MCMCpack Developer Documentation and Coding Specification
Version 0.5-1[*]

Andrew D. Martin Kevin M. Quinn

August 5, 2004

1 This Document

This document serves two purposes. First, it serves to formalize a coding and documentation standard so that the wide array of **MCMCpack** functions have a unified look and feel. Maintaining uniformity is particularly important as the package grows so that users can quickly learn to use newly available tools. The second goal of this document is to provide the formal documentation for the **MCMCpack** developer functions. Since these functions are hidden from ordinary users by the **MCMCpack** NAMESPACE, it is not possible to view their documentation files from within `R` using the `help()` function. As a result, this document is the most important resource for potential **MCMCpack** user/developers who wish to make use of these hidden functions.

By "**MCMCpack** user/developers" we mean users who would like to add new functionality to the **MCMCpack** package. **MCMCpack** user/developers may only want to add functionality for themselves, in which case there is no strict need for them to adhere completely to the coding and documentation standard laid out below. On the other hand, some **MCMCpack** user/developers may wish to have their additions distributed in future releases of **MCMCpack**. Adherence to the coding and documentation standard is extremely important for these developers. In Section 8 we discuss our policy for including user-contributed code in future releases of **MCMCpack**. This section also provides detailed instructions regarding how to submit proposed additions to **MCMCpack**.

The remainder of this document is organized as follows. Section 2 provides a general overview of **MCMCpack**. It also provides a detailed look at the package's implementation of an MCMC sampling method for the Gaussian linear model. The purpose of this detailed example is to familiarize readers with how a typical component of **MCMCpack** is constructed so that they have a better understanding of what needs to be done to construct custom components. Section 3 briefly discusses the **MCMCpack** NAMESPACE. The next two Sections provide a coding standard for `R` files

[*]This project is supported under National Science Foundation Grants SES-0350646 and SES-0350613. In addition, the Department of Political Science and the Weidenbaum Center at Washington University and the Department of Government and the Center for Basic Research in the Social Sciences at Harvard University have provided additional support. Neither the National Science Foundation, Washington University, nor Harvard University bear any responsibility for this software.

that hold model fitting functions and C and C++ source files that are used to do the MCMC sampling. Section 6 provides a documentation standard for **MCMCpack** model fitting functions. The next section details the use of an extremely powerful hidden function named `auto.Scythe.call()`. This function can be used to make calling compiled C++ code much easier. In addition, it can also be used to create template C++ source code and a template `*.Rd` file. Section 8 details how user/developers can submit additions to the primary **MCMCpack** developers for inclusion in future releases. Appendix A documents all of the hidden utility functions.

2 MCMCpack Package Overview

MCMCpack is an open-source, easy-to-use, R package that allows researchers to fit statistical models using Markov chain Monte Carlo methods. The software uses a conventional R interface, uses R for error checking, compiled C and C++ for model fitting (which is extremely fast), and the `coda` package for analysis of the posterior sample.

MCMCpack is also a development environment for user/developers who wish to implement new MCMC model fitting algorithms. While implementing a new model in **MCMCpack** requires more thought than implementing a new model in `WinBUGS` or `JAGS`, this is not necessarily a bad thing in that it forces developers to think carefully about how the sampling will be conducted and to have a better understanding of what is going on inside the black box. Further, as discussed below, **MCMCpack** contains a number of helper functions that make it very easy for a user/developer to go from a statistical/mathematical understanding of an MCMC sampling scheme to an R/C++ implementation of that sampling scheme.

Finally, for user/developers who are looking to distribute their MCMC-related code to a large body of researchers, **MCMCpack** provides a consistent user-interface and documentation standard. Since **MCMCpack** is part of the R system for data analysis and graphics (Ihaka and Gentleman, 1996) it has a very broad, knowledgeable user-base.

Fitting a model in **MCMCpack** begins with a call to an R function called `MCMCmodel()` that is defined in a file named `MCMCmodel.R`. Here "`model`" denotes a descriptive name for the model being fitted. Examples include `regress` [`MCMCregress()`] and `probit` [`MCMCprobit()`]. Typically, the `MCMCmodel()` function will take a number of arguments that govern the behavior of the MCMC sampling algorithm. In addition, the model formula, data, and prior parameters are passed to `MCMCmodel()` as arguments. Inside the `MCMCmodel()` R function three basic types of things happen. First, the inputs to `MCMCmodel()` are error checked and organized in a more useable form. A number of **MCMCpack** helper functions greatly aid this process. Second, a shared library written in a compiled language is called to do the MCMC sampling. The **MCMCpack** function `auto.Scythe.call()` substantially aids this process if the shared library was written in C or C++. The source code for this shared library is located a file called `MCMCmodel.cc`, `MCMCmodel.c`, or `MCMCmodel.f` depending on whether it is written in C++, C, or FORTRAN respectively. Finally, the posterior sample is labeled, coerced into a `coda mcmc` object, and returned.

To get a better sense of how this process works in practice we now look at the example of fitting a Bayesian linear model with Gaussian disturbances using the **MCMCpack** function `MCMCregress()`.

Readers familiar with the organization of the **MCMCpack** package, and R packages more generally, may wish to skip the next subsection.

2.1 A Detailed Example: MCMCregress

2.1.1 The R Code

To use **MCMCpack** to fit a linear regression with Gaussian disturbances a user would make a call from the R command line to the MCMCregress() function similar to the following:

```
data(LifeCycleSavings)
post.samp <- MCMCregress(sr~pop15+pop75+dpi+ddpi,
                         data=LifeCycleSavings,
                         burnin=500, mcmc=20000, thin=1,
                         b0=0, B0=0, c0=1.0, d0=10)
```

The example here uses the LifeCycleSavings data available in R base. This data set is loaded in the normal way with the data(LifeCycleSavings) command. With the data in memory we proceed to fit the regression model of interest. The output from the MCMCregress() function is sent to an object named post.samp. This object is of class mcmc as defined in the coda package. The first argument to MCMCregress() is an R model formula. This works just as in other R model fitting functions. The arguments burnin, mcmc, and thin specify the number of burn in scans the chain should be run for, the number of scans to take after the burn in phase is over, and the thinning interval. The arguments b0 and B0 specify the prior mean and prior precision of a multivariate normal prior for the regression coefficients. b0 can be either a scalar or vector and B0 can be either a scalar or matrix. If b0 is a scalar then the prior mean vector is taken to be a vector with elements equal to the given scalar. If B0 is a scalar then the prior precision is taken to be an identity matrix multiplied by the given scalar. In this case, the prior precision of 0 implies an improper uniform prior for the regression coefficients. The arguments c0 and d0 govern the inverse gamma prior on the error variance. More specifically, the shape parameter of the inverse gamma prior is c0/2 and the scale parameter of the prior is d0/2.

To get a sense of what is actually happening when the R function MCMCregress() is called let's take a look at the definition of this function. As noted above, this definition takes place in a file called MCMCregress.R. This file is located in the /R directory of an **MCMCpack** release. This file is the following:

File 1: MCMCregress.R

```
# MCMCregress.R samples from the posterior distribution of a Gaussian
# linear regression model in R using linked C++ code in Scythe
#
# Original written by ADM and KQ 5/21/2002
# Updated with helper functions ADM 5/28/2004
# Modified to meet new developer specification 6/18/2004 KQ
# Modified for new Scythe and rngs 7/22/2004 ADM

"MCMCregress" <-
  function(formula, data=parent.frame(), burnin = 1000, mcmc = 10000,
   thin=1, verbose = FALSE, seed = NA, beta.start = NA,
   b0 = 0, B0 = 0, c0 = 0.001, d0 = 0.001, ...) {

    # checks
    check.offset(list(...))
    check.mcmc.parameters(burnin, mcmc, thin)

    # seeds
    seeds <- form.seeds(seed)
    lecuyer <- seeds[[1]]
    seed.array <- seeds[[2]]
    lecuyer.stream <- seeds[[3]]

    # form response and model matrices
    holder <- parse.formula(formula, data)
    Y <- holder[[1]]
    X <- holder[[2]]
    xnames <- holder[[3]]
    K <- ncol(X)  # number of covariates

    # starting values and priors
    beta.start <- coef.start(beta.start, K, formula, family=gaussian, data)
    mvn.prior <- form.mvn.prior(b0, B0, K)
    b0 <- mvn.prior[[1]]
    B0 <- mvn.prior[[2]]
    check.ig.prior(c0, d0)

    # define holder for posterior density sample
    sample <- matrix(data=0, mcmc/thin, K+1)

    # call C++ code to draw sample
    auto.Scythe.call(output.object="posterior", cc.fun.name="MCMCregress",
                     sample=sample, Y=Y, X=X, burnin=as.integer(burnin),
                     mcmc=as.integer(mcmc), thin=as.integer(thin),
                     lecuyer=as.integer(lecuyer),
                     seedarray=as.integer(seed.array),
                     lecuyerstream=as.integer(lecuyer.stream),
                     verbose=as.integer(verbose), betastart=beta.start,
                     b0=b0, B0=B0, c0=as.double(c0), d0=as.double(d0))

    # pull together matrix and build MCMC object to return
    output <- form.mcmc.object(posterior,
                               names=c(xnames, "sigma2"),
                               title="MCMCregress Posterior Density Sample")
    return(output)
  }
```

File 1: MCMCregress.R

Looking at this file we see that lines 1-7 contain basic descriptive comments and a brief revision history. The actual definition of `MCMCregress()` begins on line 9. Lines 10-12 formally define the function's arguments along with default values.

Lines 15 and 16 use two hidden utility functions [`check.offset()` and `check.mcmc.parameters()`] to check whether an offset was passed in the call to `MCMCregress()` and to check whether the `burnin`, `mcmc`, and `thin` parameters are logically consistent. As with almost all of the hidden helper functions, `check.offset()` and `check.mcmc.parameters()` are defined in the file `hidden.R` in the `/R` directory of an **MCMCpack** release.

Lines 19-22 take the user supplied seed and, based on this value, determines whether the generator of L'Ecuyer et al. (2002) or the generator of Matsumoto and Nishimura (1998) will be used in the sampling, and then sets the seed and (in the case of the L'Ecuyer generator) the substream. The L'Ecuyer generator is particularly useful for running chains in parallel because of its ability to generate numerous independent substreams.

Lines 25-29 take the formula and dataframe supplied by the user and form a response vector (`Y`) and a matrix of predictors (`X`). The bulk of the work here is being done by the hidden function `parse.formula()`. This function takes a formula and a dataframe as arguments and returns a list with the response vector in the first position, the predictor matrix in the second position, and the names of the variables in the predictor matrix in the third position. `parse.formula()` is defined in `hidden.R`. This code is based on that from the `lm()` and `glm()` functions.

Starting values for the coefficient vector are created in line 32 by a call to the hidden function `coef.start()`. Note that this depends on the value of `beta.start` passed by the user. Starting values are for σ^2 are not needed because this parameter will be the first block of the Gibbs sampling algorithm used to fit the model. The call to `form.mvn.prior()` on line 33 checks the user-supplied values of `b0` and `B0` for admissibility and returns a list with a mean vector in the first position and a precision matrix in the second position. The call to `check.ig.prior()` on line 36 checks to make sure that the values of `c0` and `d0` passed by the user are both positive. `coef.start()`, `form.mvn.prior()`, and `check.ig.prior()` are all defined in `hidden.R`.

On line 39 a matrix called `sample` is created. This matrix will be used to store the sample from the posterior distribution. It has as many rows as there will be draws from the posterior and as many columns as there are parameters to be sampled.

Lines 42-49 are perhaps the most important lines in `MCMCregress.R`. It is here that a `C++` function named `MCMCregress()` is called to perform the MCMC sampling. The call to the `C++` function is done through the `auto.Scythe.call()` function. A thorough discussion of this function occurs in Section 7 of this document. For now, it is sufficient to note that `auto.Scythe.call()` takes care of some of the bookkeeping involved in passing matrices between `R` and `C++` using the `.C` interface. The `output.object` argument is a string that gives the name of the object that will be returned from the call to the `C++` function. `cc.fun.name` is the name of the `C++`function to be called. `sample` is the matrix used to store the draws from the MCMC sampling. `sample` is a required argument to `auto.Scythe.call()`. The remaining arguments to `auto.Scythe.call()` are data, prior parameters, or parameters that govern the behavior of the MCMC sampling.

In a moment we will examine the C++ function MCMCregress() that is being called. Before doing that let's look at lines 52-55 of MCMCregress.R. On lines 52-54 an object named output is created by a call to the hidden function form.mcmc.object(). This takes the output (named posterior) from our call to the C++ function MCMCregress() along with the number of burnin scans, mcmc scans after burnin, the thinning interval, the parameter names, and a brief descriptive title, and returns an object of class mcmc as defined in the coda package. On line 55 this mcmc object is returned to the user for summary and analysis. Typically this is done with functions defined in coda. For instance, to examine the Raftery and Lewis convergence diagnostic one could use the coda function raftery.diag(). Using the output from the call to MCMCregress() above:

```
> summary(post.samp)

Iterations = 1:20000
Thinning interval = 1
Number of chains = 1
Sample size per chain = 20000

1. Empirical mean and standard deviation for each variable,
   plus standard error of the mean:

                  Mean        SD  Naive SE Time-series SE
(Intercept) 28.6527542 7.5221305 5.319e-02      5.312e-02
pop15       -0.4628444 0.1477032 1.044e-03      1.039e-03
pop75       -1.7002924 1.1080198 7.835e-03      7.784e-03
dpi         -0.0003418 0.0009469 6.696e-06      6.655e-06
ddpi         0.4102689 0.1999713 1.414e-03      1.413e-03
sigma2      15.0431967 3.2943251 2.329e-02      2.641e-02

2. Quantiles for each variable:

                 2.5%       25%        50%        75%     97.5%
(Intercept) 13.888367 23.596171 28.6853288 33.7035332 43.365490
pop15       -0.750261 -0.562451 -0.4622327 -0.3636876 -0.175349
pop75       -3.849339 -2.442337 -1.7051552 -0.9628108  0.485041
dpi         -0.002230 -0.000964 -0.0003439  0.0003043  0.001492
ddpi         0.012083  0.277475  0.4106472  0.5445298  0.798647
sigma2       9.948094 12.702074 14.5936184 16.8817517 22.691228
```

2.1.2 The C++ Code

What exactly is happening when we call the C++ function called MCMCregress()? To get a sense of this, let's look at the definition of this function. This can be found in the file MCMCregress.cc which can be found in the /src directory of an **MCMCpack** release. The following is that file:

File 2: MCMCregress.cc

```
// MCMCregress.cc is a program that simulates draws from the posterior
// density of a linear regression model with Gaussian errors.
//
// The initial version of this file was generated by the
// auto.Scythe.call() function in the MCMCpack R package
// written by:
//
// Andrew D. Martin
// Dept. of Political Science
// Washington University in St. Louis
// admartin@wustl.edu
//
// Kevin M. Quinn
// Dept. of Government
// Harvard University
// kevin_quinn@harvard.edu
//
// This software is distributed under the terms of the GNU GENERAL
// PUBLIC LICENSE Version 2, June 1991.  See the package LICENSE
// file for more information.
//
// Copyright (C) 2004 Andrew D. Martin and Kevin M. Quinn
//
// This file was initially generated on Fri Jul 23 15:07:21 2004
//
// ADM and KQ 10/10/2002 [ported to Scythe0.3]
// ADM 6/2/04 [re-written using template]
// KQ 6/18/04 [modified to meet new developer specification]
// ADM 7/22/04 [modified to work with new Scythe and rngs]

#include "matrix.h"
#include "distributions.h"
#include "stat.h"
#include "la.h"
#include "ide.h"
#include "smath.h"
#include "MCMCrng.h"
#include "MCMCfcds.h"

#include <R.h>           // needed to use Rprintf()
#include <R_ext/Utils.h> // needed to allow user interrupts

using namespace SCYTHE;
using namespace std;

extern "C" {

   // simulate from posterior density and return an mcmc by parameters
   // matrix of the posterior density sample
   void MCMCregress(double *sampledata, const int *samplerow,
   const int *samplecol, const double *Ydata, const int *Yrow,
   const int *Ycol, const double *Xdata, const int *Xrow,
   const int *Xcol, const int *burnin, const int *mcmc,
   const int *thin, const int *lecuyer, const int *seedarray,
   const int *lecuyerstream, const int *verbose,
   const double *betastartdata, const int *betastartrow,
   const int *betastartcol, const double *b0data, const int *b0row,
   const int *b0col, const double *B0data, const int *B0row,
```

```
59    const int *B0col, const double *c0, const double *d0) {
60
61      // pull together Matrix objects
62      Matrix <double> Y = r2scythe(*Yrow, *Ycol, Ydata);
63      Matrix <double> X = r2scythe(*Xrow, *Xcol, Xdata);
64      Matrix <double> betastart = r2scythe(*betastartrow,
65        *betastartcol, betastartdata);
66      Matrix <double> b0 = r2scythe(*b0row, *b0col, b0data);
67      Matrix <double> B0 = r2scythe(*B0row, *B0col, B0data);
68
69      // define constants and form cross-product matrices
70      const int tot_iter = *burnin + *mcmc; // total number of mcmc iterations
71      const int nstore = *mcmc / *thin; // number of draws to store
72      const int k = X.cols ();
73      const Matrix <double> XpX = crossprod(X);
74      const Matrix <double> XpY = t(X) * Y;
75
76      // storage matrices
77      Matrix <double> betamatrix (k, nstore);
78      Matrix <double> sigmamatrix (1, nstore);
79
80      // initialize rng stream
81      rng *stream = MCMCpack_get_rng(*lecuyer, seedarray, *lecuyerstream);
82
83      // set starting values
84      Matrix <double> beta = betastart;
85
86      // Gibbs sampler
87      int count = 0;
88      for (int iter = 0; iter < tot_iter; ++iter) {
89        double sigma2 = NormIGregress_sigma2_draw (X, Y, beta, *c0,
90          *d0, stream);
91        beta = NormNormregress_beta_draw (XpX, XpY, b0, B0, sigma2,
92          stream);
93
94        // store draws in storage matrix (or matrices)
95        if (iter >= *burnin && (iter % *thin == 0)) {
96          sigmamatrix (0, count) = sigma2;
97          for (int j = 0; j < k; j++)
98            betamatrix (j, count) = beta[j];
99          ++count;
100       }
101
102       // print output to stdout
103       if(*verbose == 1 && iter % 500 == 0) {
104         Rprintf("\n\nMCMCregress iteration %i of %i \n",
105           (iter+1), tot_iter);
106         Rprintf("beta = \n");
107         for (int j=0; j<k; ++j)
108           Rprintf("%10.5f\n", beta[j]);
109         Rprintf("sigma2 = %10.5f\n", sigma2);
110       }
111
112       void R_CheckUserInterrupt(void); // allow user interrupts
113     } // end MCMC loop
114
115     delete stream; // clean up random number stream
116
117     // load draws into sample array
```

```
118       Matrix <double> storeagematrix = cbind (t (betamatrix), t (sigmamatrix));
119       const int size = *samplerow * *samplecol;
120       for(int i = 0; i < size; ++i)
121         sampledata[i] = storeagematrix[i];
122
123     } // end MCMCregress
124 } // end extern "C"
125
```

File 2: MCMCregress.cc

Lines 1-29 are just comments that describe the program, list the authors, provide licensing information, and provide a brief revision history. Lines 31-41 `#include` some libraries. The first 6 header files are parts of the Scythe Statistical Library (`http://scythe.wustl.edu/`). `MCMCrng.h` contains the declaration of a function that sets the pseudo-random number generator. `MCMCfcds.h` provides the declarations for functions that produce samples from various full conditional distributions. We'll look at this file in more detail later in this section. The `R` headers (`R.h` and `R_ext/Utils.h` are needed to allow reliable printing to the `R` console and to allow the user to interrupt the `C++` program. Since both of these files contain `extern "C"{}` declarations they can be `#included` outside the `extern "C"{}` declaration in `MCMCregress.cc`.

Lines 43 and 44 are `using` directives that make the names from the `SCYTHE` and `std` namespaces available.

Lines 50-123 contain the actual definition of the `C++` function `MCMCregress()` that is called from the `R` `MCMCregress()` function. Before looking at exactly what this function does, it is useful to note a couple of things about this function. First, note that it is wrapped in an `extern "C" { }` declaration. This is necessary to ensure the linkage specification of this code is compatible with that used for `R` internals (`R` is written is `C` not `C++`).

Looking at lines 50-59 we see that all of the arguments to `MCMCregress()` are pointers to (generally `const`) `int`s and `double`s. This is a requirement of using the `.C()` interface in `R`. As we will see in Section 7 `auto.Scythe.call()` actually builds and evaluates an expression that calls the given `C++` function using the `.C()` interface. For now it is sufficient to know that the expression that is generated and evaluated in lines 42-49 of `MCMCregress.R` is:

```
posterior <- .C("MCMCregress", sampledata = as.double(sample),
                samplerow = nrow(sample), samplecol = ncol(sample),
                Ydata = as.double(Y), Yrow = nrow(Y), Ycol = ncol(Y),
                Xdata = as.double(X), Xrow = nrow(X), Xcol = ncol(X),
                burnin = as.integer(burnin), mcmc = as.integer(mcmc),
                thin = as.integer(thin),
                lecuyer = as.integer(lecuyer),
                seedarray = as.integer(seed.array),
                lecuyerstream = as.integer(lecuyer.stream),
                verbose = as.integer(verbose),
                betastartdata = as.double(beta.start),
                betastartrow = nrow(beta.start),
                betastartcol = ncol(beta.start), b0data = as.double(b0),
```

```
b0row = nrow(b0), b0col = ncol(b0), B0data = as.double(B0),
B0row = nrow(B0), B0col = ncol(B0),
c0 = as.double(c0), d0 = as.double(d0),
PACKAGE = "MCMCpack")
```

Let's take a look at a few of the arguments being passed to `MCMCregress()`. The `.C()` interface does not allow one to pass `R` objects (as `R` objects) from `R` to `C`/ `C++` and vice versa. Instead, one has to work with the basic components of the `R` object. For instance, to pass an `R` matrix object to `C++` with `.C()` one needs to separately pass the data in the matrix, the number of rows, and the number of columns. Looking at the first three arguments (`double *sampledata`, `const int *samplerow`, and `const int *samplecol`) found on lines 50-51 we see how this is done. The first argument, `sampledata` is a pointer to a `double`. This actually points to the first element in an array holding the elements that make up the storage matrix for our sample (the `R` matrix `sample`). Note that this is not a pointer to a `const double`. We will be filling in this array with the sampled values later in this function and passing the result back to the calling `R` function. Since we are changing the values that `sampledata` points to, we cannot declare them to be `const`. The next two arguments (`const int *samplerow` and `const int *samplecol`) are pointers that point to the number of rows and the number of columns in the `R` matrix `sampple`. With knowledge of the contents of the `R` matrix `sample`, and its dimensions, we can create a copy of this matrix in the `MCMCregress()` `C++` function. We could accomplish this using the `Matrix` class defined in the Scythe Statistical Library. Note however, that in this example we never need to work with the MCMC sample as a matrix. As a result we do not bother to create a `Matrix` representation of these data.

However, we do want to work with the y vector, X matrix, vector of starting values, prior mean vector, and prior precision matrix as matrices. Let's look at the y vector. Similar to the above, the data necessary to create a copy of the `R Y` object are passed with three arguments: `const double *Ydata`, `const int *Yrow`, and `const int *Ycol`. Note that the pointer (`const double *Ydata`) to the contents of the `R Y` vector is `const`. This is a safeguard to prevent the elements of `Y` from being changed in the `C++` function. On line 62 we use the information in these three arguments to create a `Matrix` object called `Y` that is a copy of the `R Y` object. The function `r2scythe()` takes three arguments—an `int` giving the number of rows, an `int` giving the number of columns, a pointer to a the head of an array of `double`s—and returns a Scythe `Matrix` object. The other matrices are similarly constructed on lines 63-67.

The `C++ MCMCregress()` function also takes pointers to `int`s and `double`s as arguments. It is important to remember that these are pointers and not the thing being pointed to. For instance, if we want to add the number of burn in iterations and the number of iterations after burn in to get the total number of iterations we need to dereference the pointers `burnin` and `mcmc` during the addition. This can be done using the syntax `*burnin + *mcmc` or equivalently `burnin[0] + mcmc[0]` . See line 70.

Unlike `R`, `C` and `C++` require that the order of the arguments specified in a call to a function `foo()` be the same as the order of the arguments in the definition of `foo()`. As a result it is *extremely* important that the order of the arguments to `.C()` be the same as the order of the arguments to the `C` or `C++` function that `.C()` is calling. The `C++` template file produced by `auto.Scythe.call( , developer=TRUE)` will automatically be compatible with with the call to `.C()` it generates. When not using the `auto.Scythe.call()` interface to make calls to `C` or `C++`

one should be very careful to make sure the call and the function definition are compatible.

Returning to the structure of the C++ `MCMCregress()` function we see that a number of constants are defined on lines 70-74. For reasons of computational efficiency, it is wise to define variables that require non-trivial computation and that do not depend on the current state of the Markov chain outside the loop where the MCMC sampling occurs.

Lines 77 and 78 define two `Matrix` objects (`betamatrix` and `sigmamatrix`). These objects will be used to hold the sampled values of the coefficients and the error variance respectively. Note that `betamatrix` is defined to have `k` rows and number of columns equal to the number of iterations after burnin divided by the thinning interval, where `k` is defined to be the number of coefficients (see line 72). `sigmamatrix` has the same number of columns and only one row. By default, these matrices are filled with 0s.

Line 81 sets the seed and the substream of the pseudo random number generator

Line 84 sets the initial value of the coefficient vector (called `beta`) to the starting values passed from `R`.

Lines 88-113 are where the actual MCMC sampling takes place. At each iteration in this `for` loop a new value of the error variance parameter is drawn given the current value of the coefficient vector (lines 89-90) and a new value of the coefficient vector is drawn given the current value of the error variance parameter (lines 91-92). The functions [`NormIGregress_sigma2_draw()` and `NormNormregress_beta_draw()`] are declared in `MCMCfcds.h` and defined in `MCMCfcds.cc`. Both of these file are a located in the `/src` directory of an **MCMCpack** release.

If the sampler is past the burn in stage and the current iteration is evenly divisible by the thinning interval, then the newly sampled values of `sigma2` and `beta` are stored in `sigmamatrix` and `betamatrix` respectively. This occurs in lines 95-100. Note that there are two ways to access the internal data in a Scythe `Matrix` object. The syntax `A(i,j)` can be used to access the ith row and jth column element of a `Matrix A`. This is what is happening in line 96 and on the left hand side of the assignment in line 98. It is also possible to access the ith element of the internal data array of a Scythe `Matrix A` using the syntax `A[i]`. This is particularly convenient for accessing elements in a `Matrix` object that is equivalent to a vector. This is what is happening on the right hand side of the assignment on line 98. Scythe `Matrix` objects store data in row-major order. All indexing begins at 0.

If the verbose switch has been set to `TRUE` in the calling `R` function and the MCMC iteration number is evenly divisible by 500 then some basic output is printed. This occurs on lines 103-110. Line 104 prints the iteration number, lines 106-108 prints the current value of `beta`, and line 109 prints the current value of `sigma2`. Note that output is handled with the `Rprintf()` function. This works the same was as the `C` function `printf()` but is guaranteed to send output to the `R` console. It is also possible to handle output with `C++` iostreams. However, this prevents output from appearing on the `R` for Windows console.

Line 112 is a call to the `R_CheckUserInterrupt()` function which checks for an interrupt signal from the user and returns control back to the `R` console if an interrupt signal is found.

Line 113 closes off the `for` loop that began on line 88.

On line 115 the pointer called `stream` to a `rng` object is deleted. Failing to do this will result in a memory leak.

After the MCMC sampling has been completed we take the data in `betamatrix` and `sigmamatrix` and put in the array that `sampledata` points to. This occurs on lines 118-121. On line 118 a new `Matrix` called `storeagematrix` is formed by column binding the transpose of `betamatrix` and the transpose of `sigmamatrix`. Note the `R`-like syntax here. Line 119 defines a variable called `size` that gives the number of elements in the array that `sampledata` points to. The user needs to be sure to check that this is the same as the number of elements in `storeagematrix`. Lines 120 and 121 put each element of `storeagematrix` in the array that `sampledata` points to. At this point control returns to the calling `R` function with the MCMC sample in the array that `sampledata` points to.

At this point it may be instructive for the reader to take a look at the what the call to the `form.mcmc.object()` function is doing on lines 52-54 of `MCMCregress.R`. The object `posterior` that is created by the call to `auto.Scythe.call()` on lines 42-49 of `MCMCregress.R` is a list whose members are objects that were pointed to by the pointers that were sent to the `C++` function `MCMCregress()`. For instance, there is an element of this list called `sampledata` that is an array holding the MCMC sample. Similarly, there are elements of this list called `samplrow`, `samplecol`, `Ydata`, `Yrow`, etc. The call to `form.mcmc.object()` on lines 52-54 passes three things—the posterior object created by `auto.Scythe.call()`, a vector a names for the parameters that were sampled, and a brief title for the `mcmc` object to be created. The definition of `form.mcmc.object()` is:

```
"form.mcmc.object" <-
  function(posterior.object, names, title) {
     holder <- matrix(posterior.object$sampledata,
                      posterior.object$samplerow,
                      posterior.object$samplecol,
                      byrow=TRUE)

      output <- mcmc(data=holder, start=1,
                     end=posterior.object$mcmc,
                     thin=posterior.object$thin)
      varnames(output) <- names
      attr(output,"title") <- title
      return(output)
  }
```

Here we see that the matrix called `holder` that is created is just the MCMC sample put back into an easier-to-work-with matrix. It is assumed that the number of columns is equal to the number of parameters that were sampled. This matrix is then coerced into an object of class `mcmc`, names and title are attached, and the resulting `mcmc` object is returned.

2.1.3 The Documentation

The documentation for the `MCMCregress()` `R` function follows the same format as the usual `R` documentation (see the "Writing R Extensions" documentation available at:
`http://cran.r-project.org/doc/manuals/R-exts.pdf`). The documentation file (in `.Rd` format) for `MCMCregress()` is the following.

File 3: MCMCregress.Rd

```
\name{MCMCregress}
\alias{MCMCregress}
\title{Markov Chain Monte Carlo for Gaussian Linear Regression}
\description{
  This function generates a posterior density sample
  from a linear regression model with Gaussian errors using
  Gibbs sampling (with a multivariate Gaussian prior on the
  beta vector, and an inverse Gamma prior on the conditional
  error variance).  The user supplies data and priors, and
  a sample from the posterior density is returned as an mcmc
  object, which can be subsequently analyzed with functions
  provided in the coda package.
  }

\usage{
MCMCregress(formula, data = parent.frame(), burnin = 1000, mcmc = 10000,
   thin = 1, verbose = FALSE, seed = NA, beta.start = NA,
   b0 = 0, B0 = 0, c0 = 0.001, d0 = 0.001, ...) }

\arguments{
    \item{formula}{Model formula.}

    \item{data}{Data frame.}

    \item{burnin}{The number of burn-in iterations for the sampler.}

    \item{mcmc}{The number of MCMC iterations after burnin.}

    \item{thin}{The thinning interval used in the simulation.  The number of
      MCMC iterations must be divisible by this value.}

    \item{verbose}{A switch which determines whether or not the progress of
    the sampler is printed to the screen.  If TRUE, the iteration number, the
    \eqn{\beta}{beta} vector, and the conditional error variance is printed to
    the screen
    every 500 iterations.}

    \item{seed}{The seed for the random number generator.  If NA, the Mersenne
    Twister generator is used with default seed 12345; if an integer is
    passed it is used to seed the Mersenne twister.  The user can also
    pass a list of length two to use the L'Ecuyer random number generator,
    which is suitable for parallel computation.  The first element of the
    list is the L'Ecuyer seed, which is a vector of length six or NA (if NA
    a default seed of \code{rep(12345,6)} is used).  The second element of
    list is a positive substream number. See the MCMCpack
    specification for more details.}

    \item{beta.start}{The starting values for the \eqn{\beta}{beta} vector.
    This can either be a scalar or a
    column vector with dimension equal to the number of betas.
    The default value of of NA will use the OLS
    estimate of \eqn{\beta}{beta} as the starting value.  If this is a
    scalar,  that value will serve as the starting value
    mean for all of the betas.}

    \item{b0}{The prior mean of \eqn{\beta}{beta}.  This can either be a
    scalar or a
    column vector with dimension equal to the number of betas. If this
```

```
    takes a scalar  value, then that value will serve as the prior
    mean for all of the betas.}

    \item{B0}{The prior precision of \eqn{\beta}{beta}.  This can either be a
    scalar or a square matrix with dimensions equal to the number of betas.
    If this
    takes a scalar value, then that value times an identity matrix serves
    as the prior precision of beta. Default value of 0 is equivalent to
    an improper uniform prior for beta.}

  \item{c0}{\eqn{c_0/2}{c0/2} is the shape parameter for the inverse
    Gamma prior on \eqn{\sigma^2}{sigma^2} (the variance of the
    disturbances). The amount of information in the inverse Gamma prior
    is something like that from \eqn{c_0}{c0} pseudo-observations.}

    \item{d0}{\eqn{d_0/2}{d0/2} is the scale parameter for the
    inverse Gamma prior on \eqn{\sigma^2}{sigma^2} (the variance of the
    disturbances). In constructing the inverse Gamma prior,
    \eqn{d_0}{d0} acts like the sum of squared errors from the
    \eqn{c_0}{c0} pseudo-observations.}

    \item{...}{further arguments to be passed}
}

\value{
   An mcmc object that contains the posterior density sample.  This
   object can be summarized by functions provided by the coda package.
}

\details{
  \code{MCMCregress} simulates from the posterior density using
  standard Gibbs sampling (a multivariate Normal draw for the betas, and an
  inverse Gamma draw for the conditional error variance).  The simulation
  proper is done in compiled C++ code to maximize efficiency.  Please consult
  the coda documentation for a comprehensive list of functions that can be
  used to analyze the posterior density sample.

  The model takes the following form:
  \deqn{y_i = x_i ' \beta + \varepsilon_{i}}{y_i = x_i'beta + epsilon_i}
  Where the errors are assumed to be Gaussian:
  \deqn{\varepsilon_{i} \sim \mathcal{N}(0, \sigma^2)}{epsilon_i ~ N(0,
    sigma^2)}
  We assume standard, semi-conjugate priors:
  \deqn{\beta \sim \mathcal{N}(b_0,B_0^{-1})}{beta ~ N(b0,B0^(-1))}
  And:
  \deqn{\sigma^{-2} \sim \mathcal{G}amma(c_0/2, d_0/2)}{sigma^(-2) ~
    Gamma(c0/2, d0/2)}
  Where \eqn{\beta}{beta} and \eqn{\sigma^{-2}}{sigma^(-2)} are assumed
  \emph{a priori} independent.  Note that only starting values for
  \eqn{\beta}{beta} are allowed because simulation is done using
  Gibbs sampling with the conditional error variance
  as the first block in the sampler.
  }

\references{
   Andrew D. Martin, Kevin M. Quinn, and Daniel Pemstein.  2004.
   \emph{Scythe Statistical Library 1.0.} \url{http://scythe.wustl.edu}.

   Martyn Plummer, Nicky Best, Kate Cowles, and Karen Vines. 2002.
```

```
   \emph{Output Analysis and Diagnostics for MCMC (CODA)}.
   \url{http://www-fis.iarc.fr/coda/}.
}


\examples{
\dontrun{
line   <- list(X = c(-2,-1,0,1,2), Y = c(1,3,3,3,5))
posterior  <- MCMCregress(Y~X, data=line, verbose=TRUE)
plot(posterior)
raftery.diag(posterior)
summary(posterior)
}
}

\keyword{models}

\seealso{\code{\link[coda]{plot.mcmc}},
  \code{\link[coda]{summary.mcmc}}, \code{\link[stats]{lm}}}
```

File 3: MCMCregress.Rd

Compiling this file with LaTeX we get the following:

`MCMCregress` *Markov Chain Monte Carlo for Gaussian Linear Regression*

Description

This function generates a posterior density sample from a linear regression model with Gaussian errors using Gibbs sampling (with a multivariate Gaussian prior on the beta vector, and an inverse Gamma prior on the conditional error variance). The user supplies data and priors, and a sample from the posterior density is returned as an mcmc object, which can be subsequently analyzed with functions provided in the coda package.

Usage

```
MCMCregress(formula, data = parent.frame(), burnin = 1000, mcmc = 10000,
   thin = 1, verbose = FALSE, seed = NA, beta.start = NA,
   b0 = 0, B0 = 0, c0 = 0.001, d0 = 0.001, ...)
```

Arguments

`formula`	Model formula.

`data`	Data frame.
`burnin`	The number of burn-in iterations for the sampler.
`mcmc`	The number of MCMC iterations after burnin.
`thin`	The thinning interval used in the simulation. The number of MCMC iterations must be divisible by this value.
`verbose`	A switch which determines whether or not the progress of the sampler is printed to the screen. If TRUE, the iteration number, the β vector, and the conditional error variance is printed to the screen every 500 iterations.
`seed`	The seed for the random number generator. If NA, the Mersenne Twister generator is used with default seed 12345; if an integer is passed it is used to seed the Mersenne twister. The user can also pass a list of length two to use the L'Ecuyer random number generator, which is suitable for parallel computation. The first element of the list is the L'Ecuyer seed, which is a vector of length six or NA (if NA a default seed of `rep(12345,6)` is used). The second element of list is a positive substream number. See the MCMCpack specification for more details.
`beta.start`	The starting values for the β vector. This can either be a scalar or a column vector with dimension equal to the number of betas. The default value of of NA will use the OLS estimate of β as the starting value. If this is a scalar, that value will serve as the starting value mean for all of the betas.
`b0`	The prior mean of β. This can either be a scalar or a column vector with dimension equal to the number of betas. If this takes a scalar value, then that value will serve as the prior mean for all of the betas.
`B0`	The prior precision of β. This can either be a scalar or a square matrix with dimensions equal to the number of betas. If this takes a scalar value, then that value times an identity matrix serves as the prior precision of beta. Default value of 0 is equivalent to an improper uniform prior for beta.
`c0`	$c_0/2$ is the shape parameter for the inverse Gamma prior on σ^2 (the variance of the disturbances). The amount of information in the inverse Gamma prior is something like that from c_0 pseudo-observations.
`d0`	$d_0/2$ is the scale parameter for the inverse Gamma prior on σ^2 (the variance of the disturbances). In constructing the inverse Gamma prior, d_0 acts like the sum of squared errors from the c_0 pseudo-observations.
`...`	further arguments to be passed

Details

`MCMCregress` simulates from the posterior density using standard Gibbs sampling (a multivariate Normal draw for the betas, and an inverse Gamma draw for the conditional error variance). The simulation proper is done in compiled C++ code to maximize efficiency. Please consult the coda documentation for a comprehensive list of functions that can be used to analyze the posterior density sample.

The model takes the following form:

$$y_i = x_i'\beta + \varepsilon_i$$

Where the errors are assumed to be Gaussian:

$$\varepsilon_i \sim \mathcal{N}(0, \sigma^2)$$

We assume standard, semi-conjugate priors:

$$\beta \sim \mathcal{N}(b_0, B_0^{-1})$$

And:

$$\sigma^{-2} \sim \mathcal{G}amma(c_0/2, d_0/2)$$

Where β and σ^{-2} are assumed *a priori* independent. Note that only starting values for β are allowed because simulation is done using Gibbs sampling with the conditional error variance as the first block in the sampler.

Value

An mcmc object that contains the posterior density sample. This object can be summarized by functions provided by the coda package.

References

Andrew D. Martin, Kevin M. Quinn, and Daniel Pemstein. 2004. *Scythe Statistical Library 1.0.* `http://scythe.wustl.edu`.

Martyn Plummer, Nicky Best, Kate Cowles, and Karen Vines. 2002. *Output Analysis and Diagnostics for MCMC (CODA).* `http://www-fis.iarc.fr/coda/`.

See Also

`plot.mcmc`, `summary.mcmc`, `lm`

Examples

```
## Not run:
line   <- list(X = c(-2,-1,0,1,2), Y = c(1,3,3,3,5))
posterior  <- MCMCregress(Y~X, data=line, verbose=TRUE)
plot(posterior)
raftery.diag(posterior)
summary(posterior)
## End(Not run)
```

3 MCMCpack NAMESPACE and Helper Functions

The **MCMCpack** NAMESPACE file is the following:

File 4: MCMCpack NAMESPACE

```
1  useDynLib(MCMCpack)
2  import(coda)
3  import(MASS)
4
5  export(
6         ddirichlet,
7         dinvgamma,
8         diwish,
9         dnoncenhypergeom,
10        dtomogplot,
11        dwish,
12        MCMCbaselineEI,
13        MCMCdynamicEI,
14        MCMCfactanal,
15        MCMChierEI,
16        MCMCirt1d,
17        MCMCirtKd,
18        MCMClogit,
19        MCMCmetrop1R,
20        MCMCmixfactanal,
21        MCMCoprobit,
22        MCMCordfactanal,
23        MCMCpanel,
24        MCMCpoisson,
25        MCMCprobit,
26        MCMCregress,
27        rdirichlet,
28        read.Scythe,
29        rinvgamma,
30        riwish,
31        rnoncenhypergeom,
32        rwish,
33        tomogplot,
34        vech,
35        write.Scythe,
36        xpnd
37        )
38
```

File 4: MCMCpack NAMESPACE

The NAMESPACE file is used to accomplish three things. First, the `useDynLib()` command is used to load the shared library (or DLL on Windows) for the MCMCpack package. Lines 2-3 of the NAMESPACE takes care of package dependencies. In this case, we require the `coda` and `MASS` packages. It is possible to use a NAMESPACE file to only import a handful of functions from another package, but for our current purposes making these available to the user is useful. In Lines 5-37 we export the functions that are to be visible to the user. Note that the hidden functions are not exported! They are available to the developer within an `MCMCmodel()` function, but not to the user. All exported functions need to be documented. It is important to note that if you include documentation in `*.Rd` files for non-exported functions, the documentation will be included in the distribution but the user will be unable to execute the function. This would likely lead to confusion. Thus, we document all of our hidden functions using `*.Rd` files, but store them separately so they can be compiled and distributed only with this document. For more information about NAMESPACEs, see the "Writing R Extensions" manual.

4 The Organization of MCMC*.R Files

The model functions should be written with the following parameters in this order:

1. Model definition and/or data.
2. MCMC parameters.
3. Starting values.
4. Parameters for prior distributions.
5. Other options.

When implementing models, the first thing to do is to write the model down in some standard form. For a Gaussian linear regression:

$$y_i = \mathbf{x}_i'\boldsymbol{\beta} + \varepsilon_i \quad \varepsilon_i \sim \mathcal{N}(0, \sigma^2)$$

With priors:

$$\boldsymbol{\beta} \sim \mathcal{N}_K(\mathbf{b}_0, \mathbf{B}_0^{-1}) \quad \sigma^2 \sim \mathcal{IG}(c_0/2, d_0/2)$$

When implementing data input into the function, and when ordering parameters for priors, work left to right across the model, and for multilevel models, from the lowest level to the highest level. In this case, the function will take Y and then X, and the prior for $\boldsymbol{\beta}$ will be set before the prior for σ^2.

When implementing model functions, the programmer has a choice about whether to perform error checking in the `R` code or the `C` or `C++` code. While this depends on the application to some extent, in our experience it is best that the `R` code checks the conformability of the input data and prior parameters, and check that all arguments to the calling function take permissable values (the number of mcmc iterations is a positive integer, the number of mcmc iterations is evenly divisible by the thinning interval, etc.). Doing all of this in the `R` code (with appropriate error messages) makes it much easier for the user to find and correct errors, and it means passing fewer things to the compiled code, which decreases memory overhead. The hidden developer functions make this process much easier.

4.1 Model Definition and/or Data

The first set of parameters in **MCMCpack** model functions are used to pass the data. When possible, these parameters should use the `R` formula syntax (e.g., `y ~ x1 + x2`). When not, the data can be passed into the function directly using matrices.

4.2 Sampler Parameters

Every model function requires a set of parameters that govern the MCMC sampling. These parameters should be passed immediately after the data parameters. They should have default values so they need not be specified by the user unless requested. For some applications that defaults should be different from those listed here. But otherwise, use those specified here to standardize across models. Use these names, and in this order:

- `burnin = 1000` [the number of burn-in iterations for the sampler]
- `mcmc = 20000` [the number of iterations for the sampler after the burnin]
- `thin = 1` [the thinning interval, which must be a factor of `mcmc`]
- `tune` [the tuning parameter for Metropolis-Hastings algorithms]. Note that in some applications there may be more than one tuning parameter. If so, label them `tune.par1`, `tune.par2`, ...; where `par1` is the name of the first parameter block that requires a tuning parameter, `par2` is the name of the second parameter block that requires a tuning parameter and so forth.
- `verbose = FALSE` [print intermediate output from the sampler to the screen]. This part of the specification might change to allow the user to control the amount of output to the screen, such as picking which parameters to display and which iterations to display. For now, work under the dichotomous choice. The amount of output should be "reasonable," dictated by the specific application and the speed on existing hardware. By default, the `C++` code should print output every 500 iterations to the screen. The output should be produced by `Rprintf` statements that look like the following. Note that the name of the model function needs to be echoed. Nothing else should be sent to the screen, including system time calls, error traps, etc.

```
Rprintf("\n\nMCMCregress iteration %i of %i \n", (iter+1), tot_iter);
Rprintf("beta = \n");
for (int j=0; j<k; ++j)
  Rprintf("%10.5f\n", beta[j]);
Rprintf("sigma2 = %10.5f\n", sigma2);
```

- `seed = NA` [seed for the random number generator]. The NA option causes the program to use the default random number generator [the Mersenne twister (Matsumoto and Nishimura, 1998)] and seed (12345) in Scythe. If an integer is passed it is used to seed the Mersenne twister. The user can also pass a list of length two to use the L'Ecuyer (L'Ecuyer et al., 2002) random number generator, which is suitable for parallel computation. The first element of the list is the L'Ecuyer seed, which is a vector of length six or NA (if NA a default seed of `rep(12345,6)` is used). The second element of the list is a positive substream number.

4.3 Starting Values

To initialize the sampler, it is necessary to assign starting values to the model parameters. All models should be coded to use sensible default starting values (such as maximum likelihood estimates) unless the user specifies otherwise. All starting values should be passed in order, running from left to right, and low levels to high levels.

When writing the `MCMCmodel() R` function it will typically be best to set the default value of the starting value being passed to this function as `NA`. Within the body of this function a check should be made to see if the passed starting values are equal to `NA` and, if so, maximum likelihood estimates or some other reasonable value should be used as the starting values that are sent to the compiled code.

4.4 Parameters for Prior Distributions

Prior distributions are also assigned from left to right, and from low levels to high levels. Default priors should be specified. When priors are parameterized with vectors or matrices, use the same interface for starting values that are outline above. To standardize as much as possible across models, we adopt the following rules.

1. Parameters of prior distributions should be represented by Roman letters. It is also assumed that all model parameters are represented by Greek letters.
2. Scalar parameters should be represented by lower-case non-bold text, vectors should be represented by lower-case bold letters, and matrices should be represented by upper-case bold letters.
3. Parameters of prior distributions should have numeric subscripts corresponding to level at which they are defined in a (potentially) multilevel model. Prior parameters in a regular single-level model should all be subscripted with 0. The lowest level prior parameters in a two-level model should be subscripted with 0, and the prior parmeters at the highest level should be subscripted with a 1.
4. For prior distributions such as the multivariate Normal and the inverse Wishart that have two or more parameters that have different text representations according to rule 2 above, we use the Roman letter of the English spelling of the model parameter that has this prior distribution. For instance, if a vector "beta" has a multivariate normal prior we would write $\boldsymbol{\beta} \sim \mathcal{N}(\mathbf{b}_0, \mathbf{B}_0^{-1})$.
5. Where rule 3 is not possible we use the next available letters in the English alphabet to represent the prior parameters.
6. Normal priors should be parameterized in terms of precisions (inverse variances) rather than variances. This allows for an improper uniform prior if the precision matrix is equal to a matrix of 0s.
7. Unless otherwise noted, all priors should parameterized as in Gelman et al. (2003).

8. All other distributions should be added to the specification as needed. In the documentation, the names of the parameters should be spelled out in ASCII and coded in LaTeX for the hard copy.

9. In both the model write-up and the function call, names like $\mathbf{b}_0$ and `b0` should be used instead of **beta.mean** and `beta.mean`.

4.5 Other Options

After the priors are specified, it might be necessary to provide other options to the user, such as the specific algorithm to employ, additional options, and so forth. Those should be in the last set of parameters in the model function, and all should have default values.

5 The Organization of MCMC*.cc Files

In many respects, the organization of an `MCMC*.cc` model function is the same across all models. Of course each specific model has its own idiosyncrasies, but in general, each model will have these common components. The helper function `auto.Scythe.call()`—documented in Section 7—automatically generates `C++` code that contains all of these elements.

At the beginning of each model function it is important to have a block of comments that contain important information. This block should include a brief description of the function, the author(s) of the function, and the licensing terms. We also usually include some brief version history in the comment block.

After the comments come the headers. All of our model functions rely on the Scythe Statistical Library, so we need to include the files necessary to make Scythe functionality available: `matrix.h`, `distributions.h`, `stat.h`, `la.h`, and `ide.h`. There are two additional header files that are specific to **MCMCpack**: `MCMCrng.h` and `MCMCfcds.h`. `MCMCrng.h` contains the function used to initiate random number streams. Currently this works with the default Mersenne Twister generator and the optional parallel L'Ecuyer generator. It is modular so if we add other generators existing code will work. The file `MCMCfcds.h` contains update and utility functions that might be used in a model function. The final two headers are specific to `R`: `R.h` and `R_ext/Utils.h`. The former makes available a print command `Rprintf()` which should be use whenever echoing output to the screen. This command works on all platforms, including Windows. The latter command makes available user interrupts, so a CTRL-C will cleanly stop a simulation without crashing `R`.

After the headers comes the function that is called from `R`. This function returns a void, and must be wrapped in `extern "C"` so it can be interfaced with `R`. See the `MCMCregress()` example for typical arguments to this function. The first argument is usually an array of doubles which will be used to store the posterior density sample. We recommend putting full conditional distributions and other helper functions in the file `MCMCfcds.h`. This not only contributes to modularity, but makes the model function code much easier to update and maintain.

Within the body of the function, the following things must take place to initiate the sampler:

- All matrices must be pulled together from the array of doubles and dimensionality parameters passed from `R`. The utility function `r2scythe()` can be used to do this.
- Constants need to be defined, including the total number of iterations, the number of parameters in the model, and the number of draws to store. Also, other quantities used in the simulation, such as cross product matrices, should be done here.
- The storage matrix or matrices for the posterior density sample should be initiated.
- The random number stream should be initialized. The function `MCMCpack_get_rng()` does this automatically using the function defined in the header above. Note that this function returns a pointer to the stream which must be included in every call to a function that uses a random number generator.
- Starting values for some, if not all, of the parameters should be set.

What follows next is the actual MCMC algorithm which lies in a big loop. Within the loop comes three parts:

- The updates themselves. These are typically stored in other functions, although for certain models this is unnecessary.
- A rule to store the draws in the storage matrix or matrices. This needs to take into account the thinning interval to make sure only the correct draws are stored.
- Some meaningful text output to the screen that summarizes the progress of the sampler at some reasonable interval. These need to use the `Rprintf()` function so all `R` users will be able to see the progress of the sampler.

Within the MCMC loop it is also important to include the function `R_CheckUserInterrupt(void)` which allows for a clean termination of a simulation run using CTRL-C.

After the simulation takes place, there are two final parts of any model function. First, the random number stream needs to be cleaned up by deleting it. This keeps there from being a memory leak. Finally, the draws stored in the storage matrix or matrices need to be loaded into the array to be passed back to R. Scythe stores matrices in row major order, and `R` stores matrices in column major order, so this has to be done carefully. See the `MCMCregress()` example to see how this is done properly. Once the simulated values are loaded into this array, the function terminates.

6 The Organization of MCMC*.Rd Files

Documentation is relatively straightforward, as there are many common elements across model functions. In the Details section make sure to write down the model fully. In the References

section, make sure to cite Scythe (when it is employed), and references for the implementation of the algorithm, whether a textbook or article, if the algorithm is non-trivial. If the estimation strategy is particularly novel, it should be described fully in the Details section.

7 Using `auto.Scythe.call()`

In this section we provide an example of how to use the `auto.Scythe.call()` function to help automate the construction of an MCMC sampling scheme for a Laplace regression model. The model is given by:

$$y_i = \mathbf{x}_i'\boldsymbol{\beta} + \varepsilon_i, \quad \varepsilon_i \sim \mathcal{L}aplace(0, \sigma)$$

With priors:

$$\boldsymbol{\beta} \sim \mathcal{N}_K(\mathbf{b}_0, \mathbf{B}_0^{-1}), \quad \sigma \sim \mathcal{U}nif(c_0, d_0).$$

The MCMC sampling will be conducted using a Metropolis within Gibbs setup where $[\boldsymbol{\beta}|\mathbf{y}, \sigma]$ is sampled via a random walk Metropolis step and then $[\sigma|\mathbf{y}, \boldsymbol{\beta}]$ is similarly sampled via a random walk Metropolis step.

To begin the process of constructing the necessary sampling scheme we begin by writing a file that contains a version of the `R` function that will be used for model fitting along with some additional commands. This file is called `MCMClaplace.startbuild.R` and is shown below as File 5.

File 5: MCMClaplace.startbuild.R

```
source("../../MCMCpack/R/hidden.R")
source("../../MCMCpack/R/automate.R")


"MCMClaplace" <- function(formula, data=parent.frame(),
                          burnin=1000, mcmc=20000, thin=1,
                          beta.tune=1.2, sigma.tune=1.0,
                          verbose=FALSE, seed=NA, beta.start=NA,
                          sigma.start=NA, b0=0, B0=0, c0=1e-100, d0=1e100,
                          ...){

  ## checks
  check.offset(list(...))
  check.mcmc.parameters(burnin, mcmc, thin)
  if (d0 < c0)
    stop("\n d0 < c0 in MCMClaplace(), respecify and call MCMClaplace again.\n")
  ## seeds
  seeds <- form.seeds(seed)
  lecuyer <- seeds[[1]]
  seed.array <- seeds[[2]]
  lecuyer.stream <- seeds[[3]]

  ## form response and model matrices
  holder <- parse.formula(formula, data)
  Y <- holder[[1]]
  X <- holder[[2]]
  xnames <- holder[[3]]
  K <- ncol(X)  # number of covariates

  ## fit L1 regression to get starting values and proposal parameters
  library(quantreg)
  rq.out <- rq(formula, data=data)

  ## starting values and priors
  if (is.na(beta.start)){
    beta.start <- coef(rq.out)
  }
  beta.start <- coef.start(beta.start, K, formula, family=gaussian, data)
  if (is.na(sigma.start)){
    sigma.start <- mad(rq.out$residuals)
  }
  if (sigma.start < c0 || sigma.start > d0)
    stop("\n sigma.start not in (c0, d0) in MCMClaplace(), respecify and call MCMClaplace again.\n")

  mvn.prior <- form.mvn.prior(b0, B0, K)
  b0 <- mvn.prior[[1]]
  B0 <- mvn.prior[[2]]

  ## setup proposal variances
  beta.tune <- vector.tune(beta.tune, K)
  sigma.tune <- scalar.tune(sigma.tune)
  beta.cov <- summary(rq.out, se="boot", covariance=TRUE)$cov
  beta.propvar <- beta.tune %*% beta.cov %*% beta.tune
  sigma.propvar <- sigma.tune^2


   ## define holder for posterior density sample
  sample <- matrix(data=0, mcmc/thin, K+1)
```

```
59
60    auto.Scythe.call(output.object="posterior", cc.fun.name="MCMClaplace",
61                     developer=TRUE, help.file=TRUE,
62                     cc.file="MCMClaplace.template.cc",
63                     R.file="MCMClaplace.template.R",
64                     sample=sample, Y=Y, X=X, burnin=as.integer(burnin),
65                     mcmc=as.integer(mcmc), thin=as.integer(thin),
66                     betapropvar=beta.propvar,
67                     sigmapropvar=as.double(sigma.propvar),
68                     lecuyer=as.integer(lecuyer),
69                     seedarray=as.integer(seed.array),
70                     lecuyerstream=as.integer(lecuyer.stream),
71                     verbose=as.integer(verbose),
72                     betastart=beta.start, sigmastart=as.double(sigma.start),
73                     b0=b0, B0=B0, c0=as.double(c0),
74                     d0=as.double(d0), package="MCMCpack")
75
76    ## pull together matrix and build MCMC object to return
77    output <- form.mcmc.object(posterior,
78                               names=c(xnames, "sigma"),
79                               title="MCMCregress Posterior Density Sample")
80    return(output)
81
82  }
83
84
85  x <- rnorm(10)
86  y <- rnorm(10)
87  MCMClaplace(y~x)
88
```

File 5: MCMClaplace.startbuild.R

Lines 1 and 2 read in two files that contain **MCMCpack** helper functions. As noted in the body of this document, these functions help to automate many of the tasks that need to be done when constructing a MCMC sampling scheme in `R`.

Lines 5-82 contain a slightly modified definition of the `R` function `MCMClaplace()` that is called to do the model fitting. This function is almost exactly the final `R` function that is used for model fitting. The one exception is how `auto.Scythe.call()` is called in lines 60-74. This call to `auto.Scythe.call()` will be used to build template files. As such, `developer` has been set to `TRUE`, `help.file` has been set to `TRUE`, `cc.file` has been specified as `MCMClaplace.template.cc`, and `R.file` has been specified as `MCMClaplace.template.R`. The rest of the arguments are as they would be for a call to fit the model.

Finally, after the definition of `MCMClaplace()`, we generate some artificial data on lines 85 and 86 and then call `MCMClaplace()` with these data as arguments.

Sourcing `MCMClaplace.startbuild.R` into `R` will check the definition of `MCMClaplace()` for syntax errors and write three template files to disk. The following shows what is printed to the `R` console when `MCMClaplace.startbuild.R` is sourced into `R`.

```
> source("MCMClaplace.startbuild.R")
[1] "quantreg library loaded"
Created file named 'MCMClaplace.template.Rd'.
Edit the file and move it to the appropriate directory.

Created file named 'MCMClaplace.template.cc'.
Edit the file and move it to the appropriate directory.

Created file named 'MCMClaplace.template.R'.
Edit the file and move it to the appropriate directory.
Do not forget to edit the MCMCpack NAMESPACE file if
installing new functions as part of MCMCpack.

The call to .C is:
expression(posterior <- .C("MCMClaplace", sampledata = as.double(sample),
    samplerow = nrow(sample), samplecol = ncol(sample), Ydata = as.double(Y),
    Yrow = nrow(Y), Ycol = ncol(Y), Xdata = as.double(X), Xrow = nrow(X),
    Xcol = ncol(X), burnin = as.integer(burnin), mcmc = as.integer(mcmc),
    thin = as.integer(thin), betapropvardata = as.double(beta.propvar),
    betapropvarrow = nrow(beta.propvar), betapropvarcol = ncol(beta.propvar),
    sigmapropvar = as.double(sigma.propvar), lecuyer = as.integer(lecuyer),
    seedarray = as.integer(seed.array), lecuyerstream = as.integer(lecuyer.stream),
    verbose = as.integer(verbose), betastartdata = as.double(beta.start),
    betastartrow = nrow(beta.start), betastartcol = ncol(beta.start),
    sigmastart = as.double(sigma.start), b0data = as.double(b0),
    b0row = nrow(b0), b0col = ncol(b0), B0data = as.double(B0),
    B0row = nrow(B0), B0col = ncol(B0), c0 = as.double(c0), d0 = as.double(d0),
    PACKAGE = "MCMCpack"))

AUTOMATIC TEMPLATE FILE CREATION SUCCEEDED.
```

Here we see that a template `R` help file, a template `C++` file, and a template `R` file– named `MCMClaplace.template.Rd`, `MCMClaplace.template.cc`, and `MCMClaplace.template.R` respectively– have been created in the currently active directory. These files are shown below as Files 6, 7, and 8.

File 6: MCMClaplace.template.Rd

```
\name{MCMClaplace}
\alias{MCMClaplace}
%- Also NEED an '\alias' for EACH other topic documented here.
\title{ ~~function to do ... ~~ }
\description{
  ~~ A concise (1-5 lines) description of what the function does. ~~
}
\usage{
MCMClaplace(formula, data = parent.frame(), burnin = 1000, mcmc = 20000,
            thin = 1, beta.tune = 1.2, sigma.tune = 1, verbose = FALSE,
            seed = NA, beta.start = NA, sigma.start = NA, b0 = 0,
            B0 = 0, c0 = 1e-100, d0 = 1e+100, ...)
}
%- maybe also 'usage' for other objects documented here.
\arguments{
  \item{formula}{ ~~Describe \code{formula} here~~ }
  \item{data}{ ~~Describe \code{data} here~~ }
  \item{burnin}{ ~~Describe \code{burnin} here~~ }
  \item{mcmc}{ ~~Describe \code{mcmc} here~~ }
  \item{thin}{ ~~Describe \code{thin} here~~ }
  \item{beta.tune}{ ~~Describe \code{beta.tune} here~~ }
  \item{sigma.tune}{ ~~Describe \code{sigma.tune} here~~ }
  \item{verbose}{ ~~Describe \code{verbose} here~~ }
  \item{seed}{ ~~Describe \code{seed} here~~ }
  \item{beta.start}{ ~~Describe \code{beta.start} here~~ }
  \item{sigma.start}{ ~~Describe \code{sigma.start} here~~ }
  \item{b0}{ ~~Describe \code{b0} here~~ }
  \item{B0}{ ~~Describe \code{B0} here~~ }
  \item{c0}{ ~~Describe \code{c0} here~~ }
  \item{d0}{ ~~Describe \code{d0} here~~ }
  \item{\dots}{ ~~Describe \code{\dots} here~~ }
}
\details{
  ~~ If necessary, more details than the __description__  above ~~
}
\value{
  ~Describe the value returned
  If it is a LIST, use
  \item{comp1 }{Description of 'comp1'}
  \item{comp2 }{Description of 'comp2'}
  ...
}
\references{ ~put references to the literature/web site here ~ }
\author{ ~~who you are~~ }
\note{ ~~further notes~~ }

 ~Make other sections like Warning with \section{Warning }{....} ~

\seealso{ ~~objects to See Also as \code{\link{~~fun~~}}, ~~~ }
\examples{
##---- Should be DIRECTLY executable !! ----
##-- ==>  Define data, use random,
##--        or do  help(data=index)  for the standard data sets.

## The function is currently defined as
function(formula, data=parent.frame(),
                          burnin=1000, mcmc=20000, thin=1,
                          beta.tune=1.2, sigma.tune=1.0,
```

```
                          verbose=FALSE, seed=NA, beta.start=NA,
                          sigma.start=NA, b0=0, B0=0, c0=1e-100, d0=1e100,
                          ...){

  ## checks
  check.offset(list(...))
  check.mcmc.parameters(burnin, mcmc, thin)
  if (d0 < c0)
    stop("\n d0 < c0 in MCMClaplace(), respecify and call MCMClaplace again.\n")
  ## seeds
  seeds <- form.seeds(seed)
  lecuyer <- seeds[[1]]
  seed.array <- seeds[[2]]
  lecuyer.stream <- seeds[[3]]

  ## form response and model matrices
  holder <- parse.formula(formula, data)
  Y <- holder[[1]]
  X <- holder[[2]]
  xnames <- holder[[3]]
  K <- ncol(X)  # number of covariates

  ## fit L1 regression to get starting values and proposal parameters
  library(quantreg)
  rq.out <- rq(formula, data=data)

  ## starting values and priors
  if (is.na(beta.start)){
    beta.start <- coef(rq.out)
  }
  beta.start <- coef.start(beta.start, K, formula, family=gaussian, data)
  if (is.na(sigma.start)){
    sigma.start <- mad(rq.out$residuals)
  }
  if (sigma.start < c0 || sigma.start > d0)
    stop("\n sigma.start not in (c0, d0) in MCMClaplace(), respecify and call MCMClaplace again.\n")

  mvn.prior <- form.mvn.prior(b0, B0, K)
  b0 <- mvn.prior[[1]]
  B0 <- mvn.prior[[2]]

  ## setup proposal variances
  beta.tune <- vector.tune(beta.tune, K)
  sigma.tune <- scalar.tune(sigma.tune)
  beta.cov <- summary(rq.out, se="boot", covariance=TRUE)$cov
  beta.propvar <- beta.tune %*% beta.cov %*% beta.tune
  sigma.propvar <- sigma.tune^2


   ## define holder for posterior density sample
  sample <- matrix(data=0, mcmc/thin, K+1)

  auto.Scythe.call(output.object="posterior", cc.fun.name="MCMClaplace",
                   developer=TRUE, help.file=TRUE,
                   cc.file="MCMClaplace.template.cc",
                   R.file="MCMClaplace.template.R",
                   sample=sample, Y=Y, X=X, burnin=as.integer(burnin),
                   mcmc=as.integer(mcmc), thin=as.integer(thin),
                   betapropvar=beta.propvar,
```

```
                    sigmapropvar=as.double(sigma.propvar),
                    lecuyer=as.integer(lecuyer),
                    seedarray=as.integer(seed.array),
                    lecuyerstream=as.integer(lecuyer.stream),
                    verbose=as.integer(verbose),
                    betastart=beta.start, sigmastart=as.double(sigma.start),
                    b0=b0, B0=B0, c0=as.double(c0),
                    d0=as.double(d0), package="MCMCpack")

  ## pull together matrix and build MCMC object to return
  output <- form.mcmc.object(posterior,
                             names=c(xnames, "sigma"),
                             title="MCMCregress Posterior Density Sample")
  return(output)

  }
}
\keyword{ ~kwd1 }% at least one, from doc/KEYWORDS
\keyword{ ~kwd2 }% __ONLY ONE__ keyword per line
```
File 6: MCMClaplace.template.Rd

File 7: MCMClaplace.template.cc

```
// MCMClaplace.template.cc DESCRIPTION HERE
//
// The initial version of this file was generated by the
// auto.Scythe.call() function in the MCMCpack R package
// written by:
//
// Andrew D. Martin
// Dept. of Political Science
// Washington University in St. Louis
// admartin@wustl.edu
//
// Kevin M. Quinn
// Dept. of Government
// Harvard University
// kevin_quinn@harvard.edu
//
// This software is distributed under the terms of the GNU GENERAL
// PUBLIC LICENSE Version 2, June 1991.  See the package LICENSE
// file for more information.
//
// Copyright (C) 2004 Andrew D. Martin and Kevin M. Quinn
//
// This file was initially generated on Sat Jul 24 13:42:14 2004
// REVISION HISTORY

#include "matrix.h"
#include "distributions.h"
#include "stat.h"
#include "la.h"
#include "ide.h"
#include "smath.h"
#include "MCMCrng.h"
#include "MCMCfcds.h"

#include <R.h>           // needed to use Rprintf()
#include <R_ext/Utils.h> // needed to allow user interrupts

using namespace SCYTHE;
using namespace std;

extern "C" {

   // BRIEF FUNCTION DESCRIPTION
   void MCMClaplace(double *sampledata, const int *samplerow,
                    const int *samplecol, const double *Ydata,
                    const int *Yrow, const int *Ycol, const double *Xdata,
                    const int *Xrow, const int *Xcol, const int *burnin,
                    const int *mcmc, const int *thin,
                    const double *betapropvardata,
                    const int *betapropvarrow, const int *betapropvarcol,
                    const double *sigmapropvar, const int *lecuyer,
                    const int *seedarray, const int *lecuyerstream,
                    const int *verbose, const double *betastartdata,
                    const int *betastartrow, const int *betastartcol,
                    const double *sigmastart, const double *b0data,
                    const int *b0row, const int *b0col,
                    const double *B0data, const int *B0row,
                    const int *B0col, const double *c0, const double *d0) {
```

```
59
60      // pull together Matrix objects
61      // REMEMBER TO ACCESS PASSED ints AND doubles PROPERLY
62      Matrix <double> Y = r2scythe(*Yrow, *Ycol, Ydata);
63      Matrix <double> X = r2scythe(*Xrow, *Xcol, Xdata);
64      Matrix <double> betapropvar = r2scythe(*betapropvarrow, *betapropvarcol, betapropvardata);
65      Matrix <double> betastart = r2scythe(*betastartrow, *betastartcol, betastartdata);
66      Matrix <double> b0 = r2scythe(*b0row, *b0col, b0data);
67      Matrix <double> B0 = r2scythe(*B0row, *B0col, B0data);
68
69      // define constants
70      const int tot_iter = *burnin + *mcmc;  // total number of mcmc iterations
71      const int nstore = *mcmc / *thin;      // number of draws to store
72      const int NUMBER_OF_PARAMETERS = ????; // YOU NEED TO FILL THIS IN
73
74      // storage matrix or matrices
75      Matrix<double> STORAGEMATRIX(nstore, NUMBER_OF_PARAMETERS);
76
77      // initialize rng stream
78      rng *stream = MCMCpack_get_rng(*lecuyer, seedarray, *lecuyerstream);
79
80      // set starting values
81      PARAMETER_BLOCK1 = ????;
82      PARAMETER_BLOCK2 = ????;
83      ETC.;
84
85      ///// MCMC SAMPLING OCCURS IN THIS FOR LOOP
86      for(int iter = 0; iter < tot_iter; ++iter){
87
88        // sample the parameters
89        PARAMETER_BLOCK1 = ????;
90        PARAMETER_BLOCK2 = ????;
91        ETC;
92
93        // store draws in storage matrix (or matrices)
94        if(iter >= *burnin && (iter % *thin == 0)){
95         // PUT DRAWS IN STORAGEMATRIX HERE
96        }
97
98        // print output to stdout
99        if(*verbose == 1 && iter % 500 == 0){
100         Rprintf("\n\nMCMClaplace iteration %i of %i \n", (iter+1), tot_iter);
101         // ADD ADDITIONAL OUTPUT HERE IF DESIRED
102       }
103
104       void R_CheckUserInterrupt(void); // allow user interrupts
105     } // end MCMC loop
106
107     delete stream; // clean up random number stream
108
109     // load draws into sample array
110     const int size = *samplerow * *samplecol;
111     for(int i = 0; i < size; ++i)
112       sampledata[i] = STORAGEMATRIX[i];
113
114   } // end MCMClaplace
115 } // end extern "C"
```

File 7: MCMClaplace.template.cc

File 8: MCMClaplace.template.R

```
"MCMClaplace" <-
function(formula, data=parent.frame(),
                          burnin=1000, mcmc=20000, thin=1,
                          beta.tune=1.2, sigma.tune=1.0,
                          verbose=FALSE, seed=NA, beta.start=NA,
                          sigma.start=NA, b0=0, B0=0, c0=1e-100, d0=1e100,
                          ...){

  ## checks
  check.offset(list(...))
  check.mcmc.parameters(burnin, mcmc, thin)
  if (d0 < c0)
    stop("\n d0 < c0 in MCMClaplace(), respecify and call MCMClaplace again.\n")
  ## seeds
  seeds <- form.seeds(seed)
  lecuyer <- seeds[[1]]
  seed.array <- seeds[[2]]
  lecuyer.stream <- seeds[[3]]

  ## form response and model matrices
  holder <- parse.formula(formula, data)
  Y <- holder[[1]]
  X <- holder[[2]]
  xnames <- holder[[3]]
  K <- ncol(X)  # number of covariates

  ## fit L1 regression to get starting values and proposal parameters
  library(quantreg)
  rq.out <- rq(formula, data=data)

  ## starting values and priors
  if (is.na(beta.start)){
    beta.start <- coef(rq.out)
  }
  beta.start <- coef.start(beta.start, K, formula, family=gaussian, data)
  if (is.na(sigma.start)){
    sigma.start <- mad(rq.out$residuals)
  }
  if (sigma.start < c0 || sigma.start > d0)
    stop("\n sigma.start not in (c0, d0) in MCMClaplace(), respecify and call MCMClaplace again.\n")

  mvn.prior <- form.mvn.prior(b0, B0, K)
  b0 <- mvn.prior[[1]]
  B0 <- mvn.prior[[2]]

  ## setup proposal variances
  beta.tune <- vector.tune(beta.tune, K)
  sigma.tune <- scalar.tune(sigma.tune)
  beta.cov <- summary(rq.out, se="boot", covariance=TRUE)$cov
  beta.propvar <- beta.tune %*% beta.cov %*% beta.tune
  sigma.propvar <- sigma.tune^2


   ## define holder for posterior density sample
  sample <- matrix(data=0, mcmc/thin, K+1)

  auto.Scythe.call(output.object="posterior", cc.fun.name="MCMClaplace",
                   developer=TRUE, help.file=TRUE,
```

```
                    cc.file="MCMClaplace.template.cc",
                    R.file="MCMClaplace.template.R",
                    sample=sample, Y=Y, X=X, burnin=as.integer(burnin),
                    mcmc=as.integer(mcmc), thin=as.integer(thin),
                    betapropvar=beta.propvar,
                    sigmapropvar=as.double(sigma.propvar),
                    lecuyer=as.integer(lecuyer),
                    seedarray=as.integer(seed.array),
                    lecuyerstream=as.integer(lecuyer.stream),
                    verbose=as.integer(verbose),
                    betastart=beta.start, sigmastart=as.double(sigma.start),
                    b0=b0, B0=B0, c0=as.double(c0),
                    d0=as.double(d0), package="MCMCpack")

  ## pull together matrix and build MCMC object to return
  output <- form.mcmc.object(posterior,
                             names=c(xnames, "sigma"),
                             title="MCMCregress Posterior Density Sample")
  return(output)

}
```

File 8: MCMClaplace.template.R

Looking first at `MCMClaplace.template.R` we see that this is very close to a finished version that can be used for model fitting. All that needs to be done is some additional comment writing and the deletion of the `developer`, `help.file`, `cc.file`, and `R.file` arguments to the call to `auto.Scythe.call()`. Doing this and saving the resulting file as `MCMClaplace.R` produces the file in File 9.

File 9: MCMClaplace.R

```
## MCMClaplace returns a sample from the posterior for a linear regression
## with Laplace disturbances. It assumes a multivariate normal prior
## for the coefficient vector and a uniform(c0,d0) prior for sigma.
##
## Kevin Quinn
## Dept. of Government and CBRSS
## Harvard University
## kevin_quinn@harvard.edu
##
## 7/24/2004
##

"MCMClaplace" <-
function(formula, data=parent.frame(),
                          burnin=1000, mcmc=20000, thin=1,
                          beta.tune=1.2, sigma.tune=1.0,
                          verbose=FALSE, seed=NA, beta.start=NA,
                          sigma.start=NA, b0=0, B0=0, c0=1e-100, d0=1e100,
                          ...){

  ## checks
  check.offset(list(...))
  check.mcmc.parameters(burnin, mcmc, thin)
  if (d0 < c0)
    stop("\n d0 < c0 in MCMClaplace(), respecify and call MCMClaplace again.\n")
  ## seeds
  seeds <- form.seeds(seed)
  lecuyer <- seeds[[1]]
  seed.array <- seeds[[2]]
  lecuyer.stream <- seeds[[3]]

  ## form response and model matrices
  holder <- parse.formula(formula, data)
  Y <- holder[[1]]
  X <- holder[[2]]
  xnames <- holder[[3]]
  K <- ncol(X)  # number of covariates

  ## fit L1 regression to get starting values and proposal parameters
  library(quantreg)
  rq.out <- rq(formula, data=data)

  ## starting values and priors
  if (is.na(beta.start)){
    beta.start <- coef(rq.out)
  }
  beta.start <- coef.start(beta.start, K, formula, family=gaussian, data)
  if (is.na(sigma.start)){
    sigma.start <- mad(rq.out$residuals)
  }
  if (sigma.start < c0 || sigma.start > d0)
    stop("\n sigma.start not in (c0, d0) in MCMClaplace(), respecify and call MCMClaplace again.\n")

  mvn.prior <- form.mvn.prior(b0, B0, K)
  b0 <- mvn.prior[[1]]
  B0 <- mvn.prior[[2]]

  ## setup proposal variances
```

```
59    beta.tune <- vector.tune(beta.tune, K)
60    sigma.tune <- scalar.tune(sigma.tune)
61    beta.cov <- summary(rq.out, se="boot", covariance=TRUE)$cov
62    beta.propvar <- beta.tune %*% beta.cov %*% beta.tune
63    sigma.propvar <- sigma.tune^2
64
65
66     ## define holder for posterior density sample
67    sample <- matrix(data=0, mcmc/thin, K+1)
68
69    auto.Scythe.call(output.object="posterior", cc.fun.name="MCMClaplace",
70                     sample=sample, Y=Y, X=X, burnin=as.integer(burnin),
71                     mcmc=as.integer(mcmc), thin=as.integer(thin),
72                     betapropvar=beta.propvar,
73                     sigmapropvar=as.double(sigma.propvar),
74                     lecuyer=as.integer(lecuyer),
75                     seedarray=as.integer(seed.array),
76                     lecuyerstream=as.integer(lecuyer.stream),
77                     verbose=as.integer(verbose),
78                     betastart=beta.start, sigmastart=as.double(sigma.start),
79                     b0=b0, B0=B0, c0=as.double(c0),
80                     d0=as.double(d0), package="MCMCpack")
81
82    ## pull together matrix and build MCMC object to return
83    output <- form.mcmc.object(posterior,
84                               names=c(xnames, "sigma"),
85                               title="MCMCregress Posterior Density Sample")
86    return(output)
87
88  }
```

File 9: MCMClaplace.R

Turning our attention to `MCMClaplace.template.cc` we see that `auto.Scythe.call()` has done much of the mundane work of `#include`ing necessary header files and creating the general structure of a program file that can be used to fit the model of interest. Nonetheless, some additional work is necessary on our part.

First of all, we need to write a `C++` function that will evaluate the log-posterior of the Laplace regression model. This has been done on lines 43-66 of `MCMClaplace.cc` (File 10). We also need to define a few additional constants (lines 100-102 of `MCMClaplace.cc`), set starting values (lines 111-113) of `MCMClaplace.cc`), define the sampling scheme (lines 122-142) of `MCMClaplace.cc`), write the code to store the draws (lines 144-150) of `MCMClaplace.cc`), allow for output to the console (lines 152-165 of `MCMClaplace.cc`), put the sampled parameter values back in `sampledata`, and print information about the Metropolis sampling (lines 172-182 of `MCMClaplace.cc`).

File 10: MCMClaplace.cc

```
// MCMClaplace.template.cc DESCRIPTION HERE
//
// The initial version of this file was generated by the
// auto.Scythe.call() function in the MCMCpack R package
// written by:
//
// Andrew D. Martin
// Dept. of Political Science
// Washington University in St. Louis
// admartin@wustl.edu
//
// Kevin M. Quinn
// Dept. of Government
// Harvard University
// kevin_quinn@harvard.edu
//
// This software is distributed under the terms of the GNU GENERAL
// PUBLIC LICENSE Version 2, June 1991.  See the package LICENSE
// file for more information.
//
// Copyright (C) 2004 Andrew D. Martin and Kevin M. Quinn
//
// This file was initially generated on Sat Jul 24 13:24:09 2004
// cleaned up and finished off by Kevin Quinn 7/24/2004
//

#include "matrix.h"
#include "distributions.h"
#include "stat.h"
#include "la.h"
#include "ide.h"
#include "smath.h"
#include "MCMCrng.h"
#include "MCMCfcds.h"

#include <R.h>           // needed to use Rprintf()
#include <R_ext/Utils.h> // needed to allow user interrupts

using namespace SCYTHE;
using namespace std;

// the Laplace regression log posterior
double laplace_logpost(Matrix<double>& Y, const Matrix<double>& X,
                       const Matrix<double>& beta,
                       const double& sigma,
                       const Matrix<double>& beta_prior_mean,
                       const Matrix<double>& beta_prior_prec,
                       const double& c0, const double& d0){

  // likelihood
  Matrix<double> yhat = X * beta;
  double loglike = 0;
  for (int i=0; i<Y.rows(); ++i) {
    loglike += -::log(2.0 * sigma) - ( ::fabs(yhat[i] - Y[i]) / sigma);
  }

  // prior
  double logprior =  ::log(0.0);
```

```
  if (sigma > c0 && sigma < d0)
    logprior = ::log(1/(d0-c0));
  if (beta_prior_prec(0,0) != 0){
    logprior += lndmvn(beta, beta_prior_mean, invpd(beta_prior_prec));
  }

  return (loglike + logprior);
}

extern "C" {

   // BRIEF FUNCTION DESCRIPTION
  void MCMClaplace(double *sampledata, const int *samplerow,
                   const int *samplecol, const double *Ydata,
                   const int *Yrow, const int *Ycol, const double *Xdata,
                   const int *Xrow, const int *Xcol, const int *burnin,
                   const int *mcmc, const int *thin,
                   const double *betapropvardata,
                   const int *betapropvarrow, const int *betapropvarcol,
                   const double *sigmapropvar, const int *lecuyer,
                   const int *seedarray, const int *lecuyerstream,
                   const int *verbose, const double *betastartdata,
                   const int *betastartrow, const int *betastartcol,
                   const double *sigmastart, const double *b0data,
                   const int *b0row, const int *b0col,
                   const double *B0data, const int *B0row,
                   const int *B0col, const double *c0, const double *d0) {

    // pull together Matrix objects
    Matrix <double> Y = r2scythe(*Yrow, *Ycol, Ydata);
    Matrix <double> X = r2scythe(*Xrow, *Xcol, Xdata);
    Matrix <double> betapropvar = r2scythe(*betapropvarrow,
                                           *betapropvarcol, betapropvardata);
    Matrix <double> betastart = r2scythe(*betastartrow,
                                         *betastartcol, betastartdata);
    Matrix <double> b0 = r2scythe(*b0row, *b0col, b0data);
    Matrix <double> B0 = r2scythe(*B0row, *B0col, B0data);

    // define constants
    const int tot_iter = *burnin + *mcmc;  // total number of mcmc iterations
    const int nstore = *mcmc / *thin;      // number of draws to store
    const int k = X.cols(); // number of coefficients
    const Matrix <double> propC  = cholesky(betapropvar);
    const double sigmapropSD = ::sqrt(*sigmapropvar);

    // storage matrix or matrices
    Matrix<double> storemat(nstore,k+1);

    // initialize rng stream
    rng *stream = MCMCpack_get_rng(*lecuyer, seedarray, *lecuyerstream);

    // set starting values
    Matrix <double> beta = betastart;
    double sigma = *sigmastart;
    double logpost_cur =  laplace_logpost(Y,X,beta, sigma,
                                          b0, B0, *c0, *d0);

    int count = 0;
    int accepts_b = 0;
```

```
118       int accepts_s = 0;
119       ///// MCMC SAMPLING OCCURS IN THIS FOR LOOP
120       for(int iter = 0; iter < tot_iter; ++iter){
121
122         // sample beta
123         Matrix<double> beta_can = beta + propC * stream->rnorm(k,1);
124         double logpost_can = laplace_logpost(Y,X,beta_can, sigma,
125                                              b0, B0, *c0, *d0);
126         double ratio = ::exp(logpost_can - logpost_cur);
127         if (stream->runif() < ratio){
128           beta = beta_can;
129           logpost_cur = logpost_can;
130           ++accepts_b;
131         }
132
133         // sample sigma
134         double sigma_can = sigma + stream->rnorm(0.0, sigmapropSD);
135         logpost_can = laplace_logpost(Y,X,beta, sigma_can,
136                                       b0, B0, *c0, *d0);
137         ratio = ::exp(logpost_can - logpost_cur);
138         if (stream->runif() < ratio){
139           sigma = sigma_can;
140           logpost_cur = logpost_can;
141           ++accepts_s;
142         }
143
144         // store draws in storage matrix (or matrices)
145         if(iter >= *burnin && (iter % *thin == 0)){
146           for (int j=0; j<k; ++j)
147             storemat(count,j) = beta[j];
148           storemat(count,k) = sigma;
149           ++count;
150         }
151
152         // print output to stdout
153         if(*verbose == 1 && iter % 500 == 0){
154           Rprintf("\n\nMCMClaplace iteration %i of %i \n", (iter+1), tot_iter);
155           Rprintf("beta = \n");
156           for (int j=0; j<k; ++j)
157             Rprintf("%10.5f\n", beta[j]);
158           Rprintf("sigma = %10.5f\n", sigma);
159           Rprintf("Metropolis acceptance rate for beta = %3.5f\n\n",
160                   static_cast<double>(accepts_b) /
161                   static_cast<double>(iter+1));
162           Rprintf("Metropolis acceptance rate for sigma = %3.5f\n\n",
163                   static_cast<double>(accepts_s) /
164                   static_cast<double>(iter+1));
165         }
166
167         void R_CheckUserInterrupt(void); // allow user interrupts
168       } // end MCMC loop
169
170       delete stream; // clean up random number stream
171
172       // load draws into sample array
173       const int size = *samplerow * *samplecol;
174       for(int i = 0; i < size; ++i)
175         sampledata[i] = storemat[i];
176
```

```
    Rprintf("\n\n@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@\n");
    Rprintf("The Metropolis acceptance rate for beta was %3.5f",
            static_cast<double>(accepts_b) / static_cast<double>(tot_iter));
    Rprintf("\nThe Metropolis acceptance rate for sigma was %3.5f",
            static_cast<double>(accepts_s) / static_cast<double>(tot_iter));
    Rprintf("\n@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@\n");

  } // end MCMClaplace
} // end extern "C"
```
File 10: MCMClaplace.cc

At this point we could simply move `MCMClaplace.R` to `MCMCpack/R/` and `MCMClaplace.cc` to `MCMCpack/src/`, add `MCMClaplace` to the **MCMCpack** `NAMESPACE`, and then issue the install from source command:

```
R CMD INSTALL MCMCpack
```

This is to be issued from a shell prompt in the directory in which that the `MCMCpack` directory is located to install a version of **MCMCpack** that includes a working version of our own custom-built Laplace regression model. This will allow one to fit the Laplace regression model using MCMC but there will be no help file for `MCMClaplace`.

To produce a help file for our new function we need to edit and fill in `MCMClaplace.template.Rd`, rename the new file `MCMClaplace.Rd`, move this file to `MCMCpack/man/`, and re-install from source. A rough cut at the filled in version of `MCMClaplace.Rd` appears in File 11 below.

File 11: MCMClaplace.Rd

```
\name{MCMClaplace}
\alias{MCMClaplace}
\title{Markov Chain Monte Carlo for Laplace Regression}
\description{
  This function generates a posterior density sample
  from a linear regression model with Laplace errors using
  Metropolis sampling (with a multivariate Gaussian prior on the
  beta vector, and a proper uniform prior on the Laplace scale
  parameter). The user supplies data and priors, and
  a sample from the posterior density is returned as an mcmc
  object, which can be subsequently analyzed with functions
  provided in the coda package.
}
\usage{
MCMClaplace(formula, data = list(), burnin = 1000, mcmc = 20000,
            thin = 1, beta.tune = 1.2, sigma.tune = 1, verbose = FALSE,
            seed = NA, beta.start = NA, sigma.start = NA, b0 = 0, B0 = 0,
            c0 = 1e-100, d0 = 1e+100, ...)
}
\arguments{
  \item{formula}{Model formula.}
  \item{data}{Data frame.}
  \item{burnin}{The number of burn-in iterations for the sampler.}
  \item{mcmc}{The number of MCMC iterations after burnin.}
  \item{thin}{The thinning interval used in the simulation.  The number of
    MCMC iterations must be divisible by this value.}
  \item{beta.tune}{Tuning parameter for the Metropolis sampling of
    \eqn{\beta}{beta}. Can be either a positive scalar or a
    \eqn{k}{k}-vector, where \eqn{k}{k} is the length of
    \eqn{\beta}{beta}.}
  \item{sigma.tune}{Tuning parameter for the Metropolis sampling of
    \eqn{\sigma}{sigma}. Must be a positive scalar.}
  \item{verbose}{A switch which determines whether or not the progress of
    the sampler is printed to the screen.  If TRUE, the iteration number, the
    \eqn{\beta}{beta} vector, \eqn{\sigma}{\sigma}, and the Metropolis
    acceptance rates are printed to the screen every 500 iterations. }
  \item{seed}{The seed for the random number generator.  If NA, the Mersenne
    Twister generator is used with default seed 12345; if an integer is
    passed it is used to seed the Mersenne twister.  The user can also
    pass a list of length two to use the L'Ecuyer random number generator,
    which is suitable for parallel computation.  The first element of the
    list is the L'Ecuyer seed, which is a vector of length six or NA (if NA
    a default seed of \code{rep(12345,6)} is used).  The second element of
    list is a positive substream number. See the MCMCpack
    specification for more details.}
  \item{beta.start}{The starting values for the \eqn{\beta}{beta} vector.
    This can either be a scalar or a
    column vector with dimension equal to the number of betas.
    The default value of of NA will use the L1 regression
    estimate of \eqn{\beta}{beta} as the starting value.  If this is a
    scalar,  that value will serve as the starting value
    mean for all of the betas.}
  \item{sigma.start}{The starting value for \eqn{\sigma}{sigma}. Must be
  a positive scalar or NA. A value of NA will use MAD of the L1
  regression residuals as the starting value.}
  \item{b0}{The prior mean of \eqn{\beta}{beta}.  This can either be a
    scalar or a column vector with dimension equal to the number of
    betas. If this takes a scalar  value, then that value will serve as
```

```
    the prior mean for all of the betas. }
  \item{B0}{The prior precision of \eqn{\beta}{beta}.  This can either be a
    scalar or a square matrix with dimensions equal to the number of betas.
    If this takes a scalar value, then that value times an identity
    matrix serves as the prior precision of beta. Default value of 0 is
    equivalent to an improper uniform prior for beta.}
  \item{c0}{Lower endpoint of the uniform prior for \eqn{\sigma}{sigma}.}
  \item{d0}{Upper endpoint of the uniform prior for \eqn{\sigma}{sigma}.}
  \item{\dots}{further arguments to be passed}
}
\details{
  \code{MCMClaplace} simulates from the posterior density using
  two Metropolis steps.  The simulation proper is done in compiled C++
  code to maximize efficiency.  Please consult
  the coda documentation for a comprehensive list of functions that can be
  used to analyze the posterior density sample.

  The model takes the following form:
  \deqn{y_i = x_i ' \beta + \varepsilon_{i}}{y_i = x_i'beta + epsilon_i}
  Where the errors are assumed to follow a Laplace distribution:
  \deqn{\varepsilon_{i} \sim \mathcal{L}aplace(0, \sigma)}{epsilon_i
    ~ Laplace(0, sigma)}
  We assume the following priors:
  \deqn{\beta \sim \mathcal{N}(b_0,B_0^{-1})}{beta ~ N(b0,B0^(-1))}
  And:
  \deqn{\sigma \sim \mathcal{U}nif(c_0, d_0)}{sigma ~
    Unif(c0, d0)}
  Where \eqn{\beta}{beta} and \eqn{\sigma}{sigma} are assumed
  \emph{a priori} independent.
}
\value{
   An mcmc object that contains the posterior density sample.  This
   object can be summarized by functions provided by the coda package.
}
\references{
   Andrew D. Martin, Kevin M. Quinn, and Daniel Pemstein.  2004.
   \emph{Scythe Statistical Library 1.0.} \url{http://scythe.wustl.edu}.

   Martyn Plummer, Nicky Best, Kate Cowles, and Karen Vines. 2002.
   \emph{Output Analysis and Diagnostics for MCMC (CODA)}.
   \url{http://www-fis.iarc.fr/coda/}.
}
\author{ Kevin Quinn }
\seealso{\code{\link[coda]{plot.mcmc}},
  \code{\link[coda]{summary.mcmc}}, \code{\link[quantreg]{rq}
}
\examples{
\dontrun{
data(mtcars)
posterior <- MCMClaplace(mpg~cyl+disp+hp+wt+qsec+am+gear, data=mtcars,
                    beta.tune=0.5, sigma.tune=1.4,
                    verbose=TRUE, mcmc=500000, thin=20)
plot(posterior)
raftery.diag(posterior)
summary(posterior)

  }
}
\keyword{models}
```

118 File 11: MCMClaplace.Rd

Compiling this file with LaTeX we get the following:

`MCMClaplace` *Markov Chain Monte Carlo for Laplace Regression*

Description

This function generates a posterior density sample from a linear regression model with Laplace errors using Metropolis sampling (with a multivariate Gaussian prior on the beta vector, and a proper uniform prior on the Laplace scale parameter). The user supplies data and priors, and a sample from the posterior density is returned as an mcmc object, which can be subsequently analyzed with functions provided in the coda package.

Usage

```
MCMClaplace(formula, data = list(), burnin = 1000, mcmc = 20000,
            thin = 1, beta.tune = 1.2, sigma.tune = 1, verbose = FALSE,
            seed = NA, beta.start = NA, sigma.start = NA, b0 = 0, B0 = 0,
            c0 = 1e-100, d0 = 1e+100, ...)
```

Arguments

`formula`	Model formula.
`data`	Data frame.
`burnin`	The number of burn-in iterations for the sampler.
`mcmc`	The number of MCMC iterations after burnin.
`thin`	The thinning interval used in the simulation. The number of MCMC iterations must be divisible by this value.
`beta.tune`	Tuning parameter for the Metropolis sampling of β. Can be either a positive scalar or a k-vector, where k is the length of β.
`sigma.tune`	Tuning parameter for the Metropolis sampling of σ. Must be a positive scalar.
`verbose`	A switch which determines whether or not the progress of the sampler is printed to the screen. If TRUE, the iteration number, the β vector, σ, and the Metropolis acceptance rates are printed to the screen every 500 iterations.

`seed` The seed for the random number generator. If NA, the Mersenne Twister generator is used with default seed 12345; if an integer is passed it is used to seed the Mersenne twister. The user can also pass a list of length two to use the L'Ecuyer random number generator, which is suitable for parallel computation. The first element of the list is the L'Ecuyer seed, which is a vector of length six or NA (if NA a default seed of `rep(12345,6)` is used). The second element of list is a positive substream number. See the MCMCpack specification for more details.

`beta.start` The starting values for the β vector. This can either be a scalar or a column vector with dimension equal to the number of betas. The default value of of NA will use the L1 regression estimate of β as the starting value. If this is a scalar, that value will serve as the starting value mean for all of the betas.

`sigma.start` The starting value for σ. Must be a positive scalar or NA. A value of NA will use MAD of the L1 regression residuals as the starting value.

`b0` The prior mean of β. This can either be a scalar or a column vector with dimension equal to the number of betas. If this takes a scalar value, then that value will serve as the prior mean for all of the betas.

`B0` The prior precision of β. This can either be a scalar or a square matrix with dimensions equal to the number of betas. If this takes a scalar value, then that value times an identity matrix serves as the prior precision of beta. Default value of 0 is equivalent to an improper uniform prior for beta.

`c0` Lower endpoint of the uniform prior for σ.

`d0` Upper endpoint of the uniform prior for σ.

`...` further arguments to be passed

Details

`MCMClaplace` simulates from the posterior density using two Metropolis steps. The simulation proper is done in compiled C++ code to maximize efficiency. Please consult the coda documentation for a comprehensive list of functions that can be used to analyze the posterior density sample.

The model takes the following form:

$$y_i = x_i'\beta + \varepsilon_i$$

Where the errors are assumed to follow a Laplace distribution:

$$\varepsilon_i \sim \mathcal{L}aplace(0, \sigma)$$

We assume the following priors:

$$\beta \sim \mathcal{N}(b_0, B_0^{-1})$$

And:

$$\sigma \sim \mathcal{U}nif(c_0, d_0)$$

Where β and σ are assumed *a priori* independent.

Value

An mcmc object that contains the posterior density sample. This object can be summarized by functions provided by the coda package.

Author(s)

Kevin Quinn

References

Andrew D. Martin, Kevin M. Quinn, and Daniel Pemstein. 2004. *Scythe Statistical Library 1.0.* `http://scythe.wustl.edu`.

Martyn Plummer, Nicky Best, Kate Cowles, and Karen Vines. 2002. *Output Analysis and Diagnostics for MCMC (CODA).* `http://www-fis.iarc.fr/coda/`.

Examples

```
## Not run:
data(mtcars)
posterior <- MCMClaplace(mpg~cyl+disp+hp+wt+qsec+am+gear, data=mtcars,
                    beta.tune=0.5, sigma.tune=1.4,
                    verbose=TRUE, mcmc=500000, thin=20)
plot(posterior)
raftery.diag(posterior)
summary(posterior)

  ## End(Not run)
```

8 Submitting Additions for Inclusion in Future MCMCpack Releases

Users who wish to have their code formally included in future **MCMCpack** releases should send their code, along with examples of how to use their code, to the main **MCMCpack** developers: Andrew Martin (`admartin@wustl.edu`) and Kevin Quinn (`kevin_quinn@harvard.edu`). All submissions should be in electronic form and should conform to the coding guidelines spelled out in this document.

Responsibility for ensuring that the user-written functions work properly across major operating systems and system architectures belongs to the original writer of the code. Similarly, the author is responsible for all documentation of contributed code.

The writer of source code that is used within **MCMCpack** will be credited in the comments of that source code and, where possible, in the **author** field of the associated `R` help file. Our preference is that all code in **MCMCpack** is licensed under the GNU GPL 2 or greater. We will include code under other licenses with disclaimers (such as our inclusion of L'Ecuyer's RngStreams under a restrictive license).

The main **MCMCpack** developers will try to include user-written code in future releases but they reserve the right to not include contributed code without cause.

A MCMCpack Hidden Developer Functions

`agree.mat` *Calculate Agreement Score Matrix*

Description

agree.mat calculates an agreement score matrix from a vote matrix.

Usage

```
agree.mat(X)
```

Arguments

`X`	A matrix of roll call votes. Voters are assumed to be on the rows and the items being voted on are assumed to be on the columns.

Value

A subject by subject matrix of agreement scores.

`auto.Scythe.call` *Automated C++ Scythe Call and Program Template*

Description

This function automates the call to the Scythe C++ program, making book-keeping much easier. It wraps the .C function to pass integers, doubles, and matrices to the compiled code in model mode. When used in developer mode, it also produces template C++ and R programs which makes writing Scythe code much easier.

Usage

```
auto.Scythe.call(output.object, cc.fun.name, package = "MCMCpack",
   developer = FALSE, help.file=FALSE, cc.file="", R.file="", ...)
```

Arguments

`output.object`	The name of the posterior density sample that will be placed in the parent environment upon completion (string).
`cc.fun.name`	The name of the C++ shared library (string).
`package`	The package name, by default MCMCpack (string).
`developer`	If TRUE, use in developer mode. If FALSE, in model mode.
`help.file`	If TRUE, write a template help file in Rd format.
`cc.file`	The file to write the C++ template if in developer mode. If default `""`, the template is printed to the screen.
`R.file`	The file to write the R template if in developer mode. If default `""`, the template is printed to the screen.
`...`	The integers, doubles, and matrices to be passed to Scythe.

Details

See Section 7 of the **MCMCpack** Specification and Developer Documentation for more details and a full example of how this function can be used be developers. In the example below, we illustrate how the function is used when used in model mode in a completed model function.

Value

0 if the call is performed cleanly.

References

Andrew D. Martin, Kevin M. Quinn, and Daniel Pemstein. 2004. *Scythe Statistical Library 1.0.* `http://scythe.wustl.edu`.

Examples

```
## Not run:
    auto.Scythe.call(output.object="posterior", cc.fun.name="MCMCregress",
                     sample=sample, Y=Y, X=X, burnin=as.integer(burnin),
```

```
                    mcmc=as.integer(mcmc), thin=as.integer(thin),
                    lecuyer=as.integer(lecuyer),
                    seedarray=as.integer(seed.array),
                    lecuyerstream=as.integer(lecuyer.stream),
                    verbose=as.integer(verbose), betastart=beta.start,
                    b0=b0, B0=B0, c0=as.double(c0), d0=as.double(d0))
## End(Not run)
```

`build.factor.constraints`

Build inequality and equality constraint matrices for a factor loading matrix.

Description

Build inequality and equality constraint matrices for a factor loading matrix. Can be (and is in MCMCpack) used to build more constraint matrices of other types.

Usage

```
build.factor.constraints(lambda.constraints, X, K, factors)
```

Arguments

`lambda.constraints`	List of lists specifying possible simple equality or inequality constraints on the factor loadings. A typical entry in the list has one of three forms: `varname=list(d,c)` which will constrain the dth loading for the variable named `varname` to be equal to c, `varname=list(d,"+")` which will constrain the dth loading for the variable named `varname` to be positive, and `varname=list(d, "-")` which will constrain the dth loading for the variable named `varname` to be negative. If x is a matrix without column names defaults names of "V1","V2", ... , etc will be used.
`X`	data matrix
`K`	number of manifest variables
`factors`	number of factors

Value

A list composed of 3 items:

`Lambda.eq.constraints`
: matrix the same dimension as the factor loading matrix that has elements equal to -999 if the corresponding element of Lambda is not constrained to a constant and the constant value if it is constrained to a constant.

`Lambda.ineq.constraints`
: matrix the same dimension as the factor loading matrix that has elements equal to 0 if there are no inequality constraints on that element of Lambda and either -1 or +1 if that element of Lambda is constrained to be either negative or positive respectively.

...

`calling.function` *Return Name of MCMCpack Calling Function*

Description

Returns the name of the model calling function, e.g., MCMCregress, which is used for echoing error messages.

Usage

```
calling.function()
```

Value

The name of the calling function as a string.

`check.ig.prior` *Check Inverse Gamma Prior*

Description

Checks the parameters of an inverse Gamma prior.

Usage

```
check.ig.prior(c0, d0)
```

Arguments

`c0`	Shape parameter for inverse Gamma prior.
`d0`	Scale parameter for inverse Gamma prior.

Value

0 if tests pass, otherwise stop with an error message.

`check.mcmc.parameters` *Check MCMC Parameters*

Description

Checks burnin, mcmc, and thinning parameters to make sure they are positive and conformable.

Usage

```
check.mcmc.parameters(burnin, mcmc, thin)
```

Arguments

`burnin`	Number of burnin iterations.
`mcmc`	Number of mcmc iterations.
`thin`	Thinning interval.

Value

0 if tests pass, otherwise stop with an error message.

`check.offset` *Check For Offset*

Description

Checks whether the user specifies an offset in the model formula.

Usage

```
check.offset(args)
```

Arguments

`args`	Arguments in the original function call.

Details

Currently **MCMCpack** does not support offsets. This function checks to see if one is specified, and if so, echos a suitable error message.

Value

0 if tests pass, otherwise stop with an error message.

`coef.start` *Starting Values for Coefficients of a GLM*

Description

Creates starting values for GLM coefficients from user input. If defaults are passed GLM estimates are used.

Usage

```
coef.start(beta.start, K, formula, family, data, defaults)
```

Arguments

`beta.start`	The parameter from the **MCMCpack** model function. If NA, the GLM estimate is used. If a scalar, then that value is used for all betas. If a vector, the dimensionality is checked.
`K`	The dimensionality of the vector.
`formula`	Model formula used for the GLM.
`family`	Specific GLM to use.
`data`	Data used to fit the GLM.
`defaults`	If a value is passed, the starting values are created. If a vector is passed, it is used. If a scalar is passed, that value will serve as the starting value for all of the betas.

Value

A vector of dimension $(K \times 1)$.

`factload.start` *Checks and expands starting values for factor loading matrix.*

Description

factload.start takes user input and checks it for appropriateness and then builds a matrix that can be used as an initial factor loading matrix.

Usage

```
factload.start(lambda.start, K, factors, Lambda.eq.constraints, Lambda.ineq.constraints)
```

Arguments

`lambda.start`	Starting values for the factor loading matrix Lambda. If `lambda.start` is set to a scalar the starting value for all unconstrained loadings will be set to that scalar. If `lambda.start` is a matrix of the same dimensions as Lambda then the `lambda.start` matrix is used as the starting values (except for equality-constrained elements). If `lambda.start` is set to `NA` (the default) then starting values for unconstrained elements are set to 0, and starting values for inequality constrained elements are set to either 0.5 or -0.5 depending on the nature of the constraints
`K`	number of manifest variables
`factors`	number of factors
`Lambda.eq.constraints`	matrix the same dimension as the factor loading matrix that has elements equal to -999 if the correseponding element of Lambda is not constrained to a constant and the constant value if it is constrained to a constant.
`Lambda.ineq.constraints`	matrix the same dimension as the factor loading matrix that has elements equal to 0 if there are no inequality constraints on that element of Lambda and either -1 or +1 if that element of Lambda is constrained to be either negative or positive respectively

Value

A `K` by `factors` matrix.

`factor.score.eigen.start` *starting values for factor scores and ability parameters.*

Description

Uses an eigenvalue-eigenvector decomposition of a subject by subject agreement score matrix to arrive at starting values for factor score and ability parameters.

Usage

```
factor.score.eigen.start(A, factors)
```

Arguments

`A`	agreement score matrix
`factors`	number of factors (dimensions in the IRT case).

Value

a subject by factors matrix of starting values.

`factor.score.start.check`

Set and check starting values for factor scores

Description

factor.score.start.check sets the starting values of a factor score (in the IRT setting) ability matrix and checks them to make sure they are consistent with any constraints.

Usage

```
factor.score.start.check(theta.start, X, prior.mean, prior.prec, eq.constraints, ineq.const
```

Arguments

`theta.start`	The starting values for the factor scores (subject abilities). This can either be a scalar or a column vector with dimension equal to the number of observations (subjects). If this takes a scalar value, then that value will serve as the starting value for all of the thetas. The default value of NA will choose the starting values based on an eigenvalue-eigenvector decomposition of the aggreement score matrix formed from X
`X`	the matrix of manifest variables
`prior.mean`	A scalar parameter giving the prior mean of the factor scores (abilities).
`prior.prec`	A scalar parameter giving the prior precision (inverse variance) of the factor scores (abilities).
`eq.constraints`	matrix the same dimension as the factor loading matrix that has elements equal to -999 if the corresponding element of Lambda is not constrained to a constant and the constant value if it is constrained to a constant.

`ineq.constraints`
: matrix the same dimension as the factor loading matrix that has elements equal to 0 if there are no inequality constraints on that element of Lambda and either -1 or +1 if that element of Lambda is constrained to be either negative or positive respectively.

`factors`
: number of factors

Value

an observation (subject) by factors matrix of starting factor scores (abilities).

`factuniqueness.start`

starting values for factor uniquenesses

Description

forms starting values for factor uniquenesses

Usage

```
factuniqueness.start(psi.start, X)
```

Arguments

`psi.start`
: Starting values for the uniquenesses. If `psi.start` is set to a scalar then the starting value for all diagonal elements of `Psi` are set to this value. If `psi.start` is a k-vector (where k is the number of manifest variables) then the staring value of `Psi` has `psi.start` on the main diagonal. If `psi.start` is set to `NA` (the default) the starting values of all the uniquenesses are set to 0.5.

`X`
: datamatrix

Value

a p by p diagonal positive definite matrix where p is the number of manifest variables.

`form.factload.norm.prior`
forms a normal prior suitable for factor loadings

Description

forms a normal prior suitable for factor loadings

Usage

```
form.factload.norm.prior(l0, L0, K, factors, X.names)
```

Value

A list with 2 elements:

`Lambda.prior.mean`
: a K by factors matrix of means.

`Lambda.prior.precision`
: a K by factors matrix of precisions.

`form.ig.diagmat.prior`
form inverse gamma prior for diagonal of variance matrix

Description

forms and checks an inverse gamma prior for diagonal of variance matrix

Usage

```
form.ig.diagmat.prior(a0, b0, K)
```

Arguments

`a0`	Controls the shape of the inverse Gamma prior on the uniqueness. The actual shape parameter is set to `a0/2`. Can be either a scalar or a k-vector.
`b0`	Controls the scale of the inverse Gamma prior on the uniquenesses. The actual scale parameter is set to `b0/2`. Can be either a scalar or a k-vector.
`K`	Number of manifest variables.

`form.mcmc.object` *Form mcmc Object from Scythe C++ Simulation*

Description

Creates an mcmc object from the results of the posterior density simulation performed in the Scythe C++ coda.

Usage

```
form.mcmc.object(posterior.object, names, title)
```

Arguments

`posterior.object`	A list that has been returned from a call to `auto.Scythe.call()`. A list that has, at a minimum, the elements: `sampledata`, `samplerow`, `samplecol`, `mcmc`, and `thin` will also work. Here `sampledata` is an array holding the contents of a (row-major order) matrix of MCMC samples, `samplerow` and `samplecol` are the number of rows and columns of this matrix, and `mcmc` and `thin` are the number of MCMC scans after burn in and the thinning interval respectively.
`names`	Parameter names.
`title`	Title for mcmc object.

Value

An mcmc object. This can subsequently be analyzed using coda functions.

`form.mvn.prior` *Form Multivariate Normal Prior*

Description

Forms a multivariate Normal prior given user input.

Usage

```
form.mvn.prior(b0, B0, K)
```

Arguments

`b0`	A scalar or column vector of dimension `K`. If this takes a scalr value, then it serves as the prior mean for all of the betas.
`B0`	A scalar or a square matrix of dimension `K`. If this takes a scalar value, then that value times an identity matrix serves as the prior precision of beta.
`K`	The dimensionality of the multivariate Normal distribution.

Value

A list with elements:

`b0`	The prior mean b0.
`B0`	The prior precision B0

`form.seeds` *Form Seeds for Mersenne and L'Ecuyer*

Description

Forms a list of information from the seed argument of a model function and forms the appropriate seeds for the default Mersenne generator or the parallel-friendly L'Ecuyer generator.

Usage

```
form.seeds(seed)
```

Arguments

`seed`	The seed argument from a model function.

Details

If NA is passed, the Mersenne Twister generator is used with default seed 12345; if an integer is passed it is used to seed the Mersenne twister. The user can also pass a list of length two to use the L'Ecuyer random number generator, which is suitable for parallel computation. The first element of the list is the L'Ecuyer seed, which is a vector of length six or NA (if NA a default seed of `rep(12345,6)` is used). The second element of list is a positive substream number. See the MCMCpack specification for more details.

Value

`lecuyer`	A dummy variable that takes the value one if L'Ecuyer is to be used.
`seed.array`	An array of length six that contains the seeds. If Mersenne is used, only the first value of the array is used. If L'Ecuyer is used, all values are used.
`lecuyer.stream`	The stream number for the L'Ecuyer generator.

`form.wishart.prior` *Form Wishart Prior*

Description

Forms a Wishart prior given user input.

Usage

```
form.wishart.prior(v, S, K)
```

Arguments

v	The degrees of freedom parameter.
S	A scalar or square matrix. If this takes a scalar value, then that value times an identity matrix serves as the scale matrix.
K	The dimensionality of the Wishart distribution.

Value

A list with elements:

v	The prior degrees of freedom parameter.
S	The prior scale matrix.

---

parse.formula *Parse Model Formula*

Description

Parse the model formula, and return the response matrix, model matrix, and variable names in a list.

Usage

```
parse.formula(formula, data, intercept=TRUE, justX=FALSE)
```

Arguments

formula	Model formula.
data	Dataframe.
intercept	should an intercept (vector of ones) be returned in the first column of X? Default = TRUE.
justX	Should only the X matrix be calculated and returned? Default=FALSE.

Value

A list with elements:

`Y`	Response matrix.
`X`	Model matrix.
`xvars`	Variable names, as a list of strings.

`scalar.tune` *checks scalar tuning parameter*

Description

error checks a scalar tuning parameter for Metropolis-Hastings sampling.

Usage

```
scalar.tune(mcmc.tune)
```

Arguments

`mcmc.tune`	scalar tuning parameter

`sigma2.start` *Starting Values for the Conditional Error Variance*

Description

Creates starting values for σ^2 from user input. If defaults are used, the MLE is used.

Usage

```
sigma2.start(sigma2.start, formula, data)
```

Arguments

`sigma2.start`	The parameter from the **MCMCpack** model function. If NA, the MLE estimate is used. If scalar, it is checked.
`formula`	Model formula used for the LM.
`data`	Data used to fit the LM.

Value

A scalar.

`vector.tune` *checks a vector tuning parameter*

Description

error checks a vector tuning parameter for Metropolis-Hastings sampling.

Usage

```
vector.tune(mcmc.tune, K)
```

Arguments

`mcmc.tune`	tuning parameter– can be either a vector or scalar. If scalar the vector tuning parameter is rep(mcmc.tune, K).
`K`	length of vector tuning parameter.

References

Gelman, Andrew, John B. Carlin, Hal S. Stern, and Donald B. Rubin. 2003. *Bayesian Data Analysis*. London: Chapman & Hall, second edition.

Ihaka, Ross, and Robert Gentleman. 1996. "R: A Language for Data Analysis and Graphics." *Journal of Computational and Graphical Statistics* 5(3):299–314.

L'Ecuyer, P., R. Simard, E.J. Chen, and W.D. Kelton. 2002. "An Object-Oriented Random-Number Package With Many Long Streams and Substreams." *Operations Research* 50(6):1073–1075.

Matsumoto, M., and T. Nishimura. 1998. "Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator." *ACM Transactions on Modeling and Computer Simulation* 8(1):3–30.