Getting started with Umacs: A Universal Markov Chain Sampler version 0.912

Jouni Kerman
jouni@kerman.com

February 12, 2007

1 Introduction

Umacs (Universal Markov chain Sampler) is a framework for building and executing iterative samplers using R. A detailed introduction is found in Kerman [2006]. This document is about the practical details on how to get your model programmed in Umacs.

1.1 Getting started

Install the package `Umacs`. Recommended: install also the package 'rv' using the Package Installer command in R (from the menu). Then, load them, using:

```
library(Umacs)
library(rv)
```

2 Setting up the sampler for your model

The function `Sampler(...)` builds a sampler *function* that runs the actual sampler chains. It takes the functions and data provided by the user and outputs an R function that can be used for sampling from this particular model *only*.

Here is an example, using a model from [Gelman et al., 2003, Appendix]:

```
s <- Sampler(
  .title  = "Eight Schools Problem, varying degrees of freedom",
```

```
  J       = 8,
  sigma   = c (15, 10, 16, 11,  9, 11, 10, 18),
  y       = c (28,  8, -3,  7, -1,  1, 18, 12),
  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),
  nu      = Gibbs (nu.update, nu.init),
  Trace("theta[1]"),
  Trace("nu.inv")
)
```

A `Sampler` function call takes a list of all parameters in the model, including:

1. Special Umacs variables such as `.title`
2. Data (constants) such as `J, sigma.y, mu, tau`
3. Unknown parameters such as `theta`, `mu`, and `tau`.
4. Other directives or "virtual parameter updates" such as `Trace(...)` which displays a graphical trace plot (how this is done may change slightly in the future versions)

The name of an unknown parameter is followed by the definition of the updating scheme; the constant (data) parameters are to be given their values within this .

The parameters are updated in the given order.

2.1 Give your sampler a name

`.title` is the name that you give to this sampler. It is there *only* for convenience and for your reference. If you type the name of your sampler function (here, `s`) on the console, you'll see the title at the end of the function listing.

2.2 Decide what your unmodeled data are

You will need to declare the data variables that need to be accessed in the updating functions. In the above example, the data consists of three variables: `J`, `sigma.y`, and `y`.

If a parameter, such as `y` here, is defined in the workspace, you can leave it out since it will be accessed from there. But, for completeness we recommend that should be defined also within the `Sampler` call. If a code parameter such as `y` has missing values and you want to model them to be imputed, you MUST specify it here.

As a rule of thumb, unless your data variable has no missing data and is extraordinarily large and thus memory-consuming, you should define the data variable within the `Sampler` function call.

2.3 Set up initialization functions for all unknown parameters

The unknown parameters in the above example are `theta`, `V`, `mu`, `tau`, `nu.inv` and `nu`.

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)
nu.init     <- function () 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 for allocating space for the matrix of simulations.

As for the deterministic parameter ν (that is a function of the random parameter ν^{-1}) there is no need to draw a random starting point; however we must return some scalar so that Umacs knows that ν is a one-dimensional variable.

Some important points to remember:

1. An initializer function for a parameter (say, `theta`) must return a valid draw of a single instance of `theta`, with the correct length and dimension.

2. If a parameter is *not* an array, do *not* set the dimension attribute.

3. *All function arguments are ignored.* All initialization functions are of the form `param.init <- function () ...`.

4. The name of the function can be whatever you want; it will also be ignored.

5. All variables within these functions should refer only to the parameters that are declared within the `Sampler` function call and to global variables. However, for clarity and completeness, it's best to declare *all* dependent data variables in the sampler call.

2.4 Decide on how you want to update your parameters

Each unknown parameter must have a corresponding updating scheme defined.

2.4.1 For Gibbs sampling, set up the updater functions

A Gibbs-updated parameter is defined the following way:

```
parametername = Gibbs( updater.function, initializer.function )
```

For the example above, we have written the following updater functions. Each Gibbs update draws one simulation from the conditional distribution of that particular parameter.

```
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))
}
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)))
```

1. An updater function for a parameter (say, 'theta') must return a valid draw of a single instance of theta, (of course, of the same length and dimension as the draw generated by the initializer function of theta)

2. The updater function is treated as a function so you may use `return` statements; however arguments are ignored! You should define it as `function ()`.

3. Any variables that are referred to in these functions should be declared as parameters and data (constants) in the `Sampler` function call.

2.4.2 Parameters that are deterministic functions

In our above example, we have one parameter which is just the inverse of `nu.inv`. To ensure that this parameter is correctly updated within the loop, it must be declared as a `Gibbs` update. After all, given the other parameters, `nu` has a point-mass distribution.

```
nu.update <- function () 1/nu.inv
```

2.4.3 Metropolis updaters

This example has one parameter, ν^{-1} (`nu.inv`), updated using a Metropolis algorithm.

To define a Metropolis update, we need function that returns the logarithm of the posterior density of the parameter. Again, all arguments to the function are ignored. And, all variables within this function call must be declared in the Sampler function call (or be accessible in the workspace).

```
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)
}
```

In the above function, `nu` is a *local* variable and will not change the value of the parameter `nu`.

To guarantee rejection, you can have the function return `-Inf`.

This particular parameter is declared as,

```
  nu.inv  = SMetropolis (nu.inv.log.post, nu.inv.init),
```

where `nu.inv.init` is the initialization function.

There are several Metropolis updater classes, described briefly below.

2.5 Metropolis schemes

2.5.1 Metropolis : Vector Metropolis

This is the basic Metropolis algorithm that works for scalars and vectors, but it is best used for vectors which are at least of length 2. For scalars, use `SMetropolis`.

2.5.2 SMetropolis : Scalar Metropolis

`SMetropolis` is optimized for scalars.

2.5.3 PSMetropolis : Parallel-Scalar Metropolis

The parameter using `PSMetropolis` updating scheme is supposed be *a vector of independent* scalars can all be updated at once, independent of each other.

The log posterior function specified *must* return a vector of the same length as the parameter itself.

2.5.4 PMetropolis : Parallel (Vector) Metropolis

In this parameter declaration,

```
theta  = PMetropolis(theta.logpost, theta.init)
```

theta is *a matrix of rows of independent vectors* that are all updated at once.

theta.logpost must return a vector of the same length as the number of rows in theta (one per each independent vector).

If the independent vectors are in the *columns* of the matrix, you must specify the option `byCol=TRUE`:

```
theta  = PMetropolis(theta.logpost, theta.init, byCol=TRUE)
```

2.5.5 Defining a common log posterior function

For convenience, we may define one common log posterior function by,

```
.logpost = my.logpost.function,
```

The dot in front of 'logpost' is essential: it is the name of an internal variable.

We could then omit the log posterior function in the Metropolis class calls, just defining the initializer, e.g.,

```
alpha  = Metropolis(init=function () rnorm(1, 0, 1)),
beta   = Metropolis(init=function () rnorm(1, 0, 1))
```

The initialization function, however, *must* be specified.

It is usually best to provide a parameter-specific log posterior functions.

2.6 Tracing the chain

If you want to monitor a parameter real-time, you can define a (nameless) "virtual parameter" that "updates the graphical display," with the updater function `Trace`, e.g.:

```
Trace("theta[1]")
```

The parameter must be one-dimensional. There may be several trace directives. The trace plot is updated every (n.iter/10)th iteration.

To trace the potential scale reduction statistic ($\hat{R}$, "R-hat"), use,

```
Trace(".Rhat['theta[1]']")
```

for example. The dot "." before Rhat is there to distinguish it from your parameters (which must not have dots prepended)

$\hat{R}$ is computed after the burn-in period and then every 50 iterations.

To trace the largest R-hat of them all:

```
Trace(".Rhatmax")
```

3 Running the sampler

The Sampler function call returns a function.

```
s <- Sampler( ... )
```

This function accepts several parameters:

```
s(n.iter=200, n.chains=3, n.sims=1000)
```

1. `n.iter` is the number of iterations to run *per each chain*;
2. `n.chains` is the number of chains to run;
3. `n.sims` is the number of simulations you wish to keep eventually.

3.1 Running a certain number of chains and iterations

`n.chains` is the number of chains to run. It is by default set to 3. `n.iter` is the number of iterations to run *per each chain*. Thus Umacs will iterate a total of `n.chains*n.iter` times.

Usually you want to specify the number of iterations to some number, but if you don't specify it, the default `n.sims=10` will be used. After finishing, you'll have the option to *resume* sampling from the previous point.

3.2 Resuming the sampler after it stops

You may resume sampling after the sampler stops. The sampler function (here, `s`) contains all simulation draws already drawn, so it does not have to regenerate them.

Supposing that you have finished the 200 iterations but now want to run a total of 1000. You don't have to rebuild the sampler, just type:

```
s(n.iter=1000)
```

to continue iterations from 201 to 1000.

The result that `s(...)` returns is an *mcts* object ("mcts"="Markov chain time series"). When printed on the console, a summary of the sampling result is given, along with the convergence diagnostics.

If you want, you can add chains if you want:

```
s(n.chains=5)
```

would add two more chains to the sampler.

3.3 Burn-in/adaptation period

The burn-in period is `1...n.iter*p.burnin` ; p.burnin is by default 0.5.

If you have finished sampling with `n.iter = a`, and resume sampling with a new `n.iter` that is larger than `a`, the end of the new burn-in period is set to `b*p.burnin`.

If `b` $\leq$ `a*p.burnin`, no further adaptation is done (at least one half of the new n.iter has already been sampled so the burn-in period is done) but if $b >$`a*p.burnin`, adaptation is continued for `b*p.burnin-a` iterations (from the point where the sampling stopped last time to the half way of the end of the new burn-in period.)

Burn-in period is the same as the *adaptation period*. During this time, the Metropolis kernels are adapted to optimize the acceptance rate. These optimization routines are currently very rough and simple but we hope to improve them.

3.4 Number of simulations to keep

`n.burnin` will be set to `n.iter*p.burnin`, if it is not specified as one of the arguments. (see below for the *burn-in/adaptation period*.)

If you don't specify `n.sims`, the default value will be set to `n.chains*(n.iter-n.burnin)` If you want to set the number of simulations to keep to, say, 1000, specify `n.sims=1000` as an argument. This value will be remembered and will not be changed to the default (variable) value unless you specifically set `n.sims` to `NA`. This means that "set `n.sims` to `n.chains*(n.iter-n.burnin)`."

If you don't save the sampler result (the mcts object) into a variable, you can retrieve it any time simply by calling the sampler function without arguments:

```
x <- s()
```

This actually attempts to resume the sampler, but since the number of iterations (n.iter) has not changed and n.iter draws are already generated, all chains appear to be finished, and the sampler just returns the draws.

`n.sims` determines how many draws are actually saved.

If n.iter/2 is smaller than n.sims, all n.iter/2 simulations are used. If n.iter/2 is larger than n.sims, thinning is attempted and then a random sample is drawn from the remaining simulations to obtain exactly n.sims draws. You can change n.sims any time after the sampling is finished:

```
s(n.sims=500)
```

will set n.sims to 500 and return the mcts object with 500 simulations for each variable (of course, provided that n.iter/2>n.sims)

When the mcts object is coerced into a r.v. object by `as.rv(x)`, the simulation matrix is scrambled rowwise to simulate independence.

If you want to use a different proportion of burn-in iterations, say to 0.2, specify the argument,

```
p.burnin=0.2
```

4 Processing the results

Install the rv package and invoke it by

```
library("rv")
```

The mcts object that is created by the Umacs-generated sampler can be coerced into an rv object for easy simulation manipulation:

```
x <- as.rv(x)
```

and further split into `theta`, `mu`, and `tau`, and attach each variable separately by

```
rvattach(x)
```

Read about the design principles in Kerman and Gelman [2005].

5 Inserting "raw code"

It is possible to insert a piece of "raw code" within the iteration loop. In practice this is done by specifying a function in the Sampler function call. Its returning value will not be automatically assigned to any parameter. This is used usually to accomplish reparameterization steps.

Suppose that we have a function 'renormalize' that is supposed to make small changes in some of the parameters:

```
renormalize <- function() {
  const     <- log (sum(exp(beta[1:2])/0.0044))
  alpha    <<- alpha + const
  mu.alpha <<- mu.alpha + const
  beta     <<- beta - const
  mu.beta  <<- mu.beta - const
}
```

You must use the "double arrow assignment operator" (`<<-`) to change the value of a parameter that is initialized outside this function. Any variables that are assigned values with "`<-`" are *local* to this function and thus not saved after the function call is finished (in the above example, the value of `const` is discarded after the function call, but the values of the parameters `alpha`, `mu.alpha`, `beta`, and `mu.beta` are changed.)

6 Missing values and imputation

It is possible to use brackets in a parameter name, e.g.

```
"theta[1]"   = Gibbs( ... ),
"theta[2:3]" = Gibbs( ... ),
```

The values are then imputed into the corresponding components of theta. Note that you *must* use quotes whenever you specify a name with brackets. For regular parameter names, the quotes are optional.

"theta" must be declared or else imputing will not work. If all components of theta are unknown, declare a variable that has all `NA`, e.g.:

```
theta = rep(NA, 32),
```

You can also use variable names in the expression:

```
J = 8,
k = 16,
"theta[1:J]" = ...,
"theta[c(1,2,8,k)]" = ...
```

The parameter "names" are embedded as such in the code and parsed by R.

The indices of the missing values in a data variable (say, `y`) are stored in a variable with the same name plus an extension "`.NA`". So in the case of a data variable y, the indices with missing values are available as `y.NA`. To specify sampling for missing values, specify

```
"y[y.NA]" = ...
```

for convenience, this can be specified as

```
y.mis = ...
```

but there will be no variable called `y.mis`. It is just an alias to `"y[y.NA]"`.

Note. You can alternatively also initialize the full vector with some dummy non-missing values and then just selectively impute the components within the updating function. If you initialize a vector with missing values, you'll get an error.

7 Conventions

A parameter name must not have a dot prepended; these are reserved for "internal" parameters. The special reserved parameter names are:

```
.title
.logpost
.Rhat
.Rhatmax
```

Any other name starting with a dot "." is illegal.

8 Disclaimer

This program is a work in progress, and it may contain bugs. Many new features will be eventually (and hopefully) added.

For information about random variables in R, please refer to Kerman and Gelman [2005].

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. Umacs: A Universal Markov Chain Sampler. Technical report, Department of Statistics, Columbia University, 2006.

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