3.614 5.117 0.185   0.092
12 4.919981 4.603 2.525 3.071  4.441 6.055 0.272   0.016
13 4.753590 4.458 2.117 2.873  4.471 6.021 0.326   0.020
14 4.127134 4.366 1.417 2.901  4.366 5.888 0.632   0.028
15 3.713572 3.474 2.260 1.856  3.367 4.897 0.325   0.011
16 4.672829 4.581 1.596 3.079  4.555 6.217 0.430   0.003
17 6.930495 6.916 2.210 5.326  6.910 8.503 0.493   0.000
18 5.068904 4.075 4.139 2.672  4.171 5.659 0.113   0.058
19 6.775366 6.768 0.938 5.227  6.764 8.365 0.498   0.000
20 6.553933 6.456 3.816 4.951  6.605 8.120 0.523   0.001
21 4.890349 4.535 1.018 2.976  4.540 6.064 0.332   0.122
22 4.442651 4.554 1.339 3.074  4.567 6.209 0.558   0.007
23 2.833213 4.503 1.517 3.072  4.547 6.012 0.983   1.212
24 4.787492 4.377 2.746 2.874  4.366 5.957 0.289   0.022
25 6.933423 6.443 1.675 4.828  6.449 8.062 0.232   0.086
26 6.180017 5.573 2.265 4.058  5.624 7.188 0.254   0.072
27 5.652489 5.562 1.628 4.153  5.565 7.119 0.446   0.003
28 5.429346 5.407 2.110 3.788  5.324 6.920 0.449   0.000
29 5.634790 6.368 1.966 4.696  6.322 7.910 0.799   0.139
30 4.262680 4.172 2.110 2.474  4.065 5.749 0.387   0.002
31 3.891820 4.492 1.675 2.995  4.531 6.050 0.802   0.129
32 6.613384 6.547 1.327 4.974  6.528 8.077 0.453   0.003
33 4.919981 4.510 1.021 2.973  4.504 6.081 0.287   0.161
34 6.541030 6.579 2.618 5.093  6.617 8.074 0.546   0.000
35 6.345636 6.340 3.127 4.881  6.401 8.042 0.519   0.000
36 3.737670 4.844 1.937 3.220  4.787 6.282 0.915   0.326
37 7.356280 7.772 3.359 5.955  7.657 9.305 0.655   0.015
38 5.739793 5.996 6.634 4.210  5.800 7.369 0.533   0.001
39 5.517453 4.490 1.174 3.016  4.452 5.842 0.074   0.767
```

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

- `BPIC` is the Bayesian Predictive Information Criterion of Ando (2007). BPIC is a variation of the Deviance Information Criterion (DIC) that has been modified for predictive distributions. For more information on DIC, see the accompanying vignette entitled "Bayesian Inference".
- `Concordance` is the predictive concordance of Gelfand (1996), that indicates the 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 92% 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.

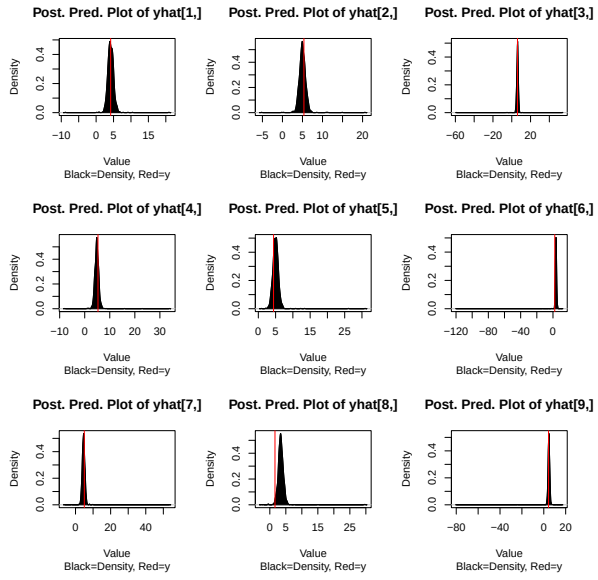

Figure 6: Posterior Predictive Densities

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

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

- The last part of the summarized output reports `y`, information about the distribution of `yhat`, and the predictive quantile (`PQ`). The mean prediction of `y[1]`, or $\mathbf{y}_1^{rep}$, given the model and data, is 4.191. Most importantly, `PQ[1]` is 0.481, indicating that 48.1% of the time, `yhat[1,]` was greater than `y[1]`, or that `y[1]` is close to the mean of `yhat[1,]`. Contrast this with the 6th record, where `y[6]`=2.197 and `PQ[6]`=0.977. Therefore, `yhat[6,]` was not a good replication of `y[6]`, because the distribution of `yhat[6,]` is almost always greater than `y[6]`. While `y[1]` is within the 95% probability interval of `yhat[1,]`, `yhat[6,]` is above `y[6]` 97.7% of the time, indicating a strong discrepancy between the model and data, in this case.

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

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

Among many other options, the fit may be observed:

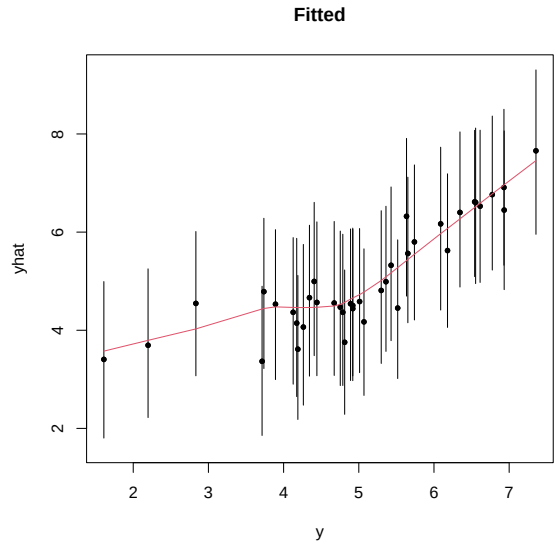

Figure 7: Posterior Predictive Fit

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

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

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

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

9. General Suggestions

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

- As suggested by Gelman (2008), continuous predictors should be centered and scaled. Here is an explicit example in R of how to center and scale a single predictor called `x`: `x.cs <- (x - mean(x)) / (2*sd(x))`. However, it is instead easier to use the `CenterScale` function provided in **LaplacesDemon**.
- Do not forget to reparameterize any bounded parameters in the `Model` function to be real-valued in the `parm` vector, and this is a good time to check for prior propriety with the `is.proper` function.
- If sufficient sample size is available, begin with a deterministic numerical approximation algorithm such as Laplace Approximation or variational Bayes.

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

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

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

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

- Once the final, intended model has begun (finally!), the mixing of the chains should be observed after a larger trial run, say, arbitrarily, for 10,000 iterations. If the chains do not mix as expected, then try a different algorithm, either one suggested by the `Consort` function (such as when diminishing adaptation is violated), or use the next least inefficient algorithm as indicated previously in the `Juxtapose` function.

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

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

- For MCMC, `Demonic Suggestion` is intended as an aid, not an infallible replacement for critical thinking. As with anything else, its suggestions are based on assumptions,

and it is the responsibility of the user to check those assumptions. For example, the `BMK.Diagnostic` may indicate stationarity (lack of a trend) when it does not exist. Or, the `Demonic Suggestion` may indicate that the next update may need to run for a million iterations in a complex model, requiring weeks to complete.

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

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

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

- After a deterministic numerical approximation algorithm has converged, consider following it up with a stochastic numerical approximation algorithm such as MCMC, if practical. When MCMC seems to have converged, consider updating the model again, this time with Population Monte Carlo (PMC). PMC may improve the model fit obtained with MCMC, and should reduce the variance of the marginal posterior distributions, which is desirable for predictive modeling.

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

10. Independence and Observability

The **LaplacesDemon** package was designed with independence and observability in mind. By independence, it is meant that a goal was to minimize dependence on other software. The **LaplacesDemon** package requires only base R, and the **parallel** package bundled with it. The variety of packages makes R extremely attractive. However, depending on multiple packages

can be problematic when different packages have functions with the same name, or when a change is made in one package, but other packages do not keep pace, and the user is dependent on packages being in sync. By avoiding dependencies on packages that are not in or accompanying base R, the **LaplacesDemon** package is attempting to be consistent and dependable for the user.

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

LaplacesDemonCpp is an optional extension package that uses C++, and is not an independent package in the sense that it imports **parallel**, but also **Rcpp** and **RcppArmadillo**. Once obtained and activated, its use is seamless to a **LaplacesDemon** user. **LaplacesDemonCpp** is a stand-alone replacement of **LaplacesDemon**, and is currently in development.

By observability, it is meant that the base **LaplacesDemon** package is written entirely in R. Certain functions could be sped up in another language such as C++, but this may prevent some R users from understanding the code. The base **LaplacesDemon** package is intended to be open and accessible. The optional **LaplacesDemonCpp** package is available for faster computations via C++. 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.

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

```
> LaplacesDemon
```

Most undocumented, internal-only functions are exported in the namespace and have an alias in the LaplacesDemon-package.Rd file.

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

11. High Performance Computing

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

11.1. Big Data

There are several definitions for big data. Here, big data is defined as data that is too big for the computer memory (RAM). The `BigData` function enables updating a Bayesian model with big data by reading in and processing smaller batches or chunks of data and performing a user-specified function on the batch before combining and outputing the result, so the entire

data set does not consume RAM. `BigData` is also parallelized. The `read.matrix` function allows sampling from big data. Finally, the Stochastic Gradient Descent (SGD) algorithm (see 12.4.16) in `LaplaceApproximation` and the Stochastic Gradient Langevin Dynamics (SGLD) algorithm in `LaplacesDemon` are designed specifically for use with big data.

11.2. Parallel Processing

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

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

11.3. Iterative Quadrature

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

11.4. Parallel Approximation within a Chain

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

11.5. Parallel Sets of Independent Chains

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

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

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

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

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

11.6. Parallel Sets of Interactive Chains

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

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

The Interchain Adaptation (INCA) algorithm (Craiu, Rosenthal, and Yang 2009; Solonen, Ollinaho, Laine, Haario, Tamminen, and Jarvinen 2012) performs Adaptive Metropolis (AM) with parallel chains that share the adaptive component, and this sharing speeds convergence. Whenever the chains are specified to adapt, adaptation is performed by pooling the historical covariance matrix across all parallel chains, and then returns the combined result to all chains. Network communication time slows the adaptation, but once returned to each CPU, chains iterate at their usual speed. This algorithm must be used with the `LaplacesDemon.hpc` function, and there is not an un-parallelized form of it.

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

11.7. Population Monte Carlo

The `PMC` function has been parallelized at each iteration to speed up the evaluation of the

model specification function over numerous importance samples.

11.8. Predict Functions

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

11.9. Model Specification Function

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

11.10. Parallelization Details

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

12. Details

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

12.1. Approximate Bayesian Computation

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

12.2. Importance Sampling

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

Population Monte Carlo

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

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

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

12.3. Iterative Quadrature

Quadrature is a historical term in mathematics that means determining area. Mathematicians of ancient Greece, according to the Pythagorean doctrine, understood determination of area of a figure as the process of geometrically constructing a square having the same area (squaring). Thus the name quadrature for this process.

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

A quadrature rule is an approximation of the definite integral of a function, usually stated as a weighted sum of function values at specified points within the domain of integration. The specified points are referred to as abscissae, abscissas, integration points, or nodes, and have associated weights. The calculation of the nodes and weights of the quadrature rule differs by the type of quadrature. There are numerous types of quadrature algorithms. Bayesian forms of quadrature usually use Gauss-Hermite quadrature (Naylor and Smith 1982), and placing

a Gaussian Process on the function is a common extension (O'Hagan 1991; Rasmussen and Ghahramani 2003) that is called 'Bayesian Quadrature'. Often, these and other forms of quadrature are also referred to as model-based integration.

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

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

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

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

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

Monte Carlo is a large family of sampling-based algorithms. O'Hagan (1987) asserts that Monte Carlo is frequentist, inefficient, regards irrelevant information, and disregards relevant information. Quadrature, he maintains (O'Hagan 1992), is the most Bayesian approach, and also the most efficient. In high dimensions, he concedes, a popular subset of Monte Carlo algorithms is currently the best for cheap model function evaluations. These algorithms are called Markov chain Monte Carlo (MCMC). High-dimensional models with expensive model

evaluation functions, however, are not well-suited to MCMC. A large number of MCMC algorithms is available in the `LaplacesDemon` function.

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

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

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

Adaptive Gauss-Hermite

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

There are four algorithm specifications: `N` is the number of univariate nodes, `Nmax` is the maximum number of univariate nodes, `Packages` accepts any package required for the model function when parallelized, and `Dyn.libs` accepts dynamic libraries for parallelization, if required. The number of univariate nodes begins at N and increases by one each iteration. The number of multivariate nodes grows quickly with N. Naylor and Smith (1982) recommend beginning with as few nodes as $N = 3$. Any of the following events will cause N to increase by 1 when N is less than `Nmax`:

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

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

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

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

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

Adaptive Gauss-Hermite Sparse Grid

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

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

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

Componentwise Adaptive Gauss-Hermite

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

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

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

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

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

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

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

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

12.4. Laplace Approximation

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

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

Since its introduction, the Laplace Approximation has been applied successfully in many disciplines. In the 1980s, the Laplace Approximation experienced renewed interest, especially in statistics, and some improvements in its implementation were introduced (Tierney and Kadane 1986; Tierney, Kass, and Kadane 1989). Only since the 1980s has the Laplace Approximation been seriously considered by statisticians in practical applications.

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

The speed of Laplace Approximation depends on the optimization algorithm selected, and typically involves many evaluations of the objective function per iteration (where an MCMC algorithm with a multivariate proposal usually evaluates once per iteration), making many MCMC algorithms faster per iteration. The attractiveness of Laplace Approximation is that it typically improves the objective function better than iterative quadrature, MCMC, and PMC when the parameters are in low-probability regions. Laplace Approximation is also typically faster than MCMC and PMC because it is seeking point-estimates, rather than attempting

to represent the target distribution with enough simulation draws. Laplace Approximation extends MLE, but shares similar limitations, such as its asymptotic nature with respect to sample size and that marginal posterior distributions are Gaussian. Bernardo and Smith (2000) note that Laplace Approximation is an attractive family of numerical approximation algorithms, and will continue to develop.

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

Most optimization algorithms assume that the logarithm of the unnormalized joint posterior density is defined and differentiable[12]. Some methods calculate an approximate gradient for each initial value as the difference in the logarithm of the unnormalized joint posterior density due to a slight increase in the parameter.

The user may select from numerous optimization algorithms:

Adaptive Gradient Ascent

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

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

BFGS

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

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

BHHH

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

Conjugate Gradient

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

Davidon-Fletcher-Powell

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

Hit-And-Run

This version of the Hit-And-Run (HAR) algorithm makes multivariate proposals and uses an adpative length parameter. The length parameter is adapted each iteration with the univariate version of the Robbins-Monro stochastic approximation in Garthwaite *et al.* (2010). The length shrinks when a proposal is rejected and expands when a proposal is accepted. This is the same algorithm as the HARM or Hit-And-Run Metropolis MCMC algorithm with adaptive length, except that a Metropolis step is not used.

Hooke-Jeeves

The Hooke-Jeeves algorithm (Hooke and Jeeves 1961) is a derivative-free, direct search method. Each iteration involves two steps: an exploratory move and a pattern move. The exploratory move explores local behavior, and the pattern move takes advantage of pattern direction. It is sometimes described as a hill-climbing algorithm. If the solution improves, it accepts the move, and otherwise rejects it. Step size decreases with each iteration. The

decreasing step size can trap it in local maxima, where it gets stuck and convergences erroneously. Users are encouraged to attempt again after what seems to be convergence, starting from the latest point. Although getting stuck at local maxima can be problematic, the Hooke-Jeeves algorithm is also attractive because it is simple, fast, does not depend on derivatives, and is otherwise relatively robust.

Levenberg-Marquardt

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

Limited-Memory BFGS

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

Nelder-Mead

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

Newton-Raphson

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

Particle Swarm Optimization

Of numerous Particle Swarm Optimization (PSO) algorithms, the Standard Particle Swarm Optimization 2007 (SPSO 07) algorithm is used here. A swarm of particles is moved according

to velocity, neighborhood, and the best previous solution. The neighborhood for each particle is a set of informing particles. PSO is derivative-free.

Resilient Backpropagation

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

Self-Organizing Migration Algorithm

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

Spectral Projected Gradient

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

Stochastic Gradient Descent

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

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

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

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

function must differ from the other examples found in this package in that multiple objects, such as `X` and `y` must be read from `Data$X`, where usually there is both `Data$X` and `Data$y`.

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

Symmetric Rank-One

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

Trust Region

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

Afterward

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

12.5. Markov Chain Monte Carlo

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

of the target distributions, which are marginal posterior distributions, later to be used for inference.

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

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

Blockwise Sampling

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

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

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

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

Slice Sampler (ESS), Hit-And-Run Metropolis (HARM), Metropolis-within-Gibbs (MWG), Random-Walk Metropolis (RWM), Robust Adaptive Metropolis (RAM), and the Univariate Eigenvector Slice Sampler (UESS).

Markov Chain Properties

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

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

Descriptions of the MCMC algorithms in the **LaplacesDemon** package are available online at https://web.archive.org/web/20150227012508/http://www.bayesian-inference.com/mcmc.

Sampler Selection

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

- diminishing adaptation, if applicable
- how many iterations it took the sampler to seem to converge
- how many minutes it took the sampler to seem to converge
- how quickly the sampler improved in the beginning
- `Juxtapose` results based on integrated autocorrelation time (`IAT`)

- mixing of the chains
- whether or not the sampler arrived at the correct solution

When the user is ready to select a general-purpose sampler, the best place to begin is with the AFSS algorithm. This is not to say that AFSS is the best sampler and everything else pales by comparison. Instead, AFSS is a great sampler with which to start in the general case, and for beginners. Although AFSS has several algorithm specifications, the default specifications are suitable for many cases.

A new user should not begin to learn AFSS and this package with a complicated and high-dimensional model. When this is necessary, the user of AFSS will need to learn how to create blocks of parameters and a list of proposal covariance matrices. In smaller cases, more suitable to learning, the user should not generally have to adjust the `m` or `w` specifications, and need only learn `A` and `n`. A new user should begin with `A=Inf` and `n=0`, and use code provided by the `Consort` function for the next run. When the user is satified that equilibrium is reached, then another run should be made without adaptation: `A=0`.

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

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

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

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

The `Demonic Suggestion` section of output from the `Consort` function also attempts to help the user to select a sampler. There are exceptions to each of these suggestions above. In some

cases, a particular algorithm will fail to update for a given example. Hopefully this section assists the user in selecting a sampler.

Afterward

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

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

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

12.6. Variational Bayes

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

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

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

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

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

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

Salimans2

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

13. Bayesian-Inference.com

Many additional resources may be found at https://web.archive.org/web/20141224051720/http://www.bayesian-inference.com/index, as well as other places online:

- Bayesian information is being compiled under https://web.archive.org/web/20150206004608/http://www.bayesian-inference.com/bayesian.
- C++ examples of model functions: https://web.archive.org/web/20140513065103/http://www.bayesian-inference.com/cpp/LaplacesDemonExamples.txt.
- MCMC algorithms are described at https://web.archive.org/web/20150430054005/http://www.bayesian-inference.com/mcmc.
- Merchandise may be found at http://www.zazzle.com/statisticat, such as **LaplacesDemon** t-shirts, coffee mugs, and more.
- **LaplacesDemon** and **LaplacesDemonCpp** development is public and occurs at https://github.com/LaplacesDemonR/LaplacesDemon and https://github.com/LaplacesDemonR/LaplacesDemonCpp, respectively.
- And, the home of **LaplacesDemon** is https://web.archive.org/web/20150430054143/http://www.bayesian-inference.com/software.

14. Conclusion

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

References

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

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.

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

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

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

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

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.

Fletcher R (1970). "A New Approach to Variable Metric Algorithms." *Computer Journal*, **13**(3), 317–322.

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

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

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

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

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

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

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

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

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

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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

R Core Team (2014). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL http://www.R-project.org.

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

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

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

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

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

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

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

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

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.

Statisticat LLC (2015). ***LaplacesDemon**: Complete Environment for Bayesian Inference*. R package version 15.03.19, URL http://www.bayesian-inference.com/software.

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.

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

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

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

Affiliation:

Statisticat, LLC
Hot Springs, SD
E-mail: defunct
URL: https://web.archive.org/web/20141224051720/http://www.bayesian-inference.com/index