The *Stata Journal* publishes reviewed papers together with shorter notes or comments, regular columns, book reviews, and other material of interest to Stata users. Examples of the types of papers include 1) expository papers that link the use of Stata commands or programs to associated principles, such as those that will serve as tutorials for users first encountering a new field of statistics or a major new technique; 2) papers that go "beyond the Stata manual" in explaining key features or uses of Stata that are of interest to intermediate or advanced users of Stata; 3) papers that discuss new commands or Stata programs of interest either to a wide spectrum of users (e.g., in data management or graphics) or to some large segment of Stata users (e.g., in survey statistics, survival analysis, panel analysis, or limited dependent variable modeling); 4) papers analyzing the statistical properties of new or existing estimators and tests in Stata; 5) papers that could be of interest or usefulness to researchers, especially in fields that are of practical importance but are not often included in texts or other journals, such as the use of Stata in managing datasets, especially large datasets, with advice from hard-won experience; and 6) papers of interest to those who teach, including Stata with topics such as extended examples of techniques and interpretation of results, simulations of statistical concepts, and overviews of subject areas.

The *Stata Journal* is indexed and abstracted by *CompuMath Citation Index*, *Current Contents/Social and Behavioral Sciences*, *RePEc: Research Papers in Economics*, *Science Citation Index Expanded* (also known as *SciSearch*), *Scopus*, and *Social Sciences Citation Index*.

For more information on the *Stata Journal*, including information for authors, see the webpage

http://www.stata-journal.com

The *Stata Journal* is published quarterly by the Stata Press, College Station, Texas, USA.

Address changes should be sent to the *Stata Journal*, StataCorp, 4905 Lakeway Drive, College Station, TX 77845, USA, or emailed to sj@stata.com.

Adaptive Markov chain Monte Carlo sampling and estimation in Mata

Matthew J. Baker
Hunter College and the Graduate Center, CUNY
New York, NY
matthew.baker@hunter.cuny.edu

Abstract. I describe algorithms for drawing from distributions using adaptive Markov chain Monte Carlo (MCMC) methods; I introduce a Mata function for performing adaptive MCMC, **amcmc()**; and I present a suite of functions, **amcmc_*()**, that allows an alternative implementation of adaptive MCMC. **amcmc()** and **amcmc_*()** can be used with models set up to work with Mata's **moptimize()** (see [M-5] **moptimize()**) or **optimize()** (see [M-5] **optimize()**) or with standalone functions. To show how the routines can be used in estimation problems, I give two examples of what Chernozhukov and Hong (2003, *Journal of Econometrics* 115: 293–346) refer to as quasi-Bayesian or Laplace-type estimators—simulation-based estimators using MCMC sampling. In the first example, I illustrate basic ideas and show how a simple linear model can be fit by simulation. In the next example, I describe simulation-based estimation of a censored quantile regression model following Powell (1986, *Journal of Econometrics* 32: 143–155); the discussion describes the workings of the command **mcmccqreg**. I also present an example of how the routines can be used to draw from distributions without a normalizing constant and used in Bayesian estimation of a mixed logit model. This discussion introduces the command **bayesmixedlogit**.

Keywords: st0354, amcmc(), amcmc_*(), bayesmixedlogit, mcmccqreg, Mata, Markov chain Monte Carlo, drawing from distributions, Bayesian estimation, mixed logit

1 Introduction

Markov chain Monte Carlo (MCMC) methods are a popular and widely used means of drawing from probability distributions that are not easily inverted, that have difficult normalizing constants, or for which a closed form cannot be found. While often considered a collection of methods with primary usefulness in Bayesian analysis and estimation, MCMC methods can be applied to a variety of estimation problems. Chernozhukov and Hong (2003), for example, show that MCMC methods can be applied to many problems of traditional statistical inference and used to fit a wide class of models—essentially, any statistical model with a pseudoquadratic objective function. This class of models encompasses many common econometric models that have traditionally been fit by maximum likelihood or generalized methods of moments. This article describes some Mata functions for drawing from distributions by using different types of "adaptive MCMC" algorithms. The Mata implementation of the algorithms is intended to allow straightforward application to estimation problems.

While it is well known that MCMC methods are useful for drawing from difficult densities, one might ask: why use MCMC methods in estimation? Sometimes, maximizing an objective function may be difficult or slow, perhaps because of discontinuities or nonconcave regions of the objective function, a large parameter space, or difficulty in programming analytic gradients or Hessians. When bootstrapping of standard errors is required, estimation problems are exacerbated because of the need to refit a model many times. MCMC methods may provide a more feasible means of estimation in these cases: estimation based on sampling directly from the joint parameter distribution does not require optimization and still provides the desired result of estimation—a description of the joint distribution of parameters. MCMC methods are a popular means of implementing Bayesian estimators because they allow one to avoid hard-to-calculate normalizing constants that often appear in posterior distributions. Unlike extrema-based estimation, Bayesian estimators do not rely on asymptotic results and thus are useful in small-sample estimation problems or when the asymptotic distribution of parameters is difficult to characterize.

In this article, I describe a Mata function, `amcmc()`, that implements adaptive or nonadaptive MCMC algorithms. I also describe a suite of routines, `amcmc_*()`, that allows implementation via a series of structured functions, as one might use Mata functions such as `moptimize()` (see [M-5] **moptimize()**) or `deriv()` (see [M-5] **deriv()**). The algorithms implemented by the Mata routines more or less follow Andrieu and Thoms (2008), who present an accessible overview of the theory and practice of adaptive MCMC.

In section 2, I provide an intuitive overview of adaptive MCMC algorithms, while in section 3, I describe how the algorithms are implemented in Mata by `amcmc()` or by creating a structured object via the suite of functions `amcmc_*()`. In section 4, I describe four applications. I show how the routines might be used in a straightforward parameter estimation problem, and I describe how methods can be applied to a more difficult problem: censored quantile regression. In this discussion, I also introduce the `mcmccqreg` command. I then show how routines can be used to sample from a distribution that is hard to invert and lacks a normalizing constant. In a final example in section 4, I apply the methods to Bayesian estimation of a mixed logit model following Train (2009) and introduce the `bayesmixedlogit` command. In section 5, I sketch a basic Mata implementation of an adaptive MCMC algorithm, which I hope will give users a template for developing adaptive MCMC algorithms in more specialized applications. In section 6, I conclude and offer some sources for additional reading.

2 An overview of adaptive MCMC algorithms

At the heart of adaptive MCMC sampling is the Metropolis–Hastings (MH) algorithm. An MH algorithm is built around a target distribution that one wishes to sample from, $\pi(X)$, and a proposal distribution, $q(Y, X)$.[1] If one is mainly interested in applying MCMC in estimation, one may think of $\pi(X)$ as a conditional likelihood function, and X can be thought of as a $1 \times n$ row vector of parameters. A basic MH algorithm is described in table 1.

1. For ease of comparison, I follow the notation of Andrieu and Thoms (2008) wherever possible.

Table 1. An MH algorithm. The proposal distribution is denoted by $q(Y, X)$, while the target distribution is $\pi(X)$. $\alpha(X, Y)$ denotes the draw acceptance probability.

```
Basic MH algorithm

1: Initialize start value $X = X_0$ and draws $T$.
2: Set $t = 0$ and repeat steps 3–6 while $t \leq T$:
        3: Draw a candidate $Y_t$ from $q(Y_t, X_t)$.
        4: Compute $\alpha(Y_t, X_t) = \min\left\{\frac{\pi(Y_t)}{\pi(X_t)}\frac{q(Y_t,X_t)}{q(X_t,Y_t)}, 1\right\}$.
        5: Set $X_{t+1} = Y_t$ with prob. $\alpha(Y_t, X_t)$,
               $X_{t+1} = X_t$ otherwise.
        6: Increment $t$.
Output: The sequence $(X_t)_{t=1}^{T}$.
```

The MH algorithm sketched in table 1 has the property that candidate draws Y_t that increase the value of the target distribution, $\pi(X)$, are always accepted, whereas candidate draws that produce lower values of the target distribution are accepted with only probability α. Under general conditions, the draws $X_1, X_2, \ldots, X_T$ converge to draws from the target distribution, $\pi(X)$; see Chib and Greenberg (1995) for proofs. One can see the convenience the algorithm provides in drawing from densities of the form $\pi(X) = \pi'(X)/K$, where K is some perhaps difficult-to-calculate normalizing constant. Computation of K is unnecessary, because it cancels out of the ratio $\pi(X)/\pi(Y)$. The proposal distribution, $q(Y, X)$, is where the "Markov chain" part of "Markov chain Monte Carlo" comes in. It is what distinguishes MCMC algorithms from more general acceptance-rejection Monte Carlo sampling: candidate draws depend upon previous draws in this function.

MCMC algorithms are simple and flexible, and they are therefore applicable to a wide variety of problems. However, they can be challenging to implement, mainly because it can be hard to find an appropriate proposal distribution, $q(Y, X)$. If $q(Y, X)$ is chosen poorly, coverage of the target distribution, $\pi(X)$, may be poor. This is where adaptive MCMC methods are used because they help "tune" the proposal distribution. As an adaptive MCMC algorithm proceeds, information about acceptance rates of previous draws is collected and embodied in some set of tuning parameters θ. Slow convergence or nonconvergence of an algorithm like that in table 1 is often caused by acceptance of too few or too many candidate draws: if the algorithm accepts too few candidate draws, candidates are too far away from regions of the support of the distribution where $\pi(X)$ is large; if too many candidates are accepted, candidates occupy an area of the support of the distribution clustered closely around a large value of $\pi(X)$. Accordingly, if the acceptance rate is too low, the tuning mechanism contracts the search range; if the acceptance rate is too high, it expands the search range. As a practical matter, one augments the proposal distribution with the tuning parameters θ so that the proposal distribution is something like $q(Y, X) = q(Y, X, \theta)$. A description of such an algorithm appears in table 2.

The algorithm in table 2 also relies on a simplification of the basic MCMC algorithm presented in table 1, which results when a symmetric proposal distribution is used so that $q(Y, X, \theta) = q(X, Y, \theta)$. With a symmetric proposal distribution—the (multivariate) normal distribution being a prominent example—the proposal distribution drops out of the calculation of the acceptance probability in step 4 of the algorithm; this results in the simplified acceptance probability $\alpha(Y, X_t) = \min[\{\pi(Y)\}/\{\pi(X_t)\}, 1]$. All the Mata routines discussed in this article use a multivariate normal density for a proposal distribution.

Table 2. Overview of an adaptive MH algorithm with tuning and a symmetric proposal distribution

```
Adaptive MH algorithm (with symmetric $q$)

1: Initialize start value $X = X_0$, draws $T$, and tuning parameters $\theta_0$.
2: Set $t = 0$ and repeat steps 3–7 while $t \leq T$:
        3: Draw a candidate $Y_t$ from $q(Y_t, X_t, \theta_t)$.
        4: Compute $\alpha(Y_t, X_t) = \min\left\{\frac{\pi(Y_t)}{\pi(X_t)}, 1\right\}$.
        5: Set $X_{t+1} = Y_t$ with prob. $\alpha(Y_t, X_t)$,
                $X_{t+1} = X_t$ otherwise.
        6: Update $\theta_{t+1} = f(\theta_t, X_0, X_1, X_2, \ldots, X_t)$.
        7: Increment $t$.
Output: The sequence $(X_t)_{t=1}^{T}$.
```

There is an important theoretical problem with an adaptive MCMC algorithm like that in table 2. Tuning the proposal distribution results in "loss of π as an invariant distribution of the process (X_t)" (Andrieu and Thoms 2008, 345) if it is not done carefully. Tuning the proposal distribution alters the long-run behavior of the algorithm so that it no longer produces the sought-after draws from the target distribution, $\pi(X)$. A solution to this problem is to tune the proposal distribution for some burn-in period and then stop tuning so that the proposal distribution is stationary. Another solution is to set up the algorithm so that tuning eventually recedes from the algorithm. The latter approach is referred to as vanishing or diminishing adaptation (Andrieu and Thoms 2008; Rosenthal 2011). With vanishing adaptation, if the algorithm runs for a sufficient number of iterations, the proposal distribution stabilizes while also (hopefully) being tuned to provide good coverage of the target distribution. The Mata functions presented in this article are built to work with vanishing adaptation, but they can also be set up so that no adaptation of the proposal distribution occurs.

2.1 Adaptive MCMC with vanishing adaptation

Before discussing implementation of vanishing adaptation, I must discuss how frequently candidate draws should be accepted by an MCMC algorithm. Ideally, the acceptance rate should be such that good coverage of the target distribution is achieved with the smallest possible number of draws. Rosenthal (2011) provides an accessible treatment on optimal acceptance rates in adaptive MCMC algorithms and a summary of the main ideas and results. At the risk of oversimplifying, I provide some guidelines. For univariate distributions, the optimal acceptance rate is about 0.44, and as the dimension of $\pi(X)$ increases to infinity, the optimal acceptance rate converges to 0.234. Rosenthal (2011) points out that moderate departure from these rates is unlikely to greatly damage algorithm performance and that often for distributions with even relatively small dimension (that is, $d \geq 5$), the optimal acceptance rate is close to the asymptotic bound of 0.234. In table 3, I describe an algorithm that is tuned toward a targeted acceptance rate α^* (presumably in or close to the range $[0.234, 0.44]$).

Table 3. Overview of an adaptive MH algorithm with a multivariate normal proposal distribution and a specific tuning mechanism.

```
Adaptive MCMC algorithm with normal proposal and vanishing adaptation

1: Set starting values $X_0$, $\mu_0$, $\Sigma_0$, $\lambda_0$, $\alpha^*$, $\delta$ ($\delta > 0$), and draws $T$.
2: Set $t = 0$ and repeat steps 3–10 while $t \leq T$:
        3: Draw a candidate $Y_t \sim MVN(X_t, \lambda_t \Sigma_t)$.
        4: Compute $\alpha(Y_t, X_t) = \min\left\{\frac{\pi(Y_t)}{\pi(X_t)}, 1\right\}$.
        5: Set $X_{t+1} = Y_t$ with prob. $\alpha(Y_t, X_t)$,
              $X_{t+1} = X_t$ otherwise.
        6: Compute weighting parameter $\gamma_t = \frac{1}{(1+t)^\delta}$.
        7: Update $\lambda_{t+1} = \exp\{\gamma_t(\alpha(Y_t, X_t) - \alpha^*)\}\lambda_t$.
        8: Update $\mu_{t+1} = \mu_t + \gamma_t(X_{t+1} - \mu_t)$.
        9: Update $\Sigma_{t+1} = \Sigma_t + \gamma_t\left\{(X_{t+1} - \mu_t)(X_{t+1} - \mu_t)' - \Sigma_t\right\}$.
        10: Increment $t$.
Output: The sequence $(X_t)_{t=1}^T$.
```

Table 3 is a fairly complete description of how an adaptive MCMC algorithm might be implemented and how the Mata functions presented in section 3 actually operate. In step 1, the algorithm starts with the initial value X_0; an initial variance–covariance matrix for proposals, Σ_0; an initial value of a scaling parameter, λ_0; and a targeted acceptance rate, α^*. The algorithm also requires a value for what can be considered an averaging or damping parameter, δ, which controls how quickly the impact of the tuning mechanism decays through the parameter $\gamma_t = 1/(1+t)^\delta$, calculated in step 6. For large values of δ, adaptation ceases quickly as γ rapidly approaches zero; for values of δ close to zero, adaptation occurs more slowly, and the algorithm uses more information about past draws in tuning proposals. The Mata routines presented below allow the user to specify such a δ parameter when implementing the algorithm.[2] In steps 8 and 9, the algorithm updates the mean and covariance matrix of the proposal distribution according to the weighting parameter γ_t, and because γ_t eventually decays to zero, updating ceases, and

2. One might prefer this value to be as close to its upper bound as possible to reduce the impact of tuning quickly; the tradeoff is that the proposal distribution may not be as well adapted.

the algorithm eventually carries on with stable proposal distribution characterized by $\lambda_{t+1} = \lambda_t$, $\mu_{t+1} = \mu_t$, and $\Sigma_{t+1} = \Sigma_t$.

If a researcher wished to write his or her own adaptive MCMC routine, the specification of the weighting scheme embodied in γ and δ on table 3 could be extended. Andrieu and Thoms (2008) describe some other possibilities for adaptation, including stochastic schemes or weighting functions that adapt as the algorithm continues. As described by Andrieu and Thoms (2008, 356), virtually anything goes with the tuning process, provided that the sequence γ_t satisfies the following properties:

$$\sum_t^{\infty} \gamma_t = \infty, \quad \sum_t^{\infty} \gamma_t^{1+\rho} < \infty; \rho > 0$$

These conditions are satisfied by the weighting parameter used in the adaptive algorithm in table 3 so long as $\delta \in (0,1)$: the reason is that under these circumstances, $\sum_t^{\infty} \gamma_t$ diverges, but a sufficiently large value of ρ that forces the series $\{1/(1+t)^{\delta}\}^{1+\rho}$ to converge can always be found.

A last detail to address is how to initialize the value of the scaling parameter λ at the start of the algorithm. According to Andrieu and Thoms (2008, 359), theory suggests that a good place to start with the scaling parameter is $\lambda \approx 2.38^2/d$, where d is the dimension of the target distribution. The Mata routines presented below all use this value as a starting point, with one exception.

There are many variations on the basic theme of the algorithm presented in table 3. One possibility is one-at-a-time, sequential sampling of values from the distribution, which produces a "Metropolis-within-Gibbs" type sampler. Another possibility is to work halfway between the "global" sampling algorithm of table 3 and the sequential sampling, creating what might be labeled a block adaptive MCMC sampler.[3] In my experience, Metropolis-within-Gibbs samplers or block samplers are often useful in situations in which variables are scaled very differently or in situations where the researcher might not have good intuition about starting values.

Related to determining how to execute the algorithm is the issue of how to choose T, the length of the run. One would like to choose T large enough so that the convergence criteria mentioned above are satisfied and enough draws are produced for reliable statistical inference. How does one know that the algorithm has achieved these goals? This is a surprisingly complex question that really does not have a good answer. While one can often detect problems with the algorithm, there is no way to guarantee that the algorithm has converged. Gelman and Shirley (2011) describe different techniques for assessing performance and convergence of the run, but they also emphasize the complementary roles of visual inspection of results, understanding the application, and understanding the subject matter. These issues are discussed at greater length in the conclusion.

3. I follow the convention of referring to a sequential sampler as a "Metropolis-within-Gibbs" sampler, even though many find this terminology misleading; see Geyer (2011, 28–29). What I call a "block" sampler, some might call a "block-Gibbs" sampler.

3 Adaptive MCMC in Mata

3.1 A Mata function

Syntax

The first Mata implementation of the algorithms described in section 2 is through the Mata function `amcmc()`,[4] which uses different types of adaptive MCMC samplers based upon user-provided information. In addition to describing details of sampling (specification of draws, weighting parameters, and acceptance rates), the user can specify whether sampling is to proceed all at once ("globally"), in blocks, or sequentially. The user can also set up `amcmc()` to work with a "stand-alone" distribution or with an objective function previously set up to work with `moptimize()` or `optimize()`. The syntax is as follows:

real matrix `amcmc(`*string rowvector alginfo*`,`
pointer (real scalar function) scalar lnf`,` *real rowvector xinit*`,`
real matrix Vinit`,` *real scalar draws*`,` *real scalar burn*`,`
real scalar delta`,` *real scalar aopt*`,` *transmorphic arate*`,`
transmorphic vals`,` *transmorphic lambda*`,`
real matrix blocks | *transmorphic M*`,` *string scalar noisy*`)`

Description

If the dimension of the target probability distribution (or the parameter vector) is characterized as a $1 \times c$ row vector, `amcmc()` returns a matrix of draws from the distribution organized in c columns and $r = draws - burn$ rows, so each row of the returned matrix can be considered a draw from the target distribution *lnf*. Additional information about the draws is collected in three arguments overwritten by `amcmc()`: *arate*, *vals*, and *lam*, which contain actual acceptance rates, the log value of the target distribution at each draw, and λ, the proposal scaling parameters. If a Metropolis-within-Gibbs sampler or a block sampler is used, *lam*, as well as *arate*, is returned as a row vector equal in length to the dimension of the distribution or the number of blocks.

Information about how to draw from the target distribution and how the distribution has been programmed is passed to the command as a sequence of strings in the (string) row vector *alginfo*. This row vector can contain information about whether sampling is to be sequential (`mwg`), in blocks (`block`), or global (`global`). If the user is interested in applying `amcmc()` to a model statement constructed with `moptimize()` or `optimize()`, information on this and the type of evaluator function used with the model should also be contained in *alginfo*. Target distribution information can be `standalone`, `moptimize`, or `optimize`. Information on evaluator type can also be of any sort (that is, `d0`, `v0`,

4. Stata 12 is required for usage of `amcmc()`.

etc.).[5] A final option that can be passed along as part of *alginfo* is the key `fast`, which will execute the adaptive MCMC algorithm more quickly but less exactly. I give some examples of what *alginfo* might look like in the remarks about syntax.

The second argument of `amcmc()`, *lnf*, is a pointer to the target distribution, which must be written in log form. *xinit* and *Vinit* are conformable initial values for the routine and an initial variance–covariance matrix for the proposal distribution. The scalar *draws* and *burn* tell the routine how many draws to make from the distribution and how many of these draws are to be discarded as an initial burn-in period. *delta* is a string scalar that describes how adaptation is to occur, while *aopt* is the desired acceptance rate; see section 2.1.

The real matrix *blocks* contains information on how `amcmc()` should proceed if the user wishes to draw from the function in blocks. If the user does not wish to draw in blocks, the user simply passes a missing value for this argument. If the user provides an argument here, but does not specify `block` as part of *alginfo*, sampling will not occur in blocks.

If the user is drawing from a function constructed with a prespecified model command written to work with either `moptimize()` or `optimize()`, this model statement is passed to `amcmc()` via the optional *M* argument. As described below, this argument can also have other uses; for example, it can pass up to 10 additional explanatory variables to `amcmc()`.

The final option is *noisy*, and if the user specifies *noisy*=`"noisy"`, `amcmc()` will produce feedback on drawing as the algorithm executes. A dot is produced every time the evaluation function *lnf* is called (not every time a "draw" is completed, because the latter is taken by `amcmc()` to mean a complete run through the routine). Thus, if a block sampler or a Metropolis-within-Gibbs style sampler is used, a draw is deemed to have occurred when all the blocks or variables have been drawn once. The value of the target distribution is reported every 50 evaluations.

Remarks

It is helpful to have a few examples of how information about the draws to be conducted can be passed to the `amcmc()` function through the first argument, *alginfo*. This is described in table 4.

Table 4. Options for using `amcmc()`, passed in the argument *alginfo*

Sampling information	`mwg`, `global`, `block`
Model definition	`moptimize`, `optimize`, `standalone`
Evaluator type	`d*`, `q*`, `e*`, `g*`, `v*`
Other information	`fast`

5. The routine will not work with evaluators of the `lnf` type.

The user can select any item from each of the rows on table 4 and pass it to `amcmc()` as part of *alginfo*. For example, if the user is trying to draw from a function that was written as a type `d2` evaluator to work with `moptimize` and the user wished to use a global sampler, he or she might specify

```
alginfo="moptimize","d2","global"
```

Order does not matter, so the user could also specify

```
alginfo="d0","moptimize","global"
```

If the user had a stand-alone function and wished to do Metropolis-within-Gibbs style sampling from this function, he or she would specify

```
alginfo="standalone","mwg"
```

or even just `alginfo="mwg"` because if no model statement is submitted, `amcmc()` will assume that the function is stand alone. The final option that the user might specify is the `"fast"` option, which tacks on the string `fast` to *alginfo*. This option is helpful when the user wishes to sample globally or in blocks but has a problem with large dimension. Because the global and block samplers use Cholesky decomposition of the proposal covariance matrix, large problems may be time consuming. The `"fast"` option circumvents the potential slowdown by working with just the diagonal elements of the proposal covariance matrix, so one can avoid Cholesky decomposition. One should, however, be cautious in using this option and should probably apply it only when the user can be reasonably certain that distribution variables are independent.[6]

The row vector *xinit* contains an initial value for the draws, while *Vinit* is an initial variance–covariance matrix that may be a conformable identity matrix. If, however, *Vinit* is a row vector, `amcmc()` will interpret this as the diagonal of a variance matrix with zero off-diagonal entries.

While the user-specified scalar *delta* controls how rapidly adaptation vanishes, the user may also specify *delta* equal to missing ($delta =$ `.`). `amcmc()` will then assume that the user does not want any adaptation to occur but instead wishes to draw from the invariant proposal distribution with mean *xinit* and covariance matrix *Vinit*. In this case, the user must supply values of *lambda* to describe to the algorithm how to scale draws from the proposal distribution. Constructing the code this way allows users to run the adaptive algorithm for a while, and once it has converged, it allows users to switch to an algorithm using an invariant proposal distribution. If a global sampler is used, only one value of *lambda* is required; otherwise, *lambda* must be conformable with the sampler. So, if the option `mwg` is used, the dimension of *lambda* must match the dimension of the target distribution; if the option `block` is used, *lambda* must contain as many entries as the number of blocks.

Whether one wishes to do Metropolis-within-Gibbs sampling, block sampling, or global sampling, the routine requires the same set of input information (although the

6. I included this option hoping that users might try it and see for what problems, if any, it does and does not work well.

overwritten values *lam* and *arate* differ slightly) with one exception. When one samples in block form, amcmc() requires a matrix to be provided in *block*, in which the number of rows is equal to the number of sampling groups, and the values to be drawn together have 1s in the appropriate positions and 0s elsewhere. So, for example, if one wished to draw from a five-dimensional distribution and wished to draw values for the first three arguments together, and then arguments four and five together, one would set up a matrix **B** as follows:

$$\mathbf{B} = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

One can also pass an identity matrix as a block matrix:

$$\mathbf{B} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

One might suspect that this would result in the same sort of algorithm obtained by specifying *alginfo*="mwg", but this is not the case. After each draw, the block algorithm updates the *entire* mean proposal vector and covariance matrix, so information on each draw is used to prepare for the next.[7] While not the intended use of the block-sampling algorithm, if one leaves a column of all 0s in the matrix **B**, the corresponding value of the parameter will never be drawn. This is a quick, albeit not particularly efficient, way of constraining parameters at particular values during the drawing process.

The argument *M* of amcmc() can contain a previously assembled model statement, or it can be used to pass additional arguments of a function to the routine.[8] For example, if the user has written a function to be sampled from that has three arguments, such as lnf(x,Y,Z), the user would specify the standalone option in the variable *alginfo*, assemble the additional arguments into a pointer, and then pass this information to amcmc(). In this instance, M might be constructed in Mata as follows:

```
M=J(2,1,NULL)
M[1,1]=&Y
M[2,1]=&Z
```

M can then be passed to amcmc(), which will use Y and Z (in order) to evaluate lnf(x,Y,Z). As shown in the examples, this usage of pointers can be handy when amcmc() is used as part of a larger algorithm: one can continually change Y and Z without actually having to explicitly declare that Y and Z have changed as the algorithm executes.

7. Using amcmc() in this way is akin to what Andrieu and Thoms (2008, 360) describe as an adaptive MCMC algorithm with “componentwise adaptive scaling”.

8. But not both; we assume that any arguments have already been built into the model statement if a previously constructed model is used.

3.2 Adaptive MCMC via a structure

Syntax

Another alternative that has advantages in certain situations, particularly when one wishes to do adaptive MCMC as one step in a larger sampling problem, is to set up an adaptive MCMC sampling problem by using the set of functions `amcmc_*()`. The user first opens a problem using the `amcmc_init()` function and then fills in the details of the drawing procedure. The user can use the following functions to set up an adaptive MCMC problem, with the arguments corresponding to those described in section 3.1:

A = `amcmc_init()`

`amcmc_lnf(`*A*`,` *pointer (real scalar function) scalar f*`)`

`amcmc_args(`*A*`,` *pointer matrix Z*`)`

`amcmc_xinit(`*A*`,` *real rowvector xinit*`)`

`amcmc_Vinit(`*A*`,` *real matrix Vinit*`)`

`amcmc_aopt(`*A*`,` *real scalar aopt*`)`

`amcmc_blocks(`*A*`,` *real matrix blocks*`)`

`amcmc_model(`*A*`,` *transmorphic M*`)`

`amcmc_noisy(`*A*`,` *string scalar noisy*`)`

`amcmc_alginfo(`*A*`,` *string rowvector alginfo*`)`

`amcmc_damper(`*A*`,` *real scalar delta*`)`

`amcmc_lambda(`*A*`,` *real rowvector lambda*`)`

`amcmc_draws(`*A*`,` *real scalar draws*`)`

`amcmc_burn(`*A*`,` *real scalar burn*`)`

Once a problem has been specified, a run can be initiated via the function

`amcmc_draw(`*A*`)`

Results can be accessed via a series of functions of the form

`amcmc_results_*(`*A*`)`

where * in the above function can be any of the following: `vals`, `arate`, `passes`, `totaldraws`, `acceptances`, `propmean`, `propvar`, or `report`. Additionally, users can recover their initial specifications by using * = `draws`, `aopt`, `alginfo`, `noisy`, `blocks`, `damper`, `xinit`, `Vinit`, or `lambda`. An additional function `amcmc_results_lastdraw()` produces the value of only the last draw. Two other functions that are useful when one is executing an adaptive MCMC draw as part of a larger algorithm are `amcmc_append(`*A*`,` *string scalar append*`)`

`amcmc_reeval(`*A*`,` *string scalar reeval*`)`

The function `amcmc_append()` allows the user to indicate that results should be overwritten by specifying *append*=`"overwrite"`. In this case, the results of only the most recent draws are kept. This can be useful when doing an analysis where nuisance parameters of a model are being drawn, and storing all the previous draws would tax the memory and impact the speed of the algorithm's operation. The function `amcmc_reeval()` allows the user to indicate whether the target distribution should be reevaluated at the last draw before a proposed value is tried by specifying *reeval*=`"reeval"`. When the draw is part of a larger algorithm, some of the arguments of the target distribution might change as the larger algorithm proceeds. In these cases, the target distribution needs to be reevaluated at the new argument values and the last previous draw to function correctly. If the user sets *reeval* to anything else, it is assumed that nothing has changed and that the value of the target distribution has not changed between draws.

Remarks

Some of the information accessible with `amcmc_results_*()` provides hints as to why a user might prefer to use a problem statement to attack an adaptive MCMC problem instead of the Mata function `amcmc()`. Using a problem statement is particularly useful because one can easily stop, restart, and append a run within Mata's structure environment. In this way, a user can perform adaptive MCMC as part of a larger algorithm; the structure makes it easy to retain information about past adaptation and runs as the algorithm proceeds and also makes it easy to modify arguments of the algorithm. In the model statement syntax, information about the number of times a given problem has been initiated is retrievable via the function `amcmc_results_passes(A)`, while the acceptance history of an entire run is accessible via `amcmc_results_acceptances(A)`.

Given the initialization of an adaptive MCMC problem `A`, one can run `amcmc_draw()` sequentially and results will be appended to previous results. Accordingly, the burn period is active only the first time the function is executed. Thereafter, it is assumed that the user wishes to retain all drawn values. As mentioned above, the user can choose whether to retain all the information about previous draws with the function `amcmc_append()`. When a user specifies *append*=`"overwrite"` to save the draws of only the last run, the routine still includes all information about adaptation contained in the entire drawing history.

When a user initializes an adaptive MCMC problem via `amcmc_init()`, some defaults are set unless overwritten by the user. The number of draws is set to 1, the burn period is set to 0, the target distribution is assumed to be stand alone, the acceptance rate is set to 0.234, and results are appended to previous results if multiple passes are made. It is also assumed that the function does not need to be reevaluated at the last value before drawing a new proposal.

Further description can be found in the help files, accessible by typing `help mata amcmc()` or `help mf_amcmc` at Stata's command prompt.

4 Examples

4.1 Parameter estimation

For my first example, I apply adaptive MCMC to a simple estimation problem. Suppose that I have already programmed a likelihood function to use with `moptimize()` in Mata, but I wish to try another means of estimating parameters—perhaps because I have found that maximization of the likelihood function is taking too long or presents other difficulties or because I am worried about small-sample properties of the estimators. I decide to try to fit the model by drawing directly from the conditional distribution of parameters. The ideas derive from Bayes's rule and the usual principles of Bayesian estimation, but they can be applied to virtually any maximum likelihood problem.[9] Via Bayes's rule, the distribution of parameters conditional on the data can be written as

$$p(\beta|X) = \frac{p(X|\beta)p(\beta)}{p(X)} = \frac{p(X|\beta)p(\beta)}{\int p(X|\beta)p(\beta)d\beta} \tag{1}$$

If one has no prior information about parameter values, one can take $p(\beta)$—the prior distribution of parameters—to be (improper) uniform over the support of the parameters. As this renders $p(\beta)$ constant, one then obtains the posterior parameter distribution as

$$p(\beta|X) \propto p(X|\beta) \tag{2}$$

So, according to (2), one might interpret a likelihood function as the distribution of parameters conditional on data up to a constant of proportionality. The conditional mean of parameter values is then

$$E(\beta|X) = \int \beta p(\beta|X)d\beta \tag{3}$$

One can estimate $E(\beta|X)$ by simulating the right-hand side of (3) via S draws from the conditional distribution $p(\beta|X)$,

$$E(\beta|X) \approx= \frac{1}{S}\sum_{s=1}^{S}\beta_{(s)}$$

These simulations can also be used to characterize higher-order moments of the parameter distribution. I shall follow the nomenclature adopted by Chernozhukov and Hong (2003) and refer to obtained estimators as Laplace-type estimators (LTEs) or quasi-Bayesian estimators (QBEs).

Returning to the example, I will posit a simple linear model with log-likelihood function,

$$\ln L \propto \frac{(y-X\beta)'(y-X\beta)}{2\sigma^2} - \frac{n}{2}\ln\sigma^2$$

9. They can also be applied to a wider variety of problems; see Chernozhukov and Hong (2003).

For comparison, in the following code, I take this simple model and fit it to some data by using a type `d0` evaluator and Mata's `moptimize()` function. One subtlety of the code is that the variance is coded in exponentiated form. This is done so that when `amcmc()` is applied to the problem, the objective function is consistent with the multivariate normal proposal distribution, which requires that parameters have support $(-\infty, \infty)$.[10] The following code develops the model statement and fits the model via maximum likelihood:

```
. sysuse auto
(1978 Automobile Data)
. mata:
------------------------------------------------- mata (type end to exit) ------
: function lregeval(M,todo,b,crit,s,H)
> {
> real colvector p1, p2
> real colvector y1
>         p1=moptimize_util_xb(M,b,1)
>         p2=moptimize_util_xb(M,b,2)
>         y1=moptimize_util_depvar(M,1)
>         crit=-(y1:-p1)´*(y1:-p1)/(2*exp(p2))-
>                 rows(y1)/2*p2
> }
note: argument todo unused
note: argument s unused
note: argument H unused
: M=moptimize_init()
: moptimize_init_evaluator(M,&lregeval())
: moptimize_init_evaluatortype(M,"d0")
: moptimize_init_depvar(M,1,"mpg")
: moptimize_init_eq_indepvars(M,1,"price weight displacement")
: moptimize_init_eq_indepvars(M,2,"")
: moptimize(M)
initial:       f(p) =     -18004
alternative:   f(p) = -10466.142
rescale:       f(p) = -298.60453
rescale eq:    f(p) = -189.39334
Iteration 0:   f(p) = -189.39334  (not concave)
Iteration 1:   f(p) = -172.06827  (not concave)
Iteration 2:   f(p) = -162.08563  (not concave)
Iteration 3:   f(p) = -156.61996  (not concave)
Iteration 4:   f(p) = -143.55991
Iteration 5:   f(p) = -129.10949
Iteration 6:   f(p) = -127.05705
Iteration 7:   f(p) = -127.05447
Iteration 8:   f(p) = -127.05447
```

10. A less efficient way to deal with parameters with restricted supports is to program the distribution so that it returns a missing value whenever a draw lands outside the appropriate range.

```
: moptimize_result_display(M)
                                                Number of obs   =         74

------------------------------------------------------------------------------
         mpg |      Coef.   Std. Err.      z    P>|z|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
eq1          |
       price |  -.0000966   .0001591    -0.61   0.544    -.0004085    .0002153
      weight |  -.0063909   .0011759    -5.43   0.000    -.0086956   -.0040862
displacement |   .0054824   .0096492     0.57   0.570    -.0134296    .0243945
       _cons |   40.10848   1.974222    20.32   0.000     36.23907    43.97788
-------------+----------------------------------------------------------------
eq2          |
       _cons |   2.433905    .164399    14.80   0.000     2.111688    2.756121
------------------------------------------------------------------------------

: end
--------------------------------------------------------------------------------
```

I now estimate model parameters via simulation by treating the likelihood function like the parameters' conditional distribution. I start with a Metropolis-within-Gibbs sequential sampler to obtain 10,000 draws for each parameter value, discarding the first 20 draws as a burn-in period. I start with this sampler because it is usually a relatively safe choice when there is little information on starting points, which I am pretending are unavailable. I set the initial values used by the sampler to 0 and use an identity matrix as an initial covariance matrix for proposals. I choose a value of *delta* = 2/3, which allows a fairly conservative amount of adaptation to occur and a desired acceptance rate of 0.4.[11]

```
. set seed 8675309

. mata:
------------------------------------------------- mata (type end to exit) ------
: alginfo="moptimize","d0","mwg"

: b_mwg=amcmc(alginfo,&lregeval(),J(1,5,0),I(5),10000,50,2/3,.4,
> arate=.,vals=.,lambda=.,.,M)

: st_matrix("b_mwg",mean(b_mwg))

: st_matrix("V_mwg",variance(b_mwg))

: end
--------------------------------------------------------------------------------

. local names eq1:price eq1:weight eq1:displacement eq1:_cons eq2:_cons

. matrix colnames b_mwg=`names´

. matrix colnames V_mwg=`names´

. matrix rownames V_mwg=`names´

. ereturn post b_mwg V_mwg
```

11. Regarding what might seem a relatively short burn-in period, I set this period to be short enough to show the convergence behavior of the algorithm.

```
. ereturn display
------------------------------------------------------------------------------
             |      Coef.   Std. Err.      z    P>|z|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
eq1          |
       price |  -.0001322   .0001714    -0.77   0.440    -.0004681    .0002036
      weight |  -.0057418   .0018016    -3.19   0.001     -.009273   -.0022107
displacement |     .00218   .0125846     0.17   0.862    -.0224854    .0268454
       _cons |   39.00328   3.095009    12.60   0.000     32.93717    45.06939
-------------+----------------------------------------------------------------
eq2          |
       _cons |   2.518081   .2071915    12.15   0.000     2.111993    2.924169
------------------------------------------------------------------------------
```

Although the algorithm was not allowed a very long burn-in time, the simulation-based parameter estimates are close to those obtained by maximum likelihood.[12] How frequently were draws of each parameter accepted, and how close is the algorithm working around the maximum value of the function? This information is returned as the overwritten arguments *arate* and *vals*.

```
. mata:
------------------------------------------------- mata (type end to exit) ------
: arate´
                  1
    +---------------+
  1 |  .3806030151  |
  2 |  .3807035176  |
  3 |  .3870351759  |
  4 |  .4020100503  |
  5 |  .3951758794  |
    +---------------+

: max(vals),mean(vals)
                   1              2
    +-------------------------------+
  1 |  -127.1097198   -130.2193494  |
    +-------------------------------+

: end
--------------------------------------------------------------------------------
```

The sampler finds and operates close to the maximum value of the log likelihood (which was −127.05), and the acceptance rates of the draws are very close to the desired acceptance rate of 0.4. To understand what the distribution of the parameters looks like, I pass the information about parameter draws to Stata and form visual pictures of results. The code below accomplishes this and creates two panels of graphs: one that shows the distribution of parameters (figure 1) and one that shows how parameter draws and the value of the function evolved as the algorithm moved (figure 2).

12. One possible issue here is whether it is appropriate to summarize the results in usual Stata format like this. One can assume that this is acceptable here because the parameters are collectively normally distributed. Whether this is true in more general problems requires careful thought.

```
. preserve

. clear

. local varnames price weight displacement constant std_dev

. getmata (`varnames´)=b_mwg

. getmata vals=vals

. generate t=_n

. local graphs

. local tgraphs

. foreach var of local varnames {
  2.     quietly {
  3.          histogram `var´, saving(`var´, replace) nodraw
  4.          twoway line `var´ t, saving(t`var´, replace) nodraw
  5.     }
  6.     local graphs "`graphs´ `var´.gph"
  7.     local tgraphs "`tgraphs´ t`var´.gph"
  8. }

. histogram vals, saving(vals,replace) nodraw
(bin=39, start=-183.40158, width=1.4433811)
(file vals.gph saved)

. twoway line vals t, saving(vals_t,replace) nodraw
(file vals_t.gph saved)

. graph combine `graphs´ vals.gph

. graph export vals_mwg.eps, replace
(file vals_mwg.eps written in EPS format)

. graph combine `tgraphs´ vals_t.gph

. graph export valst_mwg.eps, replace
(file valst_mwg.eps written in EPS format)

. restore
```

Figure 1 is composed of histograms for each parameter, with the last panel being the histogram of the log likelihood. Parameters seem to be approximately normally distributed (with a few blips), excepting the first few draws, and they are also centered around parameter values obtained via maximum likelihood.

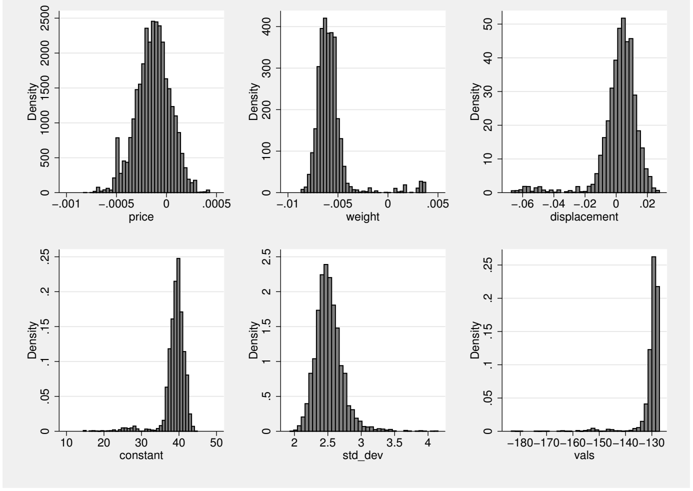

Figure 1. The distribution of the parameters after an MCMC run

Figure 2 shows how the drawn values for parameters and the value of the objective function evolved as the algorithm proceeded.

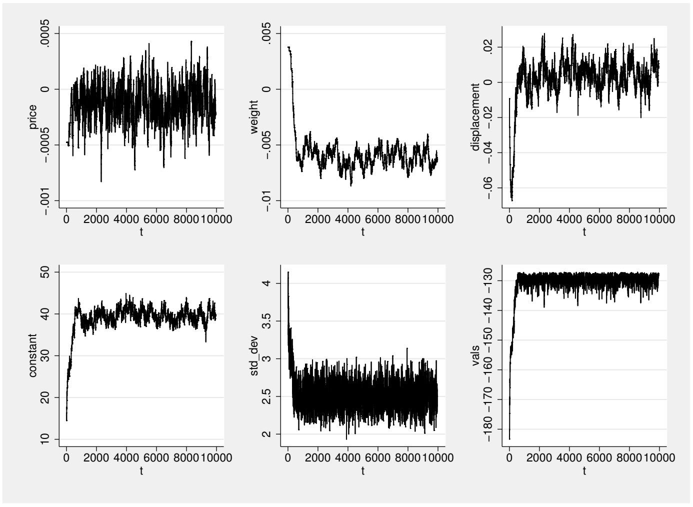

Figure 2. A look at the estimates

From figure 2, one can see that after a few iterations, the algorithm settles down to drawing from an appropriate range. The draws are also autocorrelated, and this autocorrelation is a general property of any MCMC algorithm, adaptive or not. Thus, when one applies MCMC algorithms in practice, it is sometimes beneficial to thin out the draws by keeping, say, only every 5th or 10th draw or to jumble draws.

To illustrate the use of a global sampler and some of the problems one might encounter in an MCMC-based analysis, I now apply a global sampler to the problem so that all parameter values are drawn simultaneously. The following code shows the results of a run of 12,000 draws with a burn-in period of 2,000:

```
. set seed 8675309
. mata:
-------------------------------------------------- mata (type end to exit) ------
: alginfo="global","d0","moptimize"
: b_glo=amcmc(alginfo,&lregeval(),J(1,5,0),I(5),12000,2000,2/3,.4,
> arate=.,vals=.,lambda=.,.,M)
: st_matrix("b_glo",mean(b_glo))
: st_matrix("V_glo",variance(b_glo))
: end
---------------------------------------------------------------------------------
. local names eq1:price eq1:weight eq1:displacement eq1:_cons eq2:_cons
. matrix colnames b_glo=`names´
. matrix colnames V_glo=`names´
. matrix rownames V_glo=`names´
. ereturn post b_glo V_glo
. ereturn display
------------------------------------------------------------------------------
             |      Coef.   Std. Err.      z    P>|z|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
eq1          |
       price |  -.0004614   .0019104    -0.24   0.809    -.0042057    .0032829
      weight |    .013056   .0232029     0.56   0.574    -.0324209    .0585328
displacement |  -.1798405   .3163187    -0.57   0.570    -.7998138    .4401328
       _cons |   15.16227   20.84814     0.73   0.467    -25.69933    56.02387
-------------+----------------------------------------------------------------
eq2          |
       _cons |   4.017751   1.880026     2.14   0.033     .3329679    7.702533
------------------------------------------------------------------------------
```

One can see from these results that the algorithm has not quickly found an appropriate range of values for parameter values. Figures 3 and 4 indicate why—the algorithm spends considerable time stuck away from the maximal function value.

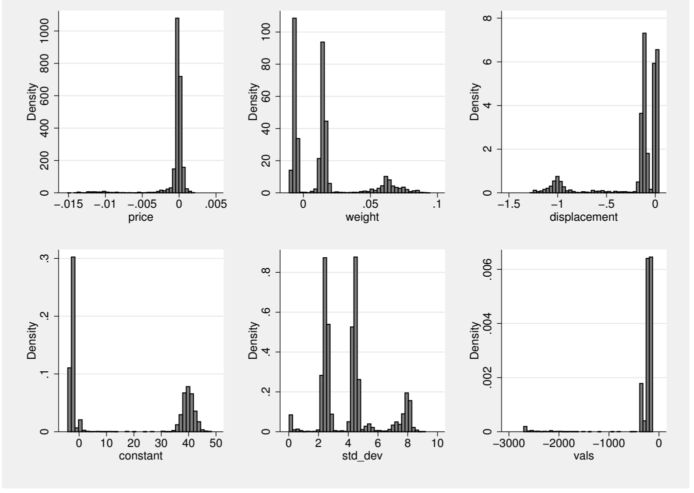

Figure 3. Distribution of parameters after a global MCMC run that is slow to converge

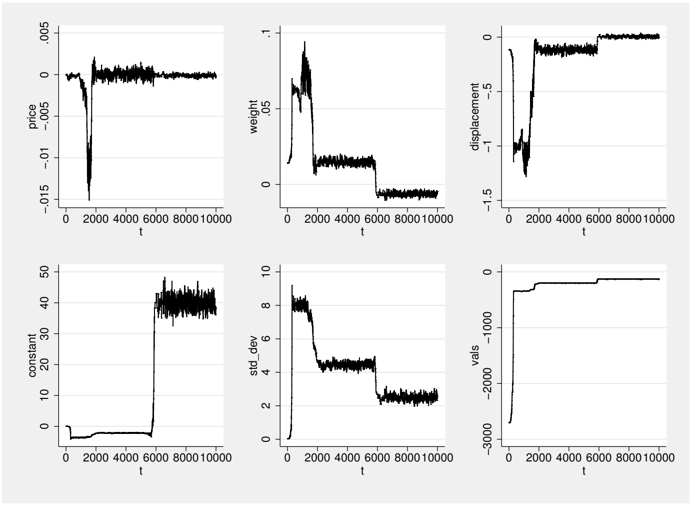

Figure 4. Characteristics of draws after a global MCMC run

The problem observed in figures 3 and 4 is that the algorithm was not allowed to burn in for a long enough time for the global MCMC algorithm to work correctly. While the parameter values eventually settled down closer to their "true" values, it took the algorithm upward of 6,000 draws to find the right range. In fact, it looks as though the algorithm settled into a stable range for draws 2,000–6,000 or so but then once again experienced a jump to the correct stable range, a phenomenon known as "pseudoconvergence" (Geyer 2011). This behavior is also responsible for the multimodal appearance of the histograms on figure 3.

While my intent is to illustrate how the Mata function `amcmc()` works, my example also illustrates what can happen when one fails to specify appropriate adjustment parameters and does not allow an adaptive MCMC algorithm to run long enough in a given estimation problem. One may unknowingly get bad results, as the case would be if the global algorithm had been allowed to run for only 5,000 iterations. This sometimes happens if poor starting values are mixed with parameters that have very different magnitudes, for example, the constant in the initial model relative to the other parameters. From inspecting figure 3, one can see that the constant did not find its correct range until just after 6,000 draws, and this is likely what caused the problem.

This discussion motivates using `amcmc()` in steps, where a slower but relatively robust sampler (a Metropolis-within-Gibbs sampler, in this case) is used to orient parameters close to their correct range before a global sampler is used, as shown in the following code:

```
. mata:
─────────────────────────────────────────────────── mata (type end to exit) ────
: alginfo="mwg","d0","moptimize"
: b_start=amcmc(alginfo,&lregeval(),J(1,5,0),I(5),5*1000,5*100,2/3,.4,
> arate=.,vals=.,lambda=.,.,M)
: alginfo="global","d0","moptimize"
: b_glo2=amcmc(alginfo,&lregeval(),mean(b_start),
> variance(b_start),11000,1000,2/3,.4,
> arate=.,vals=.,lambda=.,.,M)
: st_matrix("b_glo2",mean(b_glo2))
: st_matrix("V_glo2",variance(b_glo2))
: end
─────────────────────────────────────────────────────────────────────────────────
. local names eq1:price eq1:weight eq1:displacement eq1:_cons eq2:_cons
. matrix colnames b_glo2=`names´
. matrix colnames V_glo2=`names´
. matrix rownames V_glo2=`names´
. ereturn post b_glo2 V_glo2
. ereturn display
──────────────────────────────────────────────────────────────────────────────
             │      Coef.   Std. Err.      z    P>|z|     [95% Conf. Interval]
─────────────┼────────────────────────────────────────────────────────────────
eq1          │
       price │  -.0001059   .0001584    -0.67   0.504    -.0004164    .0002046
      weight │  -.0063727   .0012014    -5.30   0.000    -.0087275   -.0040179
displacement │   .0056462   .0099215     0.57   0.569    -.0137997     .025092
       _cons │   40.10216   1.912111    20.97   0.000     36.35449    43.84982
─────────────┼────────────────────────────────────────────────────────────────
eq2          │
       _cons │   2.480892   .1665249    14.90   0.000      2.15451    2.807275
──────────────────────────────────────────────────────────────────────────────
```

Thus one can then draw parameters that are scaled differently either alone or in blocks until the algorithm finds it footing, and then proceed with a global algorithm. I have motivated the use of a global drawing method because of its clear speed advantages, but another more subtle reason to use it that might not be obvious when visually inspecting the graphs is that global draws often exhibit less serial correlation across draws.[13] The conclusion provides sources with additional tips for setting up, analyzing, and presenting the results of an MCMC run.

Yet another alternative is to once again begin with a Metropolis-within-Gibbs sampler to characterize the distribution of the parameters and, once this is done sufficiently well, to run the algorithm without adaptation so that one is using an invariant proposal distribution and a regular MCMC algorithm. After an initial run with the `"mwg"` option, I submit the mean and variance of results to the global sampler with no adaptation parameter, passing a value of missing `(.)` for *delta*. Because I am not passing any information to `amcmc()` on how to do adaptation in this case, I am required to submit a value for lambda, so I choose $\lambda = 2.38^2/n$.[14] Finally, I also submit a missing value for *aopt*. Because no adaptation occurs, *aopt* is not used by the algorithm.

```
. mata:
─────────────────────────────────────────────────── mata (type end to exit) ───────
: alginfo="mwg","d0","moptimize"

: b_start=amcmc(alginfo,&lregeval(),J(1,5,0),I(5),5*1000,5*100,2/3,.4,
> arate=.,vals=.,lambda=.,.,M)

: alginfo="global","d0","moptimize"

: b_glo3=amcmc(alginfo,&lregeval(),mean(b_start),
> variance(b_start),10000,0,.,.,
> arate=.,vals=.,(2.38^2/5),.,M)

: arate´
  .2253

: mean(b_glo3)´
                   1
    ┌────────────────┐
  1 │  -.0000916295  │
  2 │  -.0064095109  │
  3 │   .0054916501  │
  4 │   40.14276799  │
  5 │   2.497166774  │
    └────────────────┘

: end
──────────────────────────────────────────────────────────────────────────────────
```

Apparently, the proposal distribution was successfully tuned in the initial run with the Metropolis-within-Gibbs sampler. The mean values of the parameters obtained from the global draw are close to their maximum-likelihood values, and the acceptance rate is in the healthy range.

13. I thank an anonymous referee for pointing this out.
14. Note that I did not retain and submit the values of *lambda* from the initial run—this is because the global sampler requires a scalar value for *lambda*, while the Metropolis-within-Gibbs run returns a vector of values overwritten in *lambda*.

I could have also set up this problem using a structure as follows:

```
. mata:
─────────────────────────────────────────────── mata (type end to exit) ───
: A=amcmc_init()
: amcmc_alginfo(A,("global","d0","moptimize"))
: amcmc_lnf(A,&lregeval())
: amcmc_xinit(A,J(1,5,0))
: amcmc_Vinit(A,I(5))
: amcmc_model(A,M)
: amcmc_draws(A,4000)
: amcmc_damper(A,2/3)
: amcmc_draw(A)
: end
───────────────────────────────────────────────────────────────────────────
```

I can now access results using the previously described `amcmc_results_*(A)` set of functions.

4.2 Censored quantile regression

While the previous example demonstrated the basic principles and how one might apply adaptive MCMC in problems of parameter estimation, the example did not show how the methods might work when the usual maximization-based techniques fail. Chernozhukov and Hong (2003) use as an example censored quantile regression originally developed in Powell (1984) and extended in Powell (1986), which, as Chernozhukov and Hong (2003, 296) note, provides a way to do "valid inference in Tobin–Amemiya models without distributional assumptions and with heteroskedasticity of unknown form". Unfortunately, the model is hard to handle with the usual methods. The objective function is

$$L_n(\theta) = -\sum_{i}^{n} \rho_\tau\{Y_i - \max(c_i, X_i\beta)\} \tag{4}$$

where c_i in (4) denotes a (left) censoring point that might be specific to the ith observation, and $\rho_\tau(u) = \{\tau - (\mathbf{1}(u < 0)\}u$. $\tau \in (0,1)$ is the quantile of interest. Estimation using derivative-based maximization methods is problematic because the objective function (4) has flat regions and discontinuities. While one might do well with a nonderivative-based optimization method such as Nelder–Mead, one is then confronted with the problem of characterizing the parameters' distribution and getting standard errors. For these reasons, one might opt for an LTE or a QBE estimator.

To apply amcmc() to the problem, I first program the objective function as follows:[15]

```
. mata:
------------------------------------------------- mata (type end to exit) ------
: void cqregeval(M,todo,b,crit,g,H) {
>         real colvector u,Xb,y,C
>         real scalar tau
>
>         Xb    =moptimize_util_xb(M,b,1)
>         y     =moptimize_util_depvar(M,1)
>         tau   =moptimize_util_userinfo(M,1)
>         C     =moptimize_util_userinfo(M,2)
>         u     =(y:-rowmax((C,Xb)))
>         crit  =-colsum(u:*(tau:-(u:<0)))
> }
note: argument todo unused
note: argument g unused
note: argument H unused
: end
--------------------------------------------------------------------------------
```

The following code sets up a model statement for use with the function moptimize() (see [M-5] **moptimize()**). One can follow the Mata code with moptimize(M) to verify that this model and variations on the basic theme, obtained by dropping or adding additional variables, encounter difficulties.

```
. webuse laborsub, clear
. gen censorpoint=0
. mata:
------------------------------------------------- mata (type end to exit) ------
: M=moptimize_init()
: moptimize_init_evaluator(M,&cqregeval())
: moptimize_init_depvar(M,1,"whrs")
: moptimize_init_eq_indepvars(M,1,"kl6 k618 wa")
: tau=.6
: moptimize_init_userinfo(M,1,tau)
: st_view(C=.,.,"censorpoint")
: moptimize_init_userinfo(M,2,C)
: moptimize_init_evaluatortype(M,"d0")
: end
--------------------------------------------------------------------------------
```

15. One might code the objective function without summing over observations. I sum over observations so that the objective is compatible with Nelder–Mead in Stata, which requires a type d0 evaluator.

Setting up the problem like this allows the use of `amcmc()`, where I implement the strategy of using a Metropolis-within-Gibbs-type algorithm followed by a global sampler.

```
. mata:
─────────────────────────────────────────────────── mata (type end to exit) ─────
: alginfo="mwg","d0","moptimize"
: b_start=amcmc(alginfo,&cqregeval(),J(1,4,0),I(4),5000,1000,2/3,.4,
> arate=.,vals=.,lambda=.,.,M)
: alginfo="global","d0","moptimize"
: b_end=amcmc(alginfo,&cqregeval(),mean(b_start),
> variance(b_start),20000,10000,1,.234,arate=.,vals=.,lambda=.,.,M)
: end
──────────────────────────────────────────────────────────────────────────────────
```

Because this application might be of more general interest, I developed the command `mcmccqreg`, which is a wrapper for the LTE and QBE estimation of censored quantile regression. The previous code can be executed by the command.

```
. set seed 584937
. quietly mcmccqreg whrs kl6 k618 wa, tau(.6) sampler("mwg") draws(5000)
> burn(1000) dampparm(.667) arate(.4) censorvar(censorpoint)
. matrix binit=e(b)
. matrix V=e(V)
. mcmccqreg whrs kl6 k618 wa, tau(.6) sampler("global") draws(20000)
> burn(10000) arate(.234) saving(lsub_draws) replace
> from(binit) fromv(V)

Powell´s mcmc-estimated censored quantile regression
     Observations:                 250
     Mean acceptance rate:       0.359
     Total draws:                20000
     Burn-in draws:              10000
     Draws retained:             10000
──────────────────────────────────────────────────────────────────────────────
        whrs │      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
─────────────┼────────────────────────────────────────────────────────────────
         kl6 │  -1175.389   152.6341    -7.70   0.000    -1474.583   -876.1953
        k618 │  -171.3108   23.75806    -7.21   0.000    -217.8814   -124.7402
          wa │  -29.23027   10.74507    -2.72   0.007    -50.29276   -8.167779
       _cons │   2638.497   500.8126     5.27   0.000     1656.804    3620.191
──────────────────────────────────────────────────────────────────────────────
 Value of objective function:
            Mean:            -89298.96
            Min:             -89295.52
            Max:             -89308.58
   Draws saved in: lsub_draws

    *Results are presented to conform with Stata covention, but
     are summary statistics of draws, not coefficient estimates.
```

One can see from the way the command is issued how information about the sampler, the drawing process, and the censoring point (which has default of 0 for all observations) can be controlled using the `mcmccqreg` command. The command produces "estimates" that are summary statistics of the sampling run. `mcmccqreg` allows one to save results, and the results of the run are saved in the file `lsub_draws` with the objective function value after each draw. The user can then easily analyze the draws using Stata's graphing and statistical analysis tools. While the workings of the command derive more or less directly from the description of `amcmc()`, more information about the command and some additional examples can be found in the `mcmccqreg`'s help file.

4.3 Drawing from a distribution

I now show how to use `amcmc()` to draw from a distribution. Suppose that I have developed a theory that says three variables are jointly distributed according to a distribution characterized by

$$p(x_1, x_2, x_3) \propto \exp\left\{-x_1^2 - 0.5x_2^2 + x_1x_2 - 0.05(x_3 - 100)^2\right\}$$

As written, p does not integrate to one and seems hard to invert. While Metropolis-within-Gibbs or global sampling works fine with this example, to illustrate the block sampler, I will draw from the distribution in blocks, where values for the first two arguments are drawn together, followed by a draw of the third. Thus the block matrix to be passed to `amcmc()` is

$$\mathbf{B} = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

The code that programs the function and draws from the distribution is as follows:

```
. set seed 262728
. mata:
------------------------------------------------- mata (type end to exit) ------
: real scalar ln_fun(x)
> {
>         return(-x[1]^2-1/2*x[2]^2+x[1]*x[2]-.05*(x[3]-100)^2)
> }
: B=(1,1,0) \ (0,0,1)
: alginfo="standalone","block"
: x_block=amcmc(alginfo,&ln_fun(),J(1,3,0),I(3),4000,200,2/3,.4,
> arate=.,vals=.,lambda=.,B)
: end
--------------------------------------------------------------------------------
```

The example is set up to draw 4,000 values with a burn-in period of 200. Graphs of the simulation results are shown in figures 5 and 6.

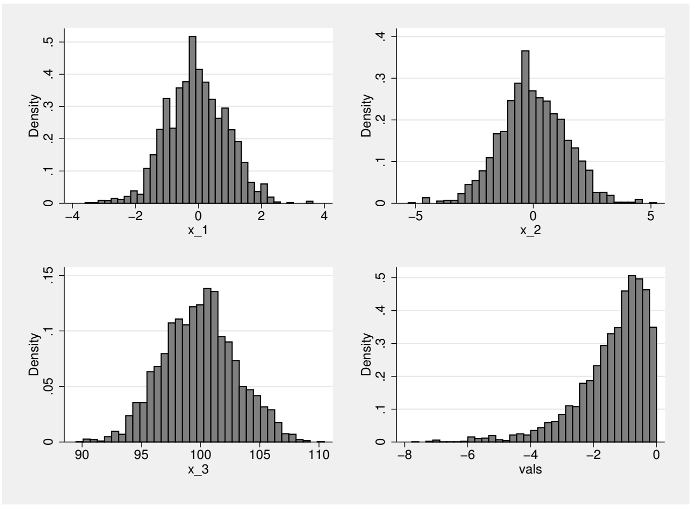

Figure 5. Draws and the log value of the distribution

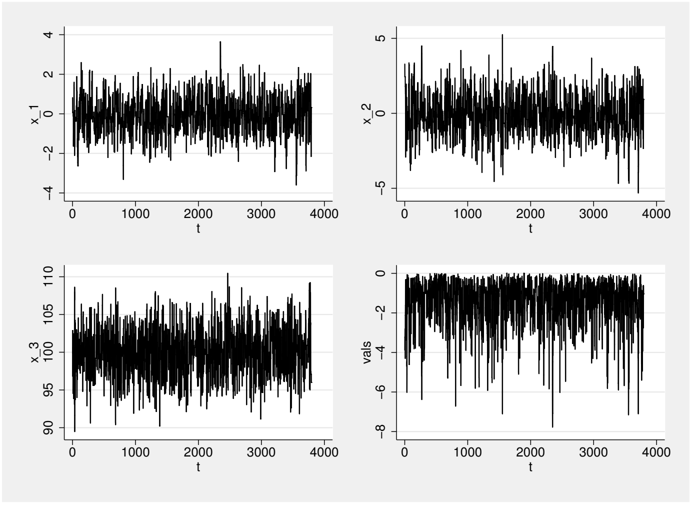

Figure 6. Behavior of draws as the algorithm proceeds

The graphs give a visual of the marginal distributions for the variables, while the time-series diagram verifies that our simulation run is getting good coverage and rapid convergence to the target distribution.

A different way to draw from this distribution would be to set up an adaptive MCMC problem via a structured set of Mata functions.

```
. mata:
─────────────────────────────────────────────── mata (type end to exit) ──────
: A=amcmc_init()
: amcmc_lnf(A,&ln_fun())
: amcmc_alginfo(A,("standalone","block"))
: amcmc_draws(A,4000)
: amcmc_burn(A,200)
: amcmc_damper(A,2/3)
: amcmc_xinit(A,J(1,3,0))
: amcmc_Vinit(A,I(3))
: amcmc_blocks(A,B)
: amcmc_draw(A)
: end
───────────────────────────────────────────────────────────────────────────────
```

4.4 Bayesian estimation of a mixed logit model

In this section, I describe the nuts and bolts of Bayesian estimation of a mixed logit model; the implementation is available via the command `bayesmixedlogit`, which I have written and made available for download. The wrapper function `bayesmixedlogit` adds some features but essentially works as described in this section.

While there is no strong reason to prefer using the `amcmc` routines as a function or a structure in the previous examples, the power and flexibility of structured objects in Mata is indispensable in this example. My exposition of the basic ideas follows Train (2009) as closely as possible, which also contains a nice overview of the principles. The example assumes that one has access to `traindata.dta`, which is used by Hole (2007) to illustrate estimation of a mixed logit model by maximum simulated likelihood.[16] The help file for `amcmc`—accessible by typing `help mata amcmc()` or `help mf_amcmc` at Stata's command prompt—describes an example that relies on data downloadable from the Stata website.

The data concern $n = 1, 2, 3, \ldots, N$ people, each of whom makes a selection from among $j = 1, 2, 3, \ldots, J$ choices on occasions $t = 1, 2, 3, \ldots, T$. For each choice made, there are a set of covariates x_{njt} that explain n's choices at t. A person's utility from the jth choice on occasion t is specified as

$$U_{njt} = \beta_n' x_{njt} + \epsilon_{njt} \tag{5}$$

16. The data are downloadable from Train's website at http://eml.berkeley.edu/∼train/ and can also be found at http://fmwww.bc.edu/repec/bocode/t/traindata.dta.

where in (5), ϵ_{njt} is an independent identically distributed extreme value, and β_n are individual-specific parameters. Variation in these parameters across the population is captured by assuming parameters normally distributed with mean b and covariance matrix $\mathbf{W}$. I denote a person's choice at t as $y_{nt} \in J$. Then the probability of observing person n's sequence of choices is

$$L(y_n|\beta) = \prod_t \frac{e^{\beta_n' x_{ny_{nt}t}}}{\sum_{j=1}^{J} e^{\beta_n' x_{njt}}} \tag{6}$$

Given the distribution of β, I can write the above conditional on the distribution of parameters, $\phi(\beta|b, \mathbf{W})$, and integrate over the distribution of parameter values to get

$$L(y_n|b, \mathbf{W}) = \int L(y_n|\beta)\phi(\beta|b, \mathbf{W})d\beta$$

In a Bayesian approach, a prior $h(b, \mathbf{W})$ is assumed, and the joint posterior likelihood of the parameters is formed using

$$H(b, \mathbf{W}|Y, X) \propto \prod_n L(y_n|b, \mathbf{W})h(b, \mathbf{W}) \tag{7}$$

Because it is difficult to compute the likelihood in (7), simulation-based methods are usually used in estimation, as in the package `mixlogit`, developed in Hole (2007).[17] An alternative is a Bayesian approach. As described by Train (2009), estimation becomes fairly easy (at least conceptually) if one breaks the problem into a sequence of conditional distributions, taking the view that each set of individual-level coefficients β_n are additional parameters to be estimated. The posterior distribution of parameters given data becomes

$$H(b, \mathbf{W}, \beta_n, n = 1, 2, 3, \ldots, N|y, X) \propto \prod_n L(y_n|\beta_n)\phi(\beta_n|b, \mathbf{W})h(b, \mathbf{W}) \tag{8}$$

Following the outline given in Train (2009, 301–302), we see that drawing from the posterior proceeds in three steps. First, b is drawn conditional on β_n and $\mathbf{W}$; then $\mathbf{W}$ is drawn conditional on b and β_n; and finally, the values of β_n are drawn conditional on b and $\mathbf{W}$. The first two steps are straightforward, assuming that the prior distribution of b is normal with extremely large variance and that the prior for $\mathbf{W}$ is an inverted Wishart with K degrees of freedom and an identity scale matrix. In this case, the conditional distribution of b is $N(\overline{\beta}, \mathbf{W}N^{-1})$, where $\overline{\beta}$ is the mean of the β_n's. The conditional distribution of $\mathbf{W}$ is an inverted Wishart with $K + N$ degrees of freedom and scale matrix $(KI + NS)/(K + N)$, where $S = N^{-1}\sum_n(\beta_n - b)(\beta_n - b)'$ is the sample variance of the β_n's about b.

The distribution of β_n given choices, data, and $(b, \mathbf{W})$ has no simple form, but from (8), we see that the distribution of a particular person's parameters obeys

$$K(\beta_n|b, \mathbf{W}, y_n, X_n) \propto L(y_n|\beta_n)\phi(\beta_n|b, \mathbf{W}) \tag{9}$$

17. From the Stata prompt, type `search mixlogit`.

where the term $L(y_n|\beta_n)$ in (9) is given by (6). This is a natural place to apply MCMC methods, and it is here where I can use the `amcmc_*()` suite of functions.

I now return to the example. `traindata.dta` contains information on the energy contract choices of 100 people, where each person faces up to 12 different choice occasions. Suppliers' contracts are differentiated by price, the type of contract offered, location to the individual, how well-known the supplier is, and the season in which the offer was made.

As a point of comparison, I fit the model in Train (2009, 305) using `mixlogit` (after download and installation).

```
. clear all
. set more off
. use http://fmwww.bc.edu/repec/bocode/t/traindata.dta
. set seed 90210
. mixlogit y, rand(price contract local wknown tod seasonal) group(gid) id(pid)
Iteration 0:   log likelihood = -1253.1345  (not concave)
Iteration 1:   log likelihood = -1163.1407  (not concave)
Iteration 2:   log likelihood = -1142.7635
Iteration 3:   log likelihood = -1123.6896
Iteration 4:   log likelihood = -1122.6326
Iteration 5:   log likelihood = -1122.6226
Iteration 6:   log likelihood = -1122.6226
Mixed logit model                                 Number of obs   =       4780
                                                  LR chi2(6)      =     467.53
Log likelihood = -1122.6226                       Prob > chi2     =     0.0000

------------------------------------------------------------------------------
           y |      Coef.   Std. Err.      z    P>|z|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
Mean         |
       price |  -.8908633   .0616638   -14.45   0.000    -1.011722   -.7700045
    contract |    -.22285   .0390333    -5.71   0.000    -.2993539   -.1463462
       local |   1.958347   .1827835    10.71   0.000     1.600098    2.316596
      wknown |   1.560163   .1507413    10.35   0.000     1.264715     1.85561
         tod |  -8.291551   .4995409   -16.60   0.000    -9.270633   -7.312469
    seasonal |  -9.108944   .5581876   -16.32   0.000    -10.20297   -8.014916
-------------+----------------------------------------------------------------
SD           |
       price |   .1541266   .0200631     7.68   0.000     .1148036    .1934495
    contract |   .3839507   .0432156     8.88   0.000     .2992497    .4686516
       local |   1.457113   .1572685     9.27   0.000     1.148873    1.765354
      wknown |  -.8979788   .1429141    -6.28   0.000    -1.178085   -.6178722
         tod |   1.313033   .1648894     7.96   0.000     .9898559     1.63621
    seasonal |   1.324614   .1881265     7.04   0.000     .9558927    1.693335
------------------------------------------------------------------------------
The sign of the estimated standard deviations is irrelevant: interpret them as
being positive
```

To implement the Bayesian estimator, I proceed in the steps outlined by Train (2009, 301–302). First, I develop a Mata function that produces a single draw from the conditional distribution of b.

```
. mata:
───────────────────────────────────────────────── mata (type end to exit) ───────
: real matrix drawb_betaW(beta,W) {
>         return(mean(beta)+rnormal(1,cols(beta),0,1)*cholesky(W)´)
> }
: end
─────────────────────────────────────────────────────────────────────────────────
```

Next I use the instructions described in Train (2009, 299) to draw from the conditional distribution of $\mathbf{W}$. The Mata function is

```
. mata
───────────────────────────────────────────────── mata (type end to exit) ───────
: real matrix drawW_bbeta(beta,b)
> {
>         v=rnormal(cols(b)+rows(beta),cols(b),0,1)
>         S1=variance(beta)
>         S=invsym((cols(b)*I(cols(b))+rows(beta)*S1)/(cols(b)+rows(beta)))
>         L=cholesky(S)
>         R=(L*v´)*(L*v´)´/(cols(b)+rows(beta))
>         return(invsym(R))
> }
: end
─────────────────────────────────────────────────────────────────────────────────
```

I now have two of the three steps of the drawing scheme in place. The last task is more nuanced and involves using structured `amcmc` problems in conjunction with the flexible ways in which one can manipulate structures in Mata. The key is to think of drawing each set of individual-level parameters β_n as a separate adaptive MCMC problem. It is helpful to first get all the data into Mata, get familiar with its structure, and then work from there.

```
. mata:
───────────────────────────────────────────────── mata (type end to exit) ───────
: st_view(y=.,.,"y")
: st_view(X=.,.,"price contract local wknown tod seasonal")
: st_view(pid=.,.,"pid")
: st_view(gid=.,.,"gid")
: end
─────────────────────────────────────────────────────────────────────────────────
```

The matrix (really, a column vector) `y` is a sequence of dummy variables marking the choices of individual n in each choice occasion, while the matrix $\mathbf{X}$ collects explanatory variables for each potential choice. `pid` and `gid` are identifiers for individuals and choice occasions, respectively. I now write a Mata function that computes the log probability for a particular vector of parameters for a given person, conditional on that person's information.

```
. mata:
─────────────────────────────────────────────── mata (type end to exit) ──────
: real scalar lnbetan_bW(betaj,b,W,yj,Xj)
> {
>         Uj=rowsum(Xj:*betaj)
>         Uj=colshape(Uj,4)
>         lnpj=rowsum(Uj:*colshape(yj,4)):-
>                 ln(rowsum(exp(Uj)))
>         var=-1/2*(betaj:-b)*invsym(W)*(betaj:-b)´-
>                 1/2*ln(det(W))-cols(betaj)/2*ln(2*pi())
>         llj=var+sum(lnpj)
>         return(llj)
> }
: end
──────────────────────────────────────────────────────────────────────────────
```

The function takes in five arguments, the first of which is a parameter vector for the person (that is, the values to be drawn). The second and third arguments characterize the mean and covariance matrix of the parameters across the population.[18] The fourth and fifth arguments contain information about an individual's choices and explanatory variables.

The first line of code multiplies parameters by explanatory variables to form utility terms, which are then shaped into a matrix with four columns. Individuals have four options available on each choice occasion. After reshaping, the utilities from potential choices on each occasion occupy a row, with separate choice occasions in columns. `lnpj` then contains the log probabilities of the choices actually made—the log of utility less the logged sum of exponentiated utilities. Finally, `var` computes the log distribution of parameters about the conditional mean, and `llj` sums the two components. The result is the log likelihood of individual n's parameter values, given choices, data, and the parameters governing the distribution of individual-level parameters.

I now set up a structured problem for each individual in the dataset. I begin by setting up a single adaptive MCMC problem and then replicate this problem using `J()` (see [M-5] **J()**) to match the number of individual-level parameter sets—the same as the number of individual-level identifiers in the data (`gid`)—characterized via Mata's `panelsetup()` (see [M-5] **panelsetup()**) function.

18. This function is not as fast as it could be, and it is also specific to the dataset. One way to speed the algorithm is to compute the Cholesky decomposition of $\mathbf{W}$ once before individual-level parameters are drawn. The wrapper `bayesmixedlogit` exploits this and a few other improvements.

```
. mata
------------------------------------------------- mata (type end to exit) ------
: m=panelsetup(pid,1)

: Ap=amcmc_init()

: amcmc_damper(Ap,1)

: amcmc_alginfo(Ap,("standalone","global"))

: amcmc_append(Ap,"overwrite")

: amcmc_lnf(Ap,&lnbetan_bW())

: amcmc_draws(Ap,1)

: amcmc_append(Ap,"overwrite")

: amcmc_reeval(Ap,"reeval")

: A=J(rows(m),1,Ap)

: end
--------------------------------------------------------------------------------
```

I also apply the `amcmc` option `"overwrite"`, which means that the results from only the last round of drawing will be saved. Specifying the `"reeval"` option means that each individual's likelihood will be reevaluated at the new parameter values and the old values of coefficients before drawing.

I now duplicate the problem by forming a matrix of adaptive MCMC problems—one for each individual—and then use a loop to fill in individual-level choices and explanatory variables as arguments. In the end, the "matrix" `A` is a collection of 100 separate adaptive MCMC problems. Before this, some initial values for `b` and `W` are set, and some initial values for individual-level parameters are drawn. I set up the pointer matrix `Args` to hold this information along with the individual-level information.

```
. mata
------------------------------------------------- mata (type end to exit) ------
: Args=J(rows(m),4,NULL)

: b=J(1,6,0)

: W=I(6)*6

: beta=b:+sqrt(diagonal(W))´:*rnormal(rows(m),cols(b),0,1)

: for (i=1;i<=rows(m);i++) {
>         Args[i,1]=&b
>         Args[i,2]=&W
>         Args[i,3]=&panelsubmatrix(y,i,m)
>         Args[i,4]=&panelsubmatrix(X,i,m)
>         amcmc_args(A[i],Args[i,])
>         amcmc_xinit(A[i],b)
>         amcmc_Vinit(A[i],W)
> }

: end
--------------------------------------------------------------------------------
```

After creating some placeholders for the draws (`bvals` and `Wvals`), we can execute the drawing algorithm as follows:

```
. mata
─────────────────────────────────────────────────── mata (type end to exit) ──────
: its=20000

: burn=10000

: bvals=J(0,cols(beta),.)

: Wvals=J(0,cols(rowshape(W,1)),.)

: for (i=1;i<=its;i++) {
>         b=drawb_betaW(beta,W/rows(m))
>         W=drawW_bbeta(beta,b)
>         bvals=bvals\b
>         Wvals=Wvals\rowshape(W,1)
>         beta_old=beta
>         for (j=1;j<=rows(A);j++) {
>                 amcmc_draw(A[j])
>                 beta[j,]=amcmc_results_lastdraw(A[j])
>         }
> }

: end
──────────────────────────────────────────────────────────────────────────────────
```

The algorithm consists of an outer loop and an inner loop, within which individual-level parameters are drawn sequentially. The current value of the `beta` vector, which holds individual-level parameters in rows, is overwritten with the last draw produced by using the `amcmc_results_lastdraw()` function.

A subtlety of the code also indicates a reason why it is useful to pass additional function arguments as pointers: each time a new value of `b` and `W` is drawn, a user does not need to reiterate to each sampling problem that `b` and `W` have changed, because pointers point to positions that hold objects and not to the values of the objects themselves. Thus, every time a new value of `b` or `W` is drawn, the arguments of all 100 problems are automatically changed. By specifying that the target distribution for each level problem is to be reevaluated, the user tells the routine to recalculate `lnbetan_bW` at the last drawn value when comparing a new draw to the previous one.

Because the technique might be of greater interest, I have developed a command that implements the algorithm `bayesmixedlogit`. For example, the algorithm described by the previous code could be executed with the following command, which also summarizes results in a way conformable with usual Stata output:

```
. set seed 475446

. bayesmixedlogit y, rand(price contract local wknown tod seasonal)
> group(gid) id(pid) draws(20000) burn(10000) samplerrand("global")
> saving(train_draws) replace

Bayesian Mixed Logit Model                        Observations    =      4780
                                                  Groups          =       100
Acceptance rates:                                 Choices         =      1195
 Fixed coefs             =                        Total draws     =     20000
 Random coefs(ave,min,max)= 0.270, 0.235, 0.289   Burn-in draws   =     10000
------------------------------------------------------------------------------
             y |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
---------------+--------------------------------------------------------------
Random         |
         price |  -1.168711   .1245738    -9.38   0.000      -1.4129   -.9245209
      contract |  -.3433208   .0682585    -5.03   0.000    -.4771212   -.2095204
         local |   2.637242   .3436764     7.67   0.000     1.963567    3.310917
        wknown |   2.138963   .2596608     8.24   0.000     1.629976    2.647951
           tod |  -11.16374   1.049769   -10.63   0.000     -13.2215   -9.105982
      seasonal |  -11.19243   1.030291   -10.86   0.000      -13.212   -9.172849
---------------+--------------------------------------------------------------
Cov_Random     |
     var_price |   .8499292   .2332495     3.64   0.000     .3927132    1.307145
cov_priceco~t  |   .1128769   .0803203     1.41   0.160     -.044567    .2703208
cov_pricelo~l  |   1.583028   .4519537     3.50   0.000     .6971079    2.468948
cov_pricewk~n  |   .8898662   .3096053     2.87   0.004     .2829775    1.496755
 cov_pricetod  |   6.106009   1.909356     3.20   0.001     2.363286    9.848731
cov_pricese~l  |   6.044055   1.892895     3.19   0.001     2.333601     9.75451
 var_contract  |   .3450904   .0670202     5.15   0.000     .2137174    .4764634
cov_contrac~l  |   .4714882   .2131141     2.21   0.027     .0537416    .8892347
cov_contrac~n  |   .3624791   .1560516     2.32   0.020     .0565865    .6683717
cov_contrac~d  |   .7592097   .6576296     1.15   0.248    -.5298765    2.048296
cov_contrac~l  |   .9147682     .65939     1.39   0.165    -.3777688    2.207305
    var_local  |   7.000292   1.883972     3.72   0.000     3.307328    10.69326
cov_localwk~n  |   4.022065   1.248119     3.22   0.001     1.575501    6.468629
 cov_localtod  |   12.84674   3.787742     3.39   0.001     5.422006    20.27148
cov_localse~l  |   13.40598   3.727253     3.60   0.000     6.099812    20.71214
   var_wknown  |   3.364285   1.012474     3.32   0.001     1.379632    5.348938
cov_wknowntod  |   6.513209    2.60766     2.50   0.013     1.401671    11.62475
cov_wknowns~l  |   7.109282   2.563623     2.77   0.006     2.084064     12.1345
      var_tod  |   57.62449   16.97876     3.39   0.001      24.3427    90.90628
cov_todseas~l  |   53.93841   16.35184     3.30   0.001     21.88551    85.99131
 var_seasonal  |   55.05572   16.54599     3.33   0.001     22.62226    87.48918
------------------------------------------------------------------------------
   Draws saved in train_draws

  *Results are presented to conform with Stata covention, but
   are summary statistics of draws, not coefficient estimates.
```

The results are similar but not identical to those obtained using `mixlogit`. Additional information and examples for `bayesmixedlogit` can be found in the help file, and some examples of estimating a mixed logit model using Bayesian methods are provided in the help file for `amcmc()`, accessible via the commands `help mf_amcmc` or `help mata amcmc()`.

5 Description

In this section, I sketch a Mata implementation of what I have been referring to as a global adaptive MCMC algorithm. The sketched routine omits a few details, mainly about parsing options, but it is relatively true to form in describing how the algorithms discussed in the article are actually implemented in Mata and might be used as a template for developing more specialized algorithms. It assumes that the user wishes to draw from a stand-alone function without additional arguments. The code is as follows:

```
. mata:
------------------------------------------------- mata (type end to exit) ------
: real matrix amcmc_global(f,xinit,Vinit,draws,burn,damper,
>                                             aopt,arate,val,lam)
> {
>         real scalar nb,old,pro,i,alpha
>         real rowvector xold,xpro,mu
>         real matrix Accept,accept,xs,V,Vsq,Vold
>
>         nb=cols(xinit)  /* Initialization */
>         xold=xinit
>         lam=2.38^2/nb
>         old=(*f)(xold)
>         val=old
>
>         Accept=0
>         xs=xold
>         mu=xold
>         V=Vinit
>         Vold=I(cols(xold))
>
>         for (i=1;i<=draws;i++) {
>                 accept=0
>                 Vsq=cholesky(V)'    /* Prep V for drawing */
>                 if (hasmissing(Vsq)) {
>                         Vsq=cholesky(Vold)'
>                         V=Vold
>                 }
>
>                 xpro=xold+lam*rnormal(1,nb,0,1)*Vsq  /* Draw, value calc. */
>
>
>                 pro=(*f)(xpro)
>
>                 if      (pro==. ) alpha=0       /* calc. of accept. prob */
>
>                 else if (pro>old) alpha=1
>                 else alpha=exp(pro-old)
>
>                 if (runiform(1,1)<alpha) {
>                         old=pro
>                         xold=xpro
>                         accept=1
>                 }
>
>                 lam=lam*exp(1/(i+1)^damper*(alpha-aopt)) /*update*/
>                 xs=xs\xold
>                 val=val\old
>                 Accept=Accept\accept
>                 mu=mu+1/(i+1)^damper*(xold-mu)
>                 Vold=V
>                 V=V+1/(i+1)^damper*((xold-mu)´(xold-mu)-V)
>                 _makesymmetric(V)
>         }
>
>         val  =val[burn+1::draws,]
>         arate=mean(Accept[burn+1::draws,])
>         return(xs[burn+1::draws,])
> }
: end
```

The function starts by setting up a variable (`nb`) to hold the dimension of the distribution, and `xold`, which functions as x_t in the algorithms discussed in table 3, is set to the user-supplied initial value. The initial value of λ (called `lam`) is set as discussed by Andrieu and Thoms (2008, 359).

Next the log value of the distribution (`f`) at `xold` is calculated and called `old`. The next few steps proceed as one would expect. However, I find it useful to have a default covariance matrix waiting—`Vold` in the code—in case the Cholesky decomposition encounters problems. For example, this could happen if the initial variance–covariance matrix is not positive definite or if there is insufficient variation in the draws, which sometimes happens in the early stages of a run. Once a usable covariance matrix has been obtained, `xpro` (which functions as Y_t in the algorithms in tables 1, 2, and 3) is formed using a conformable vector of standard normal random variates, and the function is evaluated at `xpro`.

The acceptance probability `alpha` is then calculated in a numerically stable way in an `if-else if-else` block. If the target function returns a missing value when evaluated, `alpha` is set to 0 so that the draw will not be retained. If the proposal produces a higher value of the target function, `alpha` is set to one. Otherwise, it is set as described by the algorithms.[19] Finally, a uniform random variable is drawn that determines whether the draw is to be accepted. Once this is known, all values are updated according to the scheme described in table 3. Once the `for` loop concludes, the algorithm overwrites the acceptance rate, `arate`, and the function value, `val`, and returns the results of the draw.

6 Conclusions

I have given a brief overview of adaptive MCMC methods and how they can be implemented using the Mata routine `amcmc()` and a suite of functions `amcmc_*()`. While I have given some ideas about how one might use and display obtained results, my primary purpose is to present and describe an implementation of adaptive MCMC algorithms.

19. The Mata function `exp()` does not evaluate to missing for very small values as it does for very large values.

I have not discussed how one should set up the parameters of the draw, such as the number of draws to take, whether to use a global sampler, or how aggressively to tune the proposal distribution. I have also not discussed what users should do once they have obtained draws from an adaptive MCMC algorithm. The functions leave these decisions in the hands of users. Creating, describing, and analyzing results obtained via MCMC is fortunately the subject of extensive literature. Broadly speaking, literature on MCMC is built around the related issues of assessing convergence of a run and of assessing the mixing and intensity of a run. A further issue is how one should deal with autocorrelation between draws. Whatever means are used to analyze results, it is fortunate that Stata provides a ready-made battery of tools to summarize, modify, and graph results. However, while it is often easy to spot problems in an MCMC run, it is impossible to know whether the run has actually provided draws from the intended distribution.

On the subject of convergence, there is not any universally accepted criterion, but researchers propose many guidelines. Gelman and Rubin (1992) present several useful ideas. A general discussion appears in Geyer (2011), and some practical advice appears in Gelman and Shirley (2011), who advocate discarding the first half of a run as a burn-in period and performing multiple runs in parallel from different starting points and comparing results. To be sure that one is actually sampling from the right region of the density, one can use "heated" distributions in preliminary runs. Effectively, these heated distributions raise the likelihood function to some fractional power,[20] which flattens the distribution and allows for more rapid and broader exploration of the parameter space.

One can also compare the results of multiple runs and compare the variance within runs and between runs. A useful technique is to investigate the autocorrelation function of results and then "thin" the results, retaining only a fraction of the draws so that most of the autocorrelation is rid from the data. One can use time-series tools to test for autocorrelation among draws. A possibility discussed by Gelman and Shirley (2011) is to jumble the results of the simulation. While it might seem obvious, it is worthwhile to note that solutions to these problems are interdependent. A draw that exhibits a lot of autocorrelation may require more thinning and a longer run to obtain a suitable number of draws. A good place to start with these and other aspects of analyzing results is Brooks et al. (2011).

As may have been clear from the examples presented in section 4, another option is to run the algorithm for some suitable amount of time and then restart the run without adaptation by using previous results as starting values so that one is drawing from an invariant proposal distribution. A simple yet useful starting point in judging convergence is seeing whether the algorithm produces results with graphs that look like those in figure 2 but not those in figure 4. A graph that does not contain jumps or flat spots and looks more or less like white noise is a preliminary indication that the algorithm is working well. However, pseudo-convergence can still be very difficult to detect. In addition to containing much practical advice, Geyer (2011) also advises that one should at least do an overnight run, adding only half in jest that "one should start

20. Or, equivalently, multiply the log likelihood by a fractional power.

a run when the article is submitted and keep running until the referee's reports arrive. This cannot delay the article, and may detect pseudo-convergence" (Geyer 2011, 18).

7 References

Andrieu, C., and J. Thoms. 2008. A tutorial on adaptive MCMC. *Statistics and Computing* 18: 343–373.

Brooks, S., A. Gelman, G. L. Jones, and X.-L. Meng, eds. 2011. *Handbook of Markov Chain Monte Carlo*. Boca Raton, FL: Chapman & Hall/CRC.

Chernozhukov, V., and H. Hong. 2003. An MCMC approach to classical estimation. *Journal of Econometrics* 115: 293–346.

Chib, S., and E. Greenberg. 1995. Understanding the Metropolis–Hastings algorithm. *American Statistician* 49: 327–335.

Gelman, A., and D. B. Rubin. 1992. Inference from iterative simulation using multiple sequences. *Statistical Science* 7: 457–472.

Gelman, A., and K. Shirley. 2011. Inference from simulations and monitoring convergence. In *Handbook of Markov Chain Monte Carlo*, ed. S. Brooks, A. Gelman, G. L. Jones, and X.-L. Meng, 163–174. Boca Raton, FL: Chapman & Hall/CRC.

Geyer, C. J. 2011. Introduction to Markov Chain Monte Carlo. In *Handbook of Markov Chain Monte Carlo*, ed. S. Brooks, A. Gelman, G. L. Jones, and X.-L. Meng, 3–48. Boca Raton, FL: Chapman & Hall/CRC.

Hole, A. R. 2007. Fitting mixed logit models by using maximum simulated likelihood. *Stata Journal* 7: 388–401.

Powell, J. L. 1984. Least absolute deviations estimation for the censored regression model. *Journal of Econometrics* 25: 303–325.

———. 1986. Censored regression quantiles. *Journal of Econometrics* 32: 143–155.

Rosenthal, J. S. 2011. Optimal proposal distributions and adaptive MCMC. In *Handbook of Markov Chain Monte Carlo*, ed. S. Brooks, A. Gelman, G. L. Jones, and X.-L. Meng, 93–112. Boca Raton, FL: Chapman & Hall/CRC.

Train, K. E. 2009. *Discrete Choice Methods with Simulation*. 2nd ed. Cambridge: Cambridge University Press.

About the author

Matthew Baker is an associate professor of economics at Hunter College and the Graduate Center, City University of New York. One of his current interests is simulation-based econometrics.