Umacs: A Universal Markov Chain Sampler

Jouni Kerman
Department of Statistics
Columbia University
kerman@stat.columbia.edu

April 10, 2006

Abstract

Umacs (Universal Markov chain sampler) is an R software package that facilitates the construction of the Gibbs sampler and Metropolis algorithm for Bayesian inference. Umacs is a practical tool to write samplers in R. This is sometimes necessary for large problems that cannot be fit using programs like BUGS.

The user supplies the data, parameter names, updating functions, and a procedure for generating starting points. The updating functions can be some mix of Gibbs samplers and Metropolis jumps, with the latter determined by specifying a log-posterior density function. Using these inputs, Umacs writes a customized R function that automatically updates, keeps track of Metropolis acceptances (and uses acceptance probabilities to tune the jumping kernels), monitors convergence, returns a summary of the results that can be coerced into random variable objects for further processing in R.

Umacs is extendable so that users can also write their own updater-generating classes which may be used in this framework along with the existing Gibbs and Metropolis sampler-generating classes.

1 Introduction

Markov chain (MC) sampling (also called "Markov Chain Monte Carlo," or "MCMC" sampling) is now a standard tool in Bayesian inference. To write a MC sampler, we have basically three choices: (1) use a specialized application that works for certain types of models such as

MCMCPack [Martin and Quinn, 2005]; (2) use a general application such as BUGS [Lunn et al., 2000] that builds and runs a sampler given a probability model; (3) write our own sampler. These have different advantages and disadvantages: (1) although the sampling algorithm can be optimized for the particular model, it is limited; (2) for some models and for large datasets BUGS is not always possible nor practical; (3) writing our own code is time-consuming and requires debugging.

Since Markov chain sampling takes usually thousands of iterations, the faster the program, the more convenient it will be. Not all statisticians are comfortable in programming samplers in languages such as C, C++, or Fortran, but many of us write programs in R. Programming in R is in general simpler and thanks to its interactive programming environment, it makes debugging and processing the simulations convenient. It is also possible to write functions in a compiled language and access them within R. Even though our goal might be to implement the sampler in a compiled language, to speed up the programming and debugging process, we prefer to write at least the initial prototype in R.

Here we describe an approach to Markov chain sampling in R that falls between the two extremes: automatic packages that work with particular classes of models, and programming the Gibbs sampler and Metropolis algorithm from scratch.

Our approach

The idea of Umacs was conceived of the need for write a combined Gibbs/Metropolis sampler for a complex research problem on social networks [Zheng et al., 2006]. The model was first written for BUGS, but the scale of the problem proved to be too large, causing BUGS to run too slowly. A prototype for the sampler written in R ran satisfactorily; we thought then that it would be easier to write samplers in R if there was a program that provided the common structure for a Gibbs/Metropolis sampler, while the user only would provide the functions and the data that are relevant to the particular model and nothing else.

The structure of a typical Markov chain sampler remains the same for different tasks. A program implementing a Gibbs sampler contains a loop that calls a series of user-defined functions that sample values for the parameters. The difference between two Metropolis samplers is only the joint posterior function to compute the ratio of densities at two points. In practice, to write a new Markov chain sampler, we take an old program and use it as a template, changing only some of the parts of the program.

Much of the code of a Markov chain sampler is not time-critical. The core of the program consists of a loop that calls user-defined sampling routines, but outside this innermost loop,

there are no time-consuming loops or computing-intensive tasks to be done. Apart from the inner Gibbs sampling loop, the user will thus not perceive much difference between a sampler written in a compiled language and that written in an interpreted language such as R. Since the updating functions can be written in a compiled language if desired (however, simple sampling tasks still execute sufficiently fast even when written in R), then it would be reasonable to use R for writing most of the sampler.

1.0.1 A Universal Sampler Generator

We take advantage of the generality of the Markov chain sampling paradigm and implement a "Universal Markov chain sampler," a function that provides the common program structure for most Gibbs and Metropolis sampling tasks. The user must provide initialization functions for all unknowns, as well as updating functions for each Gibbs updating step and the logarithms of the posterior density functions for each Metropolis updating step. These functions are embedded into the sampler function, which is then returned to the user for execution. In other words, the program "fills in the blanks" with the code relevant to the model and it actually writes a complete sampler function.

A "universal" sampler implements the Gibbs sampler by constructing the updating functions (using the functions supplied by the user) and then embedding them into a general sampler program structure. That is, to write a MC sampler to draw from the posterior distribution $p(\theta_1, \ldots, \theta_K|\theta_0)$, one just needs to replace the innermost updating loop by a sequence of updating functions that update each block of parameters $\theta_1, \ldots, \theta_K$ sequentially. An updater function is a program that generates a draw directly from the conditional distribution, or it implements an algorithm such as Metropolis.

1.0.2 Updater-generating functions

We do not have to require the user to write every updater function from scratch: Markov chain algorithms such as Metropolis have a common structure and thus we can automatize also the generation of such updaters f_i themselves.

A Metropolis updater is a function of the joint posterior density $p(\theta_i|\theta_{-i})$ (θ_{-i} denotes the vector without the ith component block). Also, a function to draw from a proposal-generating density must be supplied. Typically this is a (multivariate) Gaussian or t-density. It is then possible to construct a general Metropolis "updater generating function" with these two densities given as the arguments: instead of specifying the complete updater function,

we specify only the parts that vary depending on the probability model.

1.0.3 Adaptive Metropolis algorithm

A Metropolis algorithm requires a proposal-generating distribution, whose performance depends greatly on the covariance structure of the kernel. An adapting Metropolis algorithm attempts to find the optimal covariance of the proposal kernel as the chain is running, adjusting the distribution periodically. Once the chain is run for a pre-specified number of iterations, the proposal distributions are fixed and the "actual chain" is run; no iterations for the final sample are saved during this period. In practice, the burn-in is done during the adaptation period so we do not have need for a special burn-in period.

2 Implementation in R

The R implementation of Umacs is a collection of various "S4"-style object classes and methods. The main function is `Sampler` which takes a list of parameter names and information about their updating algorithms. `Sampler` then returns a customized R function that is actually the sampling function itself. The user needs to specify the number of chains and the number of iterations as arguments to this function, and run it. The computer-generated sampling function returns the generated parameter sequences $\theta^{(\ell)}$ for each chain.

The sampler function is generated by embedding the user-defined updater functions into pre-defined R language templates, and then combining these templates to form a function.

2.1 Example: A robust hierarchical model

Let us consider how a simple Bayesian hierarchical model [Gelman et al., 2003, page 451] is fit using Umacs: $(y_j|\theta_j, \sigma_j) \sim N(\theta_j, \sigma_j^2)$, $j = 1, \ldots, J$ ($J = 8$) and the means θ_j are given the prior $t_\nu(\mu, \tau^2)$, thus imposing a correlation structure on $\theta = (\theta_1, \ldots, \theta_J)$. The σ_j are considered known. The hyperparameters (μ, τ) are given a uniform prior distribution given ν. We model ν^{-1} to be uniform in $(0, 1)$. To draw θ_j from the t-distribution, we draw first a variance parameter V_j from an inverse-chi-square distribution with ν degrees of freedom and scale $\nu\tau^2$, and then draw θ_j from $N(\mu, V_j)$. This gives us a fully specified probability model:

$$p(y, \theta, V, \tau^2, \mu, \nu^{-1}|\sigma^2) = N(y|\theta, \sigma^2) \cdot N(\theta|\mu, V) \cdot \text{Inv-}\chi^2(V|\nu, \tau^2) \cdot p(\tau, \mu|\nu) \cdot p(\nu^{-1}).$$

The parameters of the model and their conditional posterior distributions for the Gibbs

sampler are as follows:

$$
\begin{aligned}
J &= 8; \\
y &= (28, 8, -3, 7, -1, 1, 18, 12); \\
\sigma &= (15, 10, 16, 11, 9, 11, 10, 18); \\
p(\theta_j|\tau, \sigma_j, \mu, J, y) &= N\left(\theta_j \left| \frac{\frac{1}{\tau^2}\mu + \frac{1}{\sigma_j^2}y}{\frac{1}{\tau^2} + \frac{1}{\sigma_j^2}}, \frac{1}{\frac{1}{\tau^2} + \frac{1}{\sigma_j^2}} \right.\right), \quad j = 1, \ldots, J; \\
p(V_j|\theta_j, \tau, \nu, \mu, J) &= \text{Inv-}\chi^2\left(V_j \left| \nu + 1, \frac{\nu\tau^2 + (\theta_j - \mu)^2}{\nu + 1} \right.\right), \quad j = 1, \ldots, J; \\
p(\mu|\theta, V, J) &= N\left(\mu \left| \frac{\sum_j \theta_j/V_j}{\sum_j 1/V_j}, \frac{1}{\sum_j 1/V_j} \right.\right); \\
p(\tau^2|V, \nu, \mu, J) &= \text{Gamma}\left(\tau^2 \left| \frac{J\nu}{2} + 1, \frac{\nu}{2} \cdot \sum_j \frac{1}{V_j} \right.\right); \\
p(\nu^{-1}|\tau, V, J) &\propto \prod_{j=1}^{J} \frac{(\nu/2)^{\nu/2}}{\Gamma(\nu/2)} \cdot \tau^{\nu} \cdot V_j^{-(\nu/2+1)} \exp\left(-\frac{\nu\tau^2}{2V_j}\right), \quad \nu^{-1} \in (0, 1].
\end{aligned}
$$

J, y, and σ are observed and are shown as constant vectors. We can easily draw from the conditional distributions of θ, V, μ, τ^2 using standard functions in R, but the distribution of ν is complicated and no standard function is available; we use the Metropolis algorithm to generate draws from $p(\nu^{-1}|\tau, V, J)$, using a function proportional to the conditional density function. The only thing that is missing from this list is a procedure how to draw starting values for each Markov chain.

The structure of the sampler is specified with Umacs as a function call:

```
s <- Sampler(
  J       = 8,
  sigma   = c (15, 10, 16, 11,  9, 11, 10, 18),
  y       = c (28,  8, -3,  7, -1,  1, 18, 12),
  nu      = function () 1/nu.inv,
  theta   = Gibbs (theta.update, theta.init),
  V       = Gibbs (V.update, V.init),
  mu      = Gibbs (mu.update, mu.init),
  tau     = Gibbs (tau.update, tau.init),
  nu.inv  = SMetropolis (nu.inv.log.post, nu.inv.init)
)
```

`Sampler` is a function that constructs an R function implementing the sampler, given the parameters and the rules how to update them. The parameters `J`, `sigma`, and `y` are constants and will remain as such during the iteration process. The function call `Gibbs` specifies that the corresponding parameter is updated using a user-defined updating function, for example `theta` is updated using `theta.update`. The function `theta.init` draws the starting value for a chain. `V`, `mu`, and `tau` are updated similarly, using user-defined functions that draw directly from their corresponding conditional distributions.

The parameter `nu.inv` (ν^{-1}) is updated using the Metropolis algorithm, here using a special version, `SMetropolis`, which is optimized for scalars. The arguments are the log posterior function `nu.inv.log.post` and the function generating a starting value for ν^{-1}.

We prefer to write code referring to a variable ν (`nu`) instead of ν^{-1}, so we need to create an "updater" `nu.update` for `nu` that is a deterministic function, and not saved, returning just $1/\nu^{-1}$. We put the function at the top of the list, so the variable will be immediately available for the updating functions of V, τ, and ν^{-1}; it is not initialized anywhere else.

2.1.1 Gibbs updating functions

The conditional distributions of θ, μ, V, and τ^2 can be easily calculated analytically, so we update them each by a direct draw from the corresponding conditional distribution. The distribution of θ given the other parameters is translated into an R function for Umacs as,

```
theta.update <- function () {
  V.theta   <- 1/(1/tau^2 + 1/sigma^2)
  theta.hat <- (mu/tau^2 + y/sigma^2) * V.theta
  rnorm(J, theta.hat, sqrt(V.theta))
}
```

The updater function takes no arguments, and is embedded into the program as such. The variables `V.theta` and `theta.hat` remain local (and are thus not accessible outside the function), but the model parameters `tau` (τ), `sigma` (σ) and `J` (J) are assumed to be accessible, but they do not need to be accessible in the (global) working environment. Umacs guarantees that the model parameters are visible to the updater functions when the sampler is run, as long as they are properly defined within the `Sampler` function call, as shown above. The distribution of μ is also Gaussian and takes a similar form. Each component of V is an inverse-Chi-square variable, and τ^2 is a Gamma variable.

```
mu.update  <- function () rnorm(1, sum(theta/V)/sum(1/V), sqrt(1/sum(1/V)))
```

```
V.update   <- function () (nu*tau^2 + (theta-mu)^2)/rchisq(J, nu+1)
tau.update <- function () sqrt(rgamma(1, 1+J*nu/2, (nu/2)*sum(1/V)))
```

Internally, the functions are parsed and re-evaluated within the closure of each chain: each chain will have its own environment and its own set of parameters such as `theta`, `tau`.

2.1.2 A Metropolis updating step

The degrees-of-freedom parameter ν does not have an easy conditional distribution form, so we use the Metropolis algorithm. The Umacs updater-generating function `Metropolis` requires a function returning the logarithm of the posterior density; the function is then embedded into a program implementing the Metropolis algorithm.

We choose to generate proposals in the inverse-ν scale. For simplicity we use the built-in Gaussian kernel; in the case the proposals extend beyond the range $(0, 1]$, the log-posterior density function returns $-\infty$, guaranteeing a rejection. This method is not of course the most effective method to generate proposals for a fixed range, but it should demonstrate a possible way of dealing with the problem. The log posterior function for the variable `nu.inv` is,

```
nu.inv.log.post <- function () {
  nu <- 1/nu.inv
  if (nu.inv<=0 || nu.inv>1) return(-Inf)
  sum( 0.5*nu*log(nu/2) + nu*log(tau) -
    lgamma(nu/2) - (1+nu/2)*log(V) - 0.5*nu*tau^2/V)
}
```

Although the Metropolis algorithm actually generates samples from the distribution of ν^{-1}, we prefer to write our functions in terms of ν. In the above function, the variable `nu` is a local variable with respect to the function, and thus `nu` will not be visible to the other updating functions such as `tau.update`.

To ensure that our other updating functions have access to a variable labeled `nu`, we have specified `nu` to be a deterministic function of the parameters in the parameter list:

```
nu = function () 1/nu.inv
```

The difference between deterministic functions and Gibbs updates is that deterministic functions are not initialized and are not saved as simulations. Another possibility would be to define `nu` as a regular `Gibbs` update.

2.1.3 Initial starting points

Umacs requires functions returning initial starting points for each of the unknown parameters. These are just very simple functions generating a single starting point for the parameter; they should be random so each chain can start at a different point:

```
theta.init  <- function () rnorm(J, mean=0, sd=1)
V.init      <- function () runif(J, 0, sd(y))^2
mu.init     <- function () rnorm(1,mean(y),sd(y))
tau.init    <- function () runif(1,0,sd(y))
nu.inv.init <- function () runif(1)
```

Besides used for drawing an initial starting value for each of the chains, these functions are used for determining the dimension of the parameter. Further, this information is used to allocate space for the matrix of simulations.

2.1.4 Running the sampler

The order of the parameters in the `Sampler` function call reflects the order of updating: in this example, the parameters are updated in the order $\theta, V, \mu, \tau, \nu^{-1}, \nu$; the observed values (constants) J, y, and σ are initialized only once and made available within each chain.

`Sampler` outputs an R function, here `s`, which accepts several arguments of which the most important are: `n.iter` (number of iterations per each chain), `n.chains` (number of chains to run), `n.sims` (size of the final sample). The proportion of the burn-in period (`p.burnin`) can be set as the proportion of `n.iter`. If the size of the final sample is smaller than the number of iterations after the burn-in period, the iterations are automatically *thinned*, that is, omitted from the final sample by skipping the saving of generated iterations at specified interval, as necessary. Thinning can be disabled by setting `thin=FALSE`.

Umacs features an adaptation period which is in practice the same as the burn-in period. Currently the Metropolis algorithms take advantage of this and attempt to modify the proposal kernel during this period.

```
m <- s(n.iter=1000, n.chains=3, n.sims=200, p.burnin=0.5)
```

Once the function is running, all sequences (chains) are saved within the closure of the chain inside the sampler function. These chains are not discarded if the function run is aborted before the desired number of iterations and chains is done. All chains are saved

within the closure of the sampling function and thus are not directly accessible from the workspace (`.GlobalEnv`).

It is possible to continue running from the last iteration on to any desired number of iterations, or to change the number of chains to run.

```
m <- s(n.iter=2000, n.chains=5)


# Examine results... then iterate further:


m <- s(n.iter=5000)
```

The chains alternate at pre-specified intervals: the current chain is stopped and another is resumed from the point where it was stopped previously, until the pre-specified number of iterations (`n.iter`) are done. The $\hat{R}$ (potential scale reduction) statistics are computed and saved periodically for each of the scalar components. These can be monitored graphically as the sampler is running, if desired.

It is possible to monitor the chains for any scalar-valued parameter graphically. By specifying a "trace" parameter within the `Sampler` function call (e.g., `Trace("theta[1]")`), a graphics window is opened as the sampler starts. The window is updated as the iteration proceeds. It is also possible to monitor the $\hat{R}$ statistics with the `Trace` function.

2.1.5 Summarizing the results

The function returns a "Markov chain time series" (`mcts`) object, which contains the sequences of all chains up to the point the function was stopped. An R `summary` of this object gives a quick summary of means, standard deviations and the most common quantiles, along with the convergence diagnostics ($\hat{R}$). Entering the `mcts` object on the console we can view a summary:

```
Object of type 'mcts'
            mean       sd   2.5%   25%   50%    75%   97.5% Rhat n.eff n.sims
theta[1]   14.54 1.11e+01  -2.94  6.62 13.19  20.96   39.43 1.01   210   1002
theta[2]    7.51 7.42e+00  -6.43  2.34  7.34  12.40   21.98 1.00  1000   1002
theta[3]    3.66 1.05e+01 -20.45 -2.14  4.55  10.40   22.72 1.01   580   1002
theta[4]    7.43 7.93e+00  -7.31  2.04  7.23  12.76   22.56 1.00  1000   1002
theta[5]    3.10 7.61e+00 -12.39 -2.00  3.41   8.39   17.85 1.01   290   1002
theta[6]    4.34 7.90e+00 -13.47 -0.56  4.83   9.57   18.99 1.00   430   1002
theta[7]   12.48 8.22e+00  -2.07  7.06 11.82  17.41   30.98 1.00  1000   1002
```

```
theta[8]     8.77 1.03e+01 -11.42  2.56  8.65  14.54   30.08 1.00  1000   1002
V[1]       305.62 7.18e+02   3.60 31.82 88.56 283.80 1851.93 1.04    67   1002
V[2]       251.56 6.20e+02   3.25 25.89 77.96 215.91 1505.01 1.05    52   1002
V[3]      7015.29 2.12e+05   2.25 28.46 85.00 254.18 2112.63 1.05    44   1002
V[4]       245.51 5.02e+02   2.25 27.37 80.45 226.27 1614.39 1.06    40   1002
V[5]       375.49 3.08e+03   2.60 29.85 80.75 219.66 1623.89 1.07    38   1002
V[6]       276.66 7.23e+02   2.54 27.49 84.70 223.47 1651.92 1.06    46   1002
V[7]       301.69 9.60e+02   3.17 29.04 83.10 238.43 1668.78 1.04    59   1002
V[8]       282.86 6.95e+02   3.04 31.97 86.33 247.23 1748.53 1.06    39   1002
mu           7.61 6.37e+00  -4.49  3.74  7.61  11.65   20.02 1.01  1000   1002
tau         10.27 7.64e+00   1.78  5.04  8.11  13.06   33.03 1.08    34   1002
nu.inv       0.35 2.80e-01   0.01  0.11  0.29   0.56    0.96 1.05    62   1002
nu          11.59 2.35e+01   1.05  1.79  3.49   8.81  113.14 1.05    62   1002
```

To further process the simulations, the `mcts` objects can be coerced into *random variable objects* that can be manipulated much like any numerical objects. The random variable objects are enabled by the package `rv` [Kerman and Gelman, 2005]. We coerce the `mcts` object to a vector of random variable objects with `as.rv`. Then for convenience, we make each of the subvectors available by their names, (that is, `theta`, `V`, `mu`, `tau`, `nu.inv` and `nu`), using the function `rvattach`:

```
library("rv")
rvattach(as.rv(m))
```

We can summarize the parameter vector θ by an interval plot:

** Figure here **

The package `rv` provides convenient functions for manipulating the random variable objects, for example, to summarize the probability $\Pr(\theta_j > 0|y)$ for each $j = 1, \ldots, J$,

```
> Pr(theta>0)
theta[1] theta[2] theta[3] theta[4] theta[5] theta[6] theta[7] theta[8]
   0.925    0.849    0.664    0.831    0.658    0.720    0.946    0.826
```

For another example, to inspect the individual standard deviations $\sqrt{V_j}$, we would simply type `sqrt(V)` to obtain a vector of eight random variable objects summarizing the distribution of the random vector $(\sqrt{V_1}, \ldots, \sqrt{V_8})$.

3 Details

Arguments to the sampler-generating function

The sampler-generating function `Sampler` takes a named list of parameter-updater objects. The names represent the actual variable names in the model, and the values are objects that contain information how to update the variables. If the object corresponding to a variable name, say `sigma`, is a constant variable such as a vector or a matrix, then `sigma` will be constant during the sampling process, taking that specified value. If the value is a function call returning an instance of an updater-generating class such as `Gibbs` and `Metropolis`, then the user-supplied function arguments are merged with the program code templates specified by these classes and eventually embedded in the program that generates the sampler function.

The function `Gibbs` takes two arguments: one "initializer" function that returns an initial value for the parameter at the start of a chain, and another that draws a value from the conditional distribution of the variable given all the others.

The function `Metropolis` takes three arguments: an initializer function, the log posterior function, and a proposal-generating function. The proposal-generating function is by default a Gaussian density whose covariance is adapted to the chain during the adaptation period.

Any arguments in parameter initializing and updating functions are ignored. Any variables defined within the `Sampler` function call are always made accessible by storing them in the closure of the sampler function.

Ideally all variables in the model (including constants such as J) should be provided within the list of arguments provided to `Sampler`. This way the sampler function generated will be self-contained and will not depend on the availability variables defined in the environment where the function is run. Once the sampler function is generated, the user does not have to worry about the data variables being accessible in the working environment. Moreover, defined within the `Sampler` function call are guaranteed to b

Missing value imputation

Umacs provides an intuitive interface to model missing data. Instead of using separate parameter names for the missing components, we can model vector components directly using the square bracket-notation. For an example, suppose that in the previous example the first component of the observed vector `y` was missing. The `Sampler` function call should replace the definition for `y` with the two lines,

```
y       = c (NA,  8, -3,  7, -1,  1, 18, 12),
"y[1]"  = Gibbs (y.1.update, y.1.init),
```

The quotes around `y[1]` are mandatory because of the brackets. y_1 can be imputed using a Gibbs algorithm since conditional on the other parameters in the model, y_1 is normal with mean θ_1 and standard deviation σ_1. For the initialization routine, we might choose a starting point near the other y_i. The two functions are then defined by,

```
y.1.update <- function () rnorm(1, mean=theta[1], sd=sigma[1])
y.1.init   <- function ()
  rnorm(1, mean=mean(y, na.rm=TRUE), sd=sd(y, na.rm=TRUE))
```

We could have also written this in a more general notation,

```
"y[y.NA]"  = Gibbs (y.mis.update, y.mis.init),
```

which would define a model the missing components. The updating and initializing functions would then have to be modified to impute the missing components, using the internal variable `y.NA`, a vector containing the indices of the missing components. A variable with the `.NA` extension is automatically generated for each data parameter. The updating function would become,

```
y.mis.update <- function () {
  rnorm(length(y.NA), mean=theta[y.NA], sd=sigma[y.NA])
}
```

returning a vector of the same length as `y.NA`. During each round of iteration, the value obtained from this function would be assigned into `y[y.NA]`, that is, into the components where the missing values are.

A distinct copy of the data vector `y` is created for each chain to avoid imputing in a common variable; any variable named `y` is left untouched in the working environment.

Using the package rv [Kerman and Gelman, 2005], it is possible then to impute the random variable objects directly into the vector of constants `y`, creating a new random vector.

```
rvattach(as.rv(m), impute=TRUE)
```

In this case the vector `y` will be coerced into a random variable object (a vector) with the missing values appearing as random scalar objects, and the existing values appearing as constants, allowing manipulation of the imputed components along with the observed components as a whole. If the option `impute=TRUE` is not specified, `rvattach` would just bring over the components that were treated as random variables, that is, `y[1]`.

Optimized Metropolis updater-generating classes

The basic Metropolis algorithm moves in a multidimensional space, each proposal being a point in this space. Often we have a situation where we have a n-dimensional vector of parameters where each component is conditionally independent of the other parameters in the model. It would be very inefficient to "do n-dimensional jumps" within this space, but rather we should define one Metropolis updater for each of the n scalars. To optimize the algorithm for special cases like this, we have written several special Metropolis updater-generating classes that feature optimized code for several particular types of model. These cases arise, for example, in hierarchical regression models, with a different vector of regression parameters for each group.

Vector and Scalar Metropolis classes

The basic Metropolis algorithm, implemented by the class `Metropolis`, jumps in a multidimensional space, and consequently the proposal-generating kernel has a multidimensional covariance matrix there.

A special case of the Metropolis algorithm, Scalar Metropolis, implemented by the class `SMetropolis`, jumps in a one-dimensional space and therefore a slightly simpler program code is required, since the covariance matrix is just a scalar.

Parallel Vector and Scalar Metropolis classes

The class `SPMetropolis` (Scalar-Parallel Metropolis) generates many proposals at once and accepts or rejects them at once; that is, it is equivalent to specifying several Scalar Metropolis updating steps separately. A vector-valued variable updated using the `SPMetropolis` rule works thus with a vector of proposals rather than each scalar separately). Consequently, the log-posterior density supplied to a `SPMetropolis` call must return a vector of the same length as the parameter.

The most general case is implemented by the Parallel Metropolis class (`PMetropolis`). which expects the parameter to be an $m \times n$ matrix with conditionally independent vectors in each row; the m proposals and subsequently their individual rejection or acceptance are computed efficiently using vectorized code.

The log-posterior density supplied to a `PMetropolis` function must return a vector of the same length as the number of rows in the matrix-valued parameter. Just as the `SPMetropolis` class, the `PMetroplis` code keeps track of m separate covariance matrices.

Future development and extension

We see several possible areas where to improve the current version of Umacs. Initial starting values could be generated automatically from the distributions previously generated for the

It is also possible to extend Umacs by adding arbitrary user-defined updater-generating classes. The structure of the program is not dependent on any sampling algorithm. One can for example create general updater-generating classes for importance sampling, rejection sampling, and grid sampling. In general, one can use Umacs for all kinds of tasks involving generating simulation draws.

4 Acknowledgements

Andrew Gelman suggested that a common framework for Markov chain samplers should be possible. We thank also Tian Zheng, Yuejing Ding, Shouhao Zhou, Chris Paciorek, Grazia Pittau, and Aleks Jakulin for testing the program and for giving valuable suggestions.

References

Andrew Gelman, John B. Carlin, Hal S. Stern, and Donald B. Rubin. *Bayesian Data Analysis*. Chapman & Hall/CRC, London, 2nd edition, 2003.

Jouni Kerman and Andrew Gelman. Manipulating and summarizing posterior simulations using random variable objects. Technical report, Department of Statistics, Columbia University, 2005.

D. J. Lunn, A. Thomas, N. Best, and D. Spiegelhalter. WinBUGS – a Bayesian modelling framework: concepts, structure, and extensibility. *Statistics and Computing*, 10:325–337, 2000.

Andrew D. Martin and Kevin M. Quinn. `MCMCpack` 0.6-6. `http://mcmcpack.wustl.edu/`, 2005.

Tian Zheng, Matthew J. Salganik, and Andrew Gelman. How many people do you know in prison?: Using overdispersion in count data to estimate social structure in networks. *Journal of the American Statistical Association*, 2006. To appear.