EMM: A Program for Efficient Method of Moments Estimation

Version 2.6

User's Guide [1]

A. Ronald Gallant
Duke University
Fuqua School of Business
Durham NC 27708-0120 USA

George Tauchen
Duke University
Department of Economics
Durham NC 27708-0097 USA

August 1993
Last Revised May 2010

[1]Research supported by the National Science Foundation. The code and this guide are available at http://econ.duke.edu/webfiles/arg/emm.

ABSTRACT

This Guide shows how to use the computer package EMM, which implements the estimator described in Gallant and Tauchen (1996*a*) "Which Moments to Match." The term EMM refers to Efficient Method of Moments. The Guide provides an overview of the estimator, instructions on how to install the software, and a description of the package. It also walks the reader through some examples.

This implementation of EMM adapts the MCMC estimator proposed by Chernozhukov and Hong (2003) "An MCMC Approach to Classical Estimation," to simulation estimators. The Chernozhukov-Hong approach is substantially superior to conventional derivative based hill climbing optimizers for this class of problems.

The EMM package is actually a general purpose implementation of the Chernozhukov-Hong estimator and is therefore not restricted to simulation estimators. An MLE option allows the code to be used for maximum likelihood or any statistical criterion function that can be given the pseudo likelihood interpretation described in Chernozhukov and Hong (2003). GMM is an example; it is illustrated by a VAR example included in the distribution. MLE is illustrated with a translog consumer demand system with demand shares distributed as the logistic normal taken from Gallant (1987). It shows off the Chernozhukov-Hong estimator to good advantage because a vexing problem with hill climbers is to keep model parameters in the region where predicted shares are positive for every observation, which is nearly impossible with conventional hill climbing algorithms but is easy using the the Chernozhukov-Hong estimator.

The EMM package can also be used for Bayesian estimation because the user can supply a prior distribution. One is not restricted to a maximum likelihood objective function in Bayesian estimation. GMM and EMM can be given a Bayesian interpretation. See Gallant and Hong (2007) for details. Three asset pricing examples illustrating Bayesian EMM are in the distribution: the habit persistence model of Campbell and Cochrane (1999), the long run risks model of Bansal and Yaron (2004), and the prospect theory model of Barberis, Huang, and Santos (2001) implemented on the data of Bansal, Gallant, and Tauchen (2007) using the prior of Aldrich and Gallant (2010).

The code and this guide are available at http://econ.duke.edu/webfiles/arg/emm.

Contents

1 Introduction **1**

1.1 Overview 1

1.2 The EMM Estimator 3

1.3 The Chernozhukov and Hong Method 4

1.4 GARCH-SNP 7

1.5 Using this Guide 8

2 Building and Running EMM **8**

2.1 Availability 8

2.2 Building and Running EMM 8

3 The Structure of the EMM Distribution **10**

3.1 User Supplied Class 10

3.2 The Input Parameter File 14

3.2.1 PARMFILE HISTORY 16

3.2.2 ESTIMATION DESCRIPTION 16

3.2.3 DATA DESCRIPTION 19

3.2.4 MODEL DESCRIPTION 20

3.2.5 MODEL PARMFILE 20

3.2.6 OBJFUN PARMFILE 20

3.2.7 PARAMETER START VALUES 21

3.2.8 PARAMETER INCREMENTS 22

3.2.9 PROPOSAL SCALING 23

3.2.10 PROPOSAL GROUPING 23

3.3 Directory Structure 23

3.3.1 snpsrc 24

3.3.2 emmsrc 24

3.3.3 svfx 24

3.3.4 self 24

3.3.5 elec 24

3.3.6 var 24
3.3.7 hab, lrr, pro 25
3.3.8 lib 25
3.3.9 snprun 25
3.3.10 emmrun 25

4 The Elementary Stochastic Volatility Model 27

5 Fitting the Score Generator 28

6 Worked Example 29

6.1 User-Supplied Classes and Files 29
6.2 Running the Program 34
6.3 Group Move Proposal 43
6.4 Putting Parameters on a Grid 44
6.5 The Effect of Temperature 50
6.6 Interpreting the Output 52
6.7 Score Diagnostics 58
6.8 More is Better? 58
6.9 Criterion Difference 59
6.10 Running on a Parallel Machine 62

7 Maximum Likelihood Estimation 68

8 References 76

1 Introduction

1.1 Overview

Gallant and Tauchen (1996a, 2002a) developed a systematic approach to generating moment conditions for the generalized method of moments (GMM) estimator (Hansen, 1982) of the parameters of a structural model. The approach, termed Efficient Method of Moments (EMM), is an alternative to the common practice of selecting a few low order moments on an *ad hoc* basis and then proceeding with GMM. The EMM methodology has proved useful and practical. An early version was used for estimating asset pricing models (Bansal, Gallant, Hussey, and Tauchen, 1993, 1995). Some recent applications include, among others, Gallant, Hong, and Khwaja (2010) for estimation of a dynamic game with a serially correlated, endogenous, unobserved state, Gallant and Hong (2007) for assessment of the plausibility of recursive utility, Chernov and Ghysels (2000), Chernov, Gallant, Ghysels, and Tauchen (2003), and Gallant and Tauchen (2001) for assessment of stochastic volatility models, Dai and Singleton (2000), Bansal and Zhou (2001), Ahn, Gallant, and Dittmar (2002), and Tauchen (1997) for interest rate modeling, Gallant and Long (1997) and Chung and Tauchen (2001) for exchange rate modeling.

The idea behind EMM is simple: Use the expectation under the structural model of the score from an auxiliary model as the vector of moment conditions. This score is the derivative of the log density of the auxiliary model with respect to the parameters of the auxiliary model. Thus, the moment conditions depend upon both the parameters of the auxiliary model and the parameters of the structural model. The dependence on the parameters of the auxiliary model is eliminated by replacing them by their maximum likelihood estimates, which are computed by maximizing the likelihood of the auxiliary model.

The auxiliary model is called the *score generator*. The score generator need not encompass (nest) the structural model. If it does, then the estimator is as efficient as the maximum likelihood estimator. Hence the approach ensures efficiency against a given parametric model. If the score generator closely approximates the actual distribution of the data, even though it does not encompass it, then the estimator is nearly fully efficient (Gallant and Tauchen, 1996a, 2001, 2002a; Tauchen 1997; Gallant and Long, 1997), which motivates the term

Efficient Method of Moments (EMM).

The estimation context is one where the structural model defines a data generation process. The key feature of this data generation process is that it is relatively easy to compute the expectation of a nonlinear function given values for the structural parameters. An expectation may be computed by simulation, by numerical quadrature, or by analytic expressions, whichever is the most convenient. In the case of simulation, one averages the nonlinear function over simulated realization for given values of the structural parameters. Denote the simulation by

$$\rho \mapsto \{\hat{y}_\tau(\rho), \hat{x}_{\tau-1}(\rho)\}_{\tau=1}^{N}$$

where ρ is the of vector structural parameters to be estimated, $\hat{y}_\tau$ are endogenous variables, and $\hat{x}_\tau$ are lagged endogenous variables. Lagged endogenous variables are generated through the internal structural model, hence the dependence on ρ. Usually we abbreviate to $\{\hat{y}_t, \hat{x}_t\}_{t=1}^{N}$.

Examples of this estimation context are the panel data models motivating the simulated method of moments approach of Pakes and Pollard (1989) and McFadden (1989). Another is the asset pricing model that motivates the dynamic method of moments estimator of Duffie and Singleton (1993). In these examples, the likelihood is difficult to compute, so maximum likelihood is infeasible. Simulation and moment matching thus naturally arise. The EMM estimator has some computational advantages relative to Indirect Inference (Smith, 1993; Gourieroux, Monfort, and Renault, 1993). EMM does not require computation of the binding function and it does not require estimation of the Hessian matrix of the auxiliary model. Tauchen (1997) provides a general overview of EMM and references to current applications, while Tauchen (1998) analyzes the behavior of the EMM objective function. A recent review article is Gallant and Tauchen (2009).

There is one important case that the EMM package does not cover. That is the case where one wishes to do classical Bayesian inference but no expression for the likelihood is available although the model can be simulated. The EMM package would require that one substitute a GMM or EMM criterion function for the likelihood. Gallant and McCulloch (2009) propose a method for handling this case. They also propose methods for both absolute and relative Bayesian model assessment. Code implementing the Gallant and McCulloch method and a

User's Guide are available at http://econ.duke.edu/webfiles/arg/gsm.

1.2 The EMM Estimator

This guide shows how to use a C++ package that implements the EMM estimator for the case in which the structural model defines a strictly stationary Markovian process and there are no covariates. The structural model is that of CASE 2 of Gallant and Tauchen (1996a). The setup subsumes a wide variety of situations in macroeconomics and finance. The SNP model is the score generator. A user should be able to modify the code to accommodate other score generators and to accommodate covariates, as in CASE 1 or CASE 3 of Gallant and Tauchen (1996a).

Let $\{\tilde{y}_t, \tilde{x}_{t-1}\}_{t=1}^n$ denote the observed data set, where $\tilde{x}_{t-1} = (\tilde{y}_{t-1}, \ldots, \tilde{y}_{t-L})$, $L \geq 1$. The first step is maximum likelihood estimation of the score generator

$$\tilde{\theta}_n = \operatorname*{argmax}_{\theta \in \Theta} \frac{1}{n} \sum_{t=1}^{n} \ln f_t(\tilde{y}_t \,|\, \tilde{x}_{t-1}, \theta)$$

For the second step, the moment criterion is

$$m_n(\rho, \tilde{\theta}_n) = \frac{1}{N} \sum_{\tau=1}^{N} (\partial/\partial\theta) \ln f[\hat{y}_\tau(\rho) \,|\, \hat{x}_{\tau-1}(\rho), \tilde{\theta}_n],$$

and the GMM estimator of the structural parameter vector is

$$\hat{\rho}_n = \operatorname*{argmin}_{\rho \in \mathcal{R}} m_n'(\rho, \tilde{\theta}_n)(\tilde{\mathcal{I}}_n)^{-1} m_n(\rho, \tilde{\theta}_n),$$

where $(\tilde{\mathcal{I}}_n)^{-1}$ is the weighting matrix.

The computations necessary to form $\tilde{\mathcal{I}}_n$ depend upon how well one thinks that the score generator approximates the true data generating process. If one is confident that the score generator is a good statistical approximation to the data generating process, then the estimator

$$\tilde{\mathcal{I}}_n = \frac{1}{n} \sum_{t=1}^{n} \Big[(\partial/\partial\theta) \ln f_t(\tilde{y}_t|\tilde{x}_{t-1}, \tilde{\theta}_n)\Big] \Big[(\partial/\partial\theta) \ln f_t(\tilde{y}_t|\tilde{x}_{t-1}, \tilde{\theta}_n)\Big]'.$$

can be used. This estimator only entails an outer-product-of-the-gradient computation and no weighted covariance matrix estimation. Conditions under which this estimator is valid are given in Gallant and Tauchen (1996a). To the extent the score generator provides a less good approximation a weighted covariance matrix estimator, i.e, a "HAC" estimator as reviewed in Andrews (1991), should be used for $\tilde{\mathcal{I}}_n$.

Regardless of what is used for $\tilde{\mathcal{I}}_n$, the sandwich standard errors computed by this adaptation of Chernozhukov and Hong (2003) are asymptotically correct. However, accurate computation of the sandwich standard errors is difficult and certain tuning parameters need to be chosen carefully to get reasonable results as discussed later in this guide. Using an accurate score generator so that sandwich standard errors do not have to be used is a better approach when feasible.

1.3 The Chernozhukov and Hong Method

The computational methods discussed here and implemented by the EMM package apply to any discrepancy function $s_n(\rho)$ that produces asymptotically normal estimates; i.e., any discrepancy function for which there exist ρ^o, $\mathcal{I}$ and $\mathcal{J}$ such that

$$\mathcal{J}\sqrt{n}(\hat{\rho}_n - \rho^o) = \sqrt{n}\frac{\partial}{\partial\rho}s_n(\rho) + o_p(1) \text{ and } \sqrt{n}\frac{\partial}{\partial\rho}s_n(\rho) \xrightarrow{\mathcal{L}} N(0,\mathcal{I}) \tag{1}$$

The $\mathcal{I}$ matrix discussed in this subsection pertains to $\hat{\rho}_n$ and is not the $\tilde{\mathcal{I}}_n$ weighting function of the immediately preceding subsection.

Quasi maximum likelihood estimation requires the computation of the estimator itself, $\hat{\rho}_n = \operatorname*{argmin}_{\rho} s_n(\rho)$, an estimate of the Hessian

$$\mathcal{J} = \frac{\partial}{\partial\rho\partial\rho'}\, s^o(\rho^o),$$

where $s^o(\rho) = \lim_{n\to\infty} s_n(\rho)$, and an estimate of Fisher's information

$$\mathcal{I} = \operatorname{Var}\left[\frac{\partial}{\partial\rho'}\sqrt{n}\, s_n(\rho^o)\right] = \mathcal{E}\left[\frac{\partial}{\partial\rho'}\sqrt{n}\, s_n(\rho^o)\right]\left[\frac{\partial}{\partial\rho'}\sqrt{n}\, s_n(\rho^o)\right]'.$$

The variance of $\sqrt{n}(\hat{\rho}_n - \rho^o)$ is then of the sandwich form

$$V_n = \operatorname{Var}\left[\sqrt{n}(\hat{\rho}_n - \rho^o)\right] = \mathcal{J}^{-1}\mathcal{I}\mathcal{J}^{-1}$$

Put $\ell(\rho) = e^{-n\, s_n(\rho)}$. Apply Bayesian MCMC methods with $\ell(\rho)$ as the likelihood. From the resulting MCMC chain $\{\rho_i\}_{i=1}^R$ put

$$\hat{\rho}_n = \bar{\rho}_R = \frac{1}{R}\sum_{t=1}^{R}\rho_i \text{ and } \hat{\mathcal{J}}^{-1} = \left(\frac{n}{R}\right)\sum_{t=1}^{R}(\rho_i - \bar{\rho}_R)(\rho_i - \bar{\rho}_R)'$$

Alternatively, and definitely for EMM, use the mode of $\ell(\rho)$ as the estimator $\hat{\rho}_n$. The EMM package computes and reports both the mean and the mode.

Actually, the mode is the better choice of an estimator in most applications because the parameter values in the mode actually have generated a simulation. The parameter values in the mean vector may not even satisfy the support conditions of the structural model.

The strategy used to estimate $\mathcal{I}$ is the following. For ρ set to the mode, simulate the model, and generate I approximately independent bootstrap data sets $\{\hat{y}_{t,i}\}_{t=1}^{n}$, $i = 1, \ldots, I$, each of exactly the same sample size n as the original data. Keeping the size to exactly n and using model simulations makes the estimator below a heteroskedastic autocovariance consistent (HAC) estimator. Keeping the size to exactly n does not imply that the simulation size N should be set to n when using the program. The simulation size N should be set much larger than n in most instances. One way to get a bootstrap sample is to split this long simulation into blocks of size n. With this approach, the estimate of $\mathcal{I}$ would be a parametric bootstrap estimate. Alternatively, a Politis and Romano (1994) stationary bootstrap or some other method could be used to construct the blocks. The bootstrap generating mechanism is coded by the user. The worked examples in Section 6 provide examples of two different approaches. The code automatically prepends the data as the first bootstrap sample to whatever the user supplies.

Let $\hat{s}_{n,i}(\rho)$ denote the criterion function corresponding to the ith bootstrap data set $\{\hat{y}_{t,i}\}_{t=1}^{n}$ and let $\hat{\rho}_n$ denote mode of $\ell(\rho)$. Compute $\frac{\partial}{\partial \rho'} \sqrt{n}\, \hat{s}_{n,i}(\hat{\rho}_n)$ numerically. An estimate of the information matrix is the average

$$\hat{\mathcal{I}} = \frac{1}{I} \sum_{i=1}^{I} \left[\frac{\partial}{\partial \rho'} \sqrt{n}\, \hat{s}_{n,i}(\hat{\rho}_n) \right] \left[\frac{\partial}{\partial \rho'} \sqrt{n}\, \hat{s}_{n,i}(\hat{\rho}_n) \right]' \tag{2}$$

Note that for the EMM estimator one must compute the likelihood of the auxiliary model from the ith bootstrap sample and optimize it in order to get the ith EMM objective function $\hat{s}_{n,i}(\rho)$. This is done using the BFGS method. This is the step that makes computing an accurate numerical derivative accurately both difficult and costly for EMM. The code attempts to detect failure of the optimizer and failure to compute an accurate derivative and discard those instances. An objective function such as mle or GMM that does not rely on a preliminary optimization is not as much of a challenge to differentiate numerically. With these one can have more confidence that the code provides the correct answer.

If the SNP model is a good approximation to the true data generating process, the computation of $\hat{\mathcal{I}}$ is not necessary because $\mathcal{I} = \mathcal{J}$. This issue is discussed in detail in Gallant and Tauchen (1996a). The same is true for maximum likelihood if the model is correctly specified. The code provides a switch, `kilse`, to turn off the computation of $\mathcal{I}$ when it is unnecessary.

The code provides the option of putting the parameter ρ on a grid. This increases speed by allowing $s_n(\rho)$ and related variables to be obtained by table lookup thus avoiding recomputation for a value of ρ that has already been visited in the MCMC chain. This is a useful feature when the objective function $s_n(\rho)$ is costly to compute.

Note that $S_n(\rho) = \tau s_n(\rho)$ is also a valid criterion according to the theory. This gives one a temperature parameter τ to use for tuning the chain. This feature is implemented in the package.

A random walk, single move, normal proposal is the workhorse of the EMM package. When parameters are put on a grid, a discrete proposal density is used instead that has probabilities assigned to grid points proportionally to this normal. Group moves are also supported. It is easy to substitute an alternative proposal density and a conditional normal that implements an automated Metropolis within Gibbs strategy is included in the package as a proof of concept. It is best regarded as a proof of concept because it doesn't seem to work all that well.

Simulated method of moments is exactly the same as the foregoing but with a GMM criterion replacing $s_n(\rho)$. An VAR example is included in the distribution. As with EMM, if the correct weight function is used with the GMM criterion function, then $\mathcal{I} = \mathcal{J}$ so that $\mathcal{I}$ need not be computed and there is no need for any numerical differentiation. But often the effectiveness of the GMM weighting function is doubtful and it can cloud the interpretation of results. One may prefer sandwich standard errors regardless. With GMM there is usually no numerical optimization to compute moments as with EMM so better accuracy can be expected.

The MCMC method described here makes the imposition of support restrictions, inequality restrictions, and informative prior information exceptionally convenient. These restrictions and prior information can be imposed on model parameters or on (nonlinear) functionals of the model that can only be known via simulation. This feature is implemented in the EMM package. Three asset pricing models are included in the distribution that use these features: the habit persistence model of Campbell and Cochrane (1999), the long run risks model of Bansal and Yaron (2004), and the prospect theory model of Barberis, Huang, and Santos (2001) implemented on the data of Bansal, Gallant, and Tauchen (2007) using the prior of Aldrich and Gallant (2010). A dynamic game application is available at http://www.econ.duke.edu/∼arg/compecon.

For some structural models it is difficult to check the validity of parameters without first substantially altering the internal state of the model. As it is wasteful to do this twice, parameters are set before member support of the user's implementation of the structural model is called. The user should be aware of this fact when writing code for the structural model because one is not guaranteed that ρ will be valid. Member support is called immediately after ρ is set. If it returns false, then no other member of the user's implementation of the structural model is called.

As mentioned earlier, these ideas are not restricted to simulation estimators. The EMM package is actually a general purpose implementation of the Chernozhukov-Hong estimator. An illustration of how the code may be used to implement maximum likelihood is included with the package and described in the *Guide*. The application used for this illustration is a translog consumer demand system for electricity by time of day with demand shares distributed as the logistic normal that is taken from Gallant (1987). It shows off the Chernozhukov-Hong estimator to good advantage because a vexing problem with hill climbers is trying to keep model parameters in the region where predicted shares are positive for every observed price/expenditure vector. This is nearly impossible to achieve when using conventional derivative based hill climbing algorithms but is trivially easy to achieve using the the Chernozhukov-Hong estimator as implemented in the EMM package.

1.4 GARCH-SNP

The most recent version of SNP is 9.0, which permits a GARCH specification for the conditional variance of the leading term of the score generator that has the BEKK multivariate form with modifications to directly incorporate variance structures exhibiting level and leverage effects. See the *SNP User's Guide* for more details.

1.5 Using this Guide

New users should work through the simple stochastic volatility model developed in Sections 4 through 6. Use Section 3 for reference purposes.

2 Building and Running EMM

2.1 Availability

C++ code and this *Guide* as a PostScript or PDF file are available at `http://econ.duke.edu/webfiles/arg/emm`.

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.

Since this implementation of EMM uses SNP model as the score generator, SNP is included in the distribution.

2.2 Building and Running EMM

Download `emm.tar` from `http://econ.duke.edu/webfiles/arg/emm`. On a Unix machine use `tar -xf emm.tar` to expand the tar archive into a directory that will be named `emm`. On a Windows machine use unzip; i.e., Windows recognizes a Unix tar archive as a zip file. The distribution has the following directory structure:

emm

elec
emmman
emmrun
emmsrc
hab
lib
  libscl
  libsmm
  libsnp
lrr
pro
self
snpman
snprun
snpsrc
svfx
utility
var

Often one changes the name `emm` of the parent directory to a name that represents the project one is working on.

First the three libraries libscl, libsnp, and libsmm must be built, in that order. On a Unix machine change directory to `lib/libscl/gpp` and type `make`. For Microsoft Windows, a batch file supplied by Microsoft with their compiler must be executed first. The following is an example:

```
C:"\Program Files\Microsoft Visual Studio .NET\Vc7\bin\"vcvars32.bat
```

The exact syntax will depend on where the Microsoft C++ compiler is installed. Change directory to `lib\libscl\ms` and type `nmake`. Building libsnp and libsmm is similar.

To run the SNP example that comes with the distribution on a Unix machine within the directory `snprun` copy `makefile.gpp` to `makefile`, type `make` and then `./snp`. Similarly for EMM, within `emmrun` copy `makefile.gpp` to `makefile`, type `make` and then `./emm sv.ctrl.000`. For Microsoft copy `makefile.ms` to `makefile`, type `C:"\...\" vcvars32.bat,` `nmake`, then `snp`, etc.

At http://econ.duke.edu/webfiles/arg/djgpp is a free C++ compiler that runs under Windows. Using the djgpp compiler is similar to using the Microsoft compiler. There is a batch file gppsetup.bat included with the distribution that must be executed first. Then all else is for Microsoft except one uses the subdirectory gpp rather than ms for compilation of libraries, copies (overwrites) makefile.djgpp to makefile, and uses make instead of nmake.

3 The Structure of the EMM Distribution

The structure in the discussion of the example distributed with the distribution is presumed to be as above. The user is free to set up another file structure provided that the references in the makefiles are changed to correspond. Indeed, we often redo the file structure to suit a particular application. Below we will use the Unix file naming conventions. For Windows substitute a back slash directory separator for a forward slash.

3.1 User Supplied Class

As described in the worked examples farther on, the user supplies a class, which here we shall call sv_usrmod. The declaration for the class is in file emmusr.h, the code implementing it is in file emmusr.cpp. The functionality that sv_usrmod must provide is dictated by inheritance from class usrmod_base declared in libsmm/src/libsmm_base.h. Here is the relevant portion of libsmm/libsmm_base.h

```
#include "libscl.h"

namespace libsmm {

  /* Now in libscl

  struct den_val {
    bool    positive;
    REAL    log_den;
            den_val() : positive(false), log_den(-REAL_MAX) { }
            den_val(bool p, REAL l) : positive(p), log_den(l) { }
  };

  */

  class usrmod_base {
  public:
    virtual INTEGER len_rho() = 0;
    virtual INTEGER len_stats() = 0;
```

```
    virtual bool gen_sim(scl::realmat& sim, scl::realmat& stats) = 0;
      //Same seed every call
    virtual void get_rho(scl::realmat& rho) = 0;
    virtual void set_rho(const scl::realmat& rho) = 0;
    virtual bool support(const scl::realmat& rho) = 0;
    virtual den_val prior(const scl::realmat& rho,
      const scl::realmat& stats) = 0;
    virtual void write_usrvar(const char* filename) { return; }
    virtual ~usrmod_base() {}
    virtual bool gen_bootstrap(std::vector<scl::realmat>& bs)
      //New seed each call
      {return false;}
    virtual void set_data(const scl::realmat& dat) {}
    virtual libsmm::den_val likelihood(scl::realmat& predicted,
      scl::realmat& residuals)
      {return den_val(false,-REAL_MAX);}
  };

}
```

and here is the corresponding `emmusr.h`

```
#ifndef __FILE_EMMUSR_H_SEEN__
#define __FILE_EMMUSR_H_SEEN__
#include "libsnp.h"
#include "libsmm.h"
#include "emm_base.h"
#include "snp.h"

namespace emm {

  class sv_usrmod;

  typedef sv_usrmod usrmod_type;

  const INT_32BIT fixed_seed = 770116;

  class sv_usrmod : public libsmm::usrmod_base {
  private:
    scl::realmat data;
    scl::realmat rho;
    INTEGER slen;
    INTEGER spin;
    INTEGER lrho;
    INTEGER lstats;
    INT_32BIT variable_seed;
    bool sv_simulate (INT_32BIT& seed, INTEGER len,
      scl::realmat& sim, scl::realmat& stats, scl::realmat& latent);
  public:
    sv_usrmod (const scl::realmat& dat, INTEGER len_mod_parm,
      INTEGER len_mod_func, const std::vector<std::string>& mod_pfvec,
      const std::vector<std::string>& mod_alvec, std::ostream& detail);
    INTEGER len_rho() {return lrho;}
    INTEGER len_stats() {return lstats;}
    bool gen_sim(scl::realmat& sim, scl::realmat& stats)
      //Same seed every call
      {
        scl::realmat latent;
        INT_32BIT seed = fixed_seed;
        return sv_simulate(seed,slen,sim,stats,latent);
```

```
      }
    bool gen_bootstrap(std::vector<scl::realmat>& bs);
      //Different seed each call
    void get_rho(scl::realmat& parm) { parm = rho; }
    void set_rho(const scl::realmat& parm) { rho = parm; }
    bool support(const scl::realmat& rho);
    scl::den_val prior(const scl::realmat& rho, const scl::realmat& stats);
    void write_usrvar(const char* filename)
      {
        scl::realmat sim, stats, latent;
        INT_32BIT seed = fixed_seed;
        if ( sv_simulate(seed,slen,sim,stats,latent) ) {
          scl::vecwrite(filename, scl::rbind(sim,latent));
        }
      }
  };

}
#endif
```

Class `sv_usrmod` gets bound to program `emm` via the statement

```
typedef sv_usrmod usrmod_type;
```

as shown.

The types `REAL`, `INTEGER`, and `INT_32BIT` are defined by `typedef`'s in `scltypes.h` which gets included with `libscl.h`. On most machines these are `double`, `int`, and `int`, respectively. Class `realmat` is presented in `realmat.h` which gets included with `libscl.h`. This is a fairly complete matrix class that supports most linear algebra related to statistical applications including equation solving, inversion, and singular value decomposition. In general there is much in `libscl` that will aid the user in writing `sv_usrmod`, including a nonlinear equation solver and a nonlinear optimizer.

The idea behind `stats` is that there is more information about a simulation that the user needs to know besides the value of `rho` that generated it. A moment of a latent variable is an obvious example. The realmat `stats` of length `lstats` gets written to a file by program `emm` as does `rho` and much else as described later.

Also, `stats` may need to be computed to know if the simulation is useable. For instance, if usrmod simulates an asset pricing model, the real rate may be a latent variable and one may want to hold its average over the simulation fixed at about 1%. If `rho` generates a real rate that is not near 1%, then that `rho` should be rejected by the Metropolis-Hastings algorithm in `mcmc`. To determine usability, `stats` gets passed by `mcmc` to member `prior` which makes the decision and returns the verdict as a `den_val`. If `den_val.positive` is false, then the rho that generated the simulation is rejected by the Metropolis-Hastings algorithm. If den_val.positive is true and den_val.log_den==0, then the Metropolis-Hastings considers rho in the usual way as determined by the proposal and objective function. If den_val.positive is true and den_val.log_den!=0, then the value returned is added to the log objective function before the Metropolis-Hastings accept/reject decision is made; i.e. acts as an informative prior.

Stated differently, prior can be used exactly as in Bayesian inference. If a model is estimated by maximum likelihood, as discussed in Section 7, this is just classical Bayesian inference. However, it is possible to give EMM a Bayesian interpretation; see Gallant and Hong (2007).

Member support plays a similar role: it returns false if if rho is to be rejected. The difference between support and prior is that support is called before the simulation and prior after. The intent is to save the cost of an unnecessary simulation if rho violates support conditions that can be cheaply determined. Be aware that set_rho is always called before support.

In this example, we simulate the model in three member functions (methods): gen_sim, gen_bootstrap, and write_usrvar. We shall write one simulator, the private member function sv_simulate, and call it for these three methods. Note that stats is actually only needed by gen_sim. In most situations the cost of computing stats is trivial and so class usrmod can be structured so that stats is computed for every simulation. With this structure, the methods functions gen_sim and write_usrvar become trivial and can be coded in the header emmusr.h as shown. This leaves the constructor and members sv_simulate, gen_bootstrap, and usrvar as the items remaining to discuss and code in emmusr.cpp.

The constructor gets passed the data, lengths of the parameter and stat vectors, two std::vector of std::string named mod_pfvec and mod_alvec that the user controls through the parmfile as described immediately below and a std::ostream named detail to which to write if desired. For most applications this constructor argument list is sufficiently general and no modification to the constructor call in emm.cpp will be required.

The role of `sv_simulate` is to compute a simulated data set `sim` of dimension `lrho` by `len`, a vector of additional variables `latent` of the user's choice that has `slen` columns and however many rows the user's choice dictates, and the `stats` described above. (If one changes how the files are written from that coded in method `write_usrvar`, then `latent` can have any desired number of rows and columns.) Member `sv_simulate` is called in turn by members `gen_sim`, by member `gen_bootstrap`, and by member `write_usrvar` as shown. If `stats` is costly to compute because, e.g., a nonlinear optimization is required, then one would arrange the structure differently so `stats` is only computed when `gen_sim(sim,stats)` is called.

Member `write_usrvar` gets called with ρ set to the mode. It will get called each time the mode is recomputed, which is `nfile`+1 times. As written, `write_usrvar`, writes the simulation and the latent variable, which is log volatility in this instance, joined together as a single matrix to a file. However, `write_usrvar` can be changed to write anything internal to class `usrmod` to a file. An example where one might write something different is when a model is simulated at a monthly frequency but `sim` is annual data computed from the monthly simulation. One may want to write the monthly data to a file. Also, one can use the input variable `filename` as a stem, add suffixes, and write multiple files.

The job of `bootstrap` is to break a simulation from `sv_simulate` into blocks stored as a `std::vector` of `realmat`, each block of which is exactly the size of `data`.

3.2 The Input Parameter File

The EMM input parameter file contains several blocks of control information. An example, from Section 6, is

```
PARMFILE HISTORY (optional)
#
# This parmfile was written by EMM Version 2.5 using the following line from
# control.dat, which was read as char*, char*
# --------------------------------------------------------------------------
#     sv.parm.004                 sv
# --------------------------------------------------------------------------
#
ESTIMATION DESCRIPTION (required)
    SpotRate   Project name, pname, char*
         2.6   EMM version, defines format of this file, emmver, float
           0   Objfun type, 0 EMM, 1 MLE, 2 usr, objtype, int
           0   Proposal type, 0 group_move, 1 cond_move, 2 usr, proptype, int
           1   Write detailed output if print=1, int
```

```
         457   Seed for MCMC simulations, iseed, int
        5000   Number of MCMC simulations per file, lchain, int
           5   Number of MCMC simulation files beyond the first, nfile, int
         1.0   Rescale proposal scaling by this value, sclfac, float
         1.0   Rescale parameter increments by this value, incfac, float
         1.0   Rescale objfun by this value, temperature, float
           1   Sandwich variance not computed if kilse=1, int
           1   The stride used to write MCMC simulations, stride, int
           0   Draw from prior if draw_from_prior=1, int
           0   Max cache size, max_cache_size, int
DATA DESCRIPTION (required) (mod and obj constructors see realmat data(M,n))
           1   Dimension of the data, M, int
         834   Number of observations, n, int
dmark.dat      File name, any length, no embedded blanks, dsn, string
4              Read these white space separated fields, fields, intvec
MODEL DESCRIPTION (required)
           6   Number of modal parameters, len_mod_parm, int
           8   Number of model functionals, len_mod_func, int
MODEL PARMFILE (required) (constructor sees as vector<string> pfvec, alvec)
__none__       File name, code __none__ if none, mod_parmfile, string
#begin additional lines
        5000   Number of observations in simulated data, slen (=N), int
         500   Initial simulations to eliminate transients, spin, int
#end additional lines
OBJFUN PARMFILE (required) (constructor sees as vector<string> pfvec, alvec)
11114000.fit   File name, code __none__ if none, obj_parmfile, string
#begin additional lines
#end additional lines
PARAMETER START VALUES (required)
   8.14895773998386974e-02    1
   2.60176173224806462e-02    1
   7.28064108382113773e-02    1
   9.34458288546352378e-01    1
   1.82976626342656323e-01    1
   1.54595167239619968e-01    1
PROPOSAL SCALING (required)
   1.56250000000000000e-02
   1.56250000000000000e-02
   1.56250000000000000e-02
   1.56250000000000000e-02
   1.56250000000000000e-02
   3.12500000000000000e-02
PARAMETER INCREMENTS (optional) (fractional powers of two recommended)
   1.56250000000000000e-02
   1.56250000000000000e-02
   3.12500000000000000e-02
   3.12500000000000000e-02
   3.12500000000000000e-02
   3.12500000000000000e-02
PROPOSAL GROUPING (optional) (frequencies are relative)
 0.1    1
   1  1.0
 0.1    2
   2  1.0
 0.2     3     4     5
   3   1.0   0.7  -0.6
   4   0.7   1.0  -0.8
   5  -0.6  -0.8   1.0
 0.1    6
   6  1.0
```

A description of each block of the input file follows.

3.2.1 PARMFILE HISTORY

This block is optional. It is written by `emm` to the output parmfile `parmfile.fit` at the end of every run. It consists of seven lines that begin with # that should be left alone. After these seven lines, the user can add additional lines that begin with a # and these will get copied from the input parmfile to the output parmfile.

3.2.2 ESTIMATION DESCRIPTION

Under the block labeled `ESTIMATION DESCRIPTION`, there are parameters that govern the computations:

pname: Project name. Chosen by the user for identification purposes.

emmver: Version of EMM. Versions 2.1 through 2.4 parmfiles will work with Version 2.6.

objtype: The objective function to be used in the estimation. The code is set up so the user can code an alternative and select it by setting objtype=2. One would code it in `emmusr.h` and `emmusr.cpp`. At the beginning of `emmusr.h` one would need to insert the compiler directive

```
#define USR_OBJFUN_TYPE_IMPLEMENTED
```

and at the beginning of `namespace emm` add a binding such as

```
class usr_objfun;
typedef usr_objfun objfun_type;
```

Examples are `emmusr.h` and `emmusr.cpp` in directory `var` of the distribution.

proptype: Standard is the group move proposal which defaults to a single move proposal when the optional block `PROPOSAL GROUPING` is missing from the parmfile. How to specify group moves in the parmfile is discussed in Subsection 6.3. When the `PROPOSAL GROUPING` block is missing, the `proptype`=0 proposal randomly selects an element of ρ to move and the draws from a normal; i.e. a move-one-at-a-time random walk. When `PROPOSAL GROUPING` block is present the proposal randomly selects one of the groups defined therein to move and draws from a user specified multivariate normal. Setting proptype=1 selects a proposal that attempts to automate group moves with indifferent success. It serves as an example to show how a alternative proposal is coded. A user coded proposal would be selected by setting proptype=2. One would code it in emmusr.h and emmusr.cpp. At the beginning of emmusr.h one would need to insert the compiler directive

```
#define USR_PROPOSAL_TYPE_IMPLEMENTED
```

and at the beginning of namespace emm add a binding such as

```
class usr_proposal;
typedef usr_proposal proposal_type;
```

Examples are in proposal.cpp in libsmm/src.

print: If print=1, then voluminous debugging information is written to file detail.dat. Setting print=0 suppresses printing. To completely suppress printing, the control variable print in the SNP parmfile should be set to 0 also.

seed: Seed for the MCMC chain.

lchain: The MCMC chain is broken up into pieces and written to files rho.000.dat, rho.001.dat, etc. The variable lchain determines the number of draws per file.

nfile: Determines how many files in addition to rho.000.dat are generated. The total length of the MCMC chain is $R = \texttt{lchain}*(\texttt{nfile}+1)$. Many other files are produced to describe the chain such as reject.000.dat, pi.000.dat, stats.000.dat as well as summary files, files containing variance matrices, etc.

incfac: To increase speed, the chain is cached. The cache is emmcache.dat. After every run, a new cache emmcache.new is produced which the user should copy to emmcache.dat if the run was successful. To generate the cache, ρ is put on grid as determined by the PARAMETER INCREMENTS block described below. The variable incfac allows one to make this grid coarser or finer without changing the relative increments; it should be a power of two, e.g. 8 or 0.125.

sclfac: Rescales the proposal standard deviations that are set in the PROPOSAL SCALING block without changing relative values.

temperature: This variable controls the peakedness of the objective function. Putting temperature = 2 is like doubling the number of observations from which the SNP score was computed, which makes the objective function more peaked. Putting temperature = 0.5 would be like halving them. For Bayesian inference it is essential that temperature = 1.

kilse : Computing sandwich standard errors is costly and often unnecessary, as discussed in Subsection 1.3. Setting kilse = 1 will stop them from being computed. When kilse = 1, `bootstrap` is not called and does not need to be coded. Even for an estimator that does require the computation of sandwich standard errors, one should set `kilse` = 1 during the early hill climbing phase of the chain . When the objective function has reached its plateau and the stationary portion of the chain has been reached, `kilse` can be set to 0. This point is determined graphically as discussed in Section 6.

stride : EMM writes the MCMC chains to files of length `lchain` as explained above. If `stride=1`, every element of the chain is written. If `stride=2`, every other element is written and the length of an output files becomes `lchain/2`. Similarly for higher values of `stride`. Stride greater than one reduces memory requirements because values not written are not stored anywhere. One consequence of this is that statistics such as the Hessian are computed only from the elements of the MCMC chain that are written, not from all that are generated. The exceptions are that the mode and the rejection count are computed from all elements that were generated.

draw_from_prior : When EMM is used for Bayesian estimation and the prior is proper, it is useful to be able to draw from the prior for at least two purposes. The first is to be able to compare the prior and posterior distribution of estimates of parameters and functionals. The other is as an intermediate step in computing posterior probabilities for model selection as discussed in, e.g., Gamerman and Lopes (2006, Section 7.2.1). The essential information for model selection is in the output files named `pi.000.dat`, `pi.001.dat`, etc. (to which a user defined prefix is prepended). Their structure is discussed in more detail later but, breifly, the information one needs are the likelihood draws, in the second row, and the prior draws in the third row. When `draw_from_prior=0` these will be draws made by comparing the posterior at the accept/reject step of the MCMC chain, as will be true of all other output files such as `rho.000.dat`, `rho.001.dat`, etc. When `draw_from_prior=1` these will be draws made by comparing the prior at the accept/reject step of the MCMC chain, as will be true of all other output files. Setting `draw_from_prior=1` when the prior is not proper is a ghastly error.

max_cache_size: EMM caches past values so that rather than compute the objective function and stats afresh they can be gotten by table lookup. If either of these are costly to compute then using a cache can reduce run times significantly provided that the parameters are put on a grid using the `PARAMETER INCREMENTS` block described below. At the conclusion of a run a file `emmcache.new` is written. This can be renamed `emmcache.dat` and it will be read in and used in the next run. The variable `max_cache_size` limits the internal size of this cache and hence the size of `emmcache.new`. The number of lines in `emmcache.new` is `max_cache_size` times the sum of the number of parameters and the number of stats, plus one. The cache hit rate is printed to file `detail.dat` to help guide the choice of `max_cache_size`. If the parameters are not put on a grid, the hit rate will be so low as to make using a cache pointless. Therefore, when the the `PARAMETER INCREMENTS` block is missing, set `max_cache_size=0`.

3.2.3 DATA DESCRIPTION

In the block labeled `DATA DESCRIPTION` are parameters that specify the dimension of the data, the number of observations, and govern reading of the data. The data are presumed to be stored in a file containing rows that have values separated by blanks containing the data for each observation y_t and perhaps additional values such as dates or the index t. There should be one line for each $t = 1, \ldots, n$. The presence of the line terminating character is important because the C++ function `getline` does the reading.

M: The dimension of the vector y_t.

n: The number of observations to be read. The value can be smaller than the number of observations in the file in which case those at the end will not be read.

dsn: The name of the file from which the data are to be read.

fields : Lastly, one has `fields`. One must use care here because errors can cause the program to crash with misleading diagnostic messages, if any at all. As just mentioned, the presumptions is that the data are arranged in a table with time t as the row index and the elements of y_t in the columns. The blank separated numbers here specify the fields (columns) of the data in the order in which they are to be assigned to the elements $y_{1t}, y_{2t}, \ldots, y_{Mt}$ of y_t. It does not hurt to have too many fields listed because only the first M are read. The disaster is when there are too few (less than M) or one of them is larger than the actual number of columns in the data set. A few of the first and last values of y_t read in are printed in the file `detail.dat` which should be checked to make sure the data were read correctly. Fields can be specified as a single digit or as a range. Thus, one can enter either "1 2 3 5" or

“1:3 5”. (At time of writing, neither SNP nor GSM permit fields to be entered as a range.)

3.2.4 MODEL DESCRIPTION

The MODEL DESCRIPTION block is straightforward, it gives the dimensions of the parameters of the model.

len_mod_parm : The dimension of `rho`, which is the parameter vector of the model.

len_mod_func: The dimension of `stats`, which is the vector of statistics (functionals) of the model that are computed from a simulation of the model.

3.2.5 MODEL PARMFILE

The vectors `mod_pfvec` and `mod_alvec` of type `vector<string>` that are passed to the `usrmod` constructor are defined in the MODEL PARMFILE block. Note that the size of the simulation computed by `sv_simulate` is determined by the user supplied `usrmod` constructor. If the user wants the simulation size to be a value specified in the parmfile, then that value goes in this block as in our sample parmfile.

mod_parmfile: This is the name of a file containing lines of the user's choosing. This file is read and passed to the `usrmod` constructor as the `std::vector` of `std::string` `mod_pfvec`. If there is no `mod_parmfile` then code `__none__` as the filename. In the distributed code, the example `self` reads the SNP parmfile so that `11114000.fit` is entered here instead of `__none__` for that example.

#begin additional lines, #end additional lines: Lines between these two markers are read and passed to the `usrmod` constructor as `mod_alvec` of type `vector<string>`. The two marker lines are passed as well so that the first user line is `mod_alvec[1]` and not `mod_alvec[0]`. In the distributed code, many examples use the two lines that set the simulation length via the variables `slen` and `spin`. Even if they are not simulation estimators as the example `elec`, these values might still be useful to determine the bootstrap simulation.

3.2.6 OBJFUN PARMFILE

The `objfun` constructor sees everything that the `usrmod` constructor sees plus the two vectors `obj_pfvec` and `obj_alvec` of type `vector<string>` that are defined in the OBJFUN PARMFILE block. The reason that `objfun` sees everything that `usrmod` sees is that `objfun` may have to construct an instance of `usrmod` to compute the objective function as does the maximum likelihood example `elec`.

obj_parmfile: This is the name of a file containing lines of the user's choosing. For our sample parmfile this file is `11114000.fit` which is the output parmfile written by SNP. This file is read and passed to the `objfun` constructor as the `std::vector` of `std::string` `obj_pfvec`. If there is no `obj_parmfile` then code `__none__` as the filename. In the distributed code, the SNP parmfile is read when the EMM objective function (`objtype=0`) is selected.

#begin additional lines, #end additional lines: Lines between these two markers are read and passed to the `usrmod` constructor as `obj_alvec` of type `vector<string>`. The two marker lines are passed as well so that the first user line is `obj_alvec[1]` and not `obj_alvec[0]`. In the distributed code, the EDF `objfun` (`objtype=1`) uses the variable `lag`.

3.2.7 PARAMETER START VALUES

The block labeled `PARAMETER START VALUES` specifies the first value for the chain. The simulation it generates must satisfy the support conditions; i.e. `sv_usrmod::gen_sim` must return true, `sv_usrmod::support` must return true, and `sv_usrmod::prior` must return `scl::dev_val.positive` = true for this initial value of ρ. The numbers to the right, 0 or 1, determine whether that element is held fixed or is active. If 0, then the proposal never moves that element of ρ. To the right of this 0 or 1 the user may add text such as the name of the parameter. New files `parmfile.fit`, `parmfile.end` and `parmfile.alt` are written as the MCMC chain progress with the current putative mode of the objective function replacing the values in `PARAMETER START VALUES` for `.fit` and `.alt` and the last value of ρ in the chain in the case of `.end`. The `parmfile.end` is used to recommence where one left off; `parmfile.fit` is used to recommence starting at the mode, which is what one usually wants to do; and `parmfile.alt` is used when switching to the conditional move proposal (`proptype`=1). If the number of parameters exceeds 20, then `parmfile.alt` will not be written. Once the mode has been found, it will not change. Therefore if a `parmfile.fit` is used to recommence and nothing in the parmfile is changed, then it may happen that the previous run is just reproduced. If the purpose of the new run is to try and improve the mode, then change `seed` in block `ESTIMATION DESCRIPTION` or use `parmfile.end`.

3.2.8 PARAMETER INCREMENTS

The block labeled `PARAMETER INCREMENTS` determines the grid for caching. These increments should be determined by scientific relevance and be as coarse as possible. This is the ideal, which may not be achievable as discussed below. The increments should be either integer or fractional powers of two.

These increments determine a grid for the parameters. The proposal will propose moves of size one or more grid increments. The probability assigned to an increment is proportional to the ordinate of the normal density with scale as specified in the `PROPOSAL SCALING` block. For the case when the proposal scale is the same as the parameter increment, the moves will be (-2, -1 ,0 ,1 ,2) with probabilities (0.0912128, 0.4087872, 0.0, 0.4087872, 0.0912128). When the proposal scale is half the parameter increment the moves will be (-1, 0, 1) with probabilities (0.5, 0.0, 0.5). With a group move, the number of possible moves grows exponentially with the group size. Thus, if increment and scale are set so that (-1, 0, 1) would be the possibilities for move-one-at-a-time, then there would be $3^{\text{len\_mod\_parm}}$ group move possibilities.

These facts limit the practical choice of grid increments. The minimum effective proposal scale is one grid increment. If this scale causes the rejection rate to be too large, then the increment must be reduced regardless of the scientific considerations that dictated the increment size. If the number of increments that are assigned positive probability becomes too large, then then there will be little improvement to speed. The fact that the number of possible moves increases exponentially with group size limits the size of groups that one can consider.

Here are some fractional powers of two.

```
5.00000000000000000e-01   =   0.50000000000000000000000
2.50000000000000000e-01   =   0.25000000000000000000000
1.25000000000000000e-01   =   0.12500000000000000000000
6.25000000000000000e-02   =   0.06250000000000000000000
3.12500000000000000e-02   =   0.03125000000000000000000
1.56250000000000000e-02   =   0.01562500000000000000000
7.81250000000000000e-03   =   0.00781250000000000000000
3.90625000000000000e-03   =   0.00390625000000000000000
1.95312500000000000e-03   =   0.00195312500000000000000
9.76562500000000000e-04   =   0.00097656250000000000000
4.88281250000000000e-04   =   0.00048828125000000000000
```

```
2.44140625000000000e-04   =   0.00024414062500000000000
1.22070312500000000e-04   =   0.00012207031250000000000
6.10351562500000000e-05   =   0.00006103515625000000000
3.05175781250000000e-05   =   0.00003051757812500000000
1.52587890625000000e-05   =   0.00001525878906250000000
7.62939453125000000e-06   =   0.00000762939453125000000
3.81469726562500000e-06   =   0.00000381469726562500000
1.90734863281250000e-06   =   0.00000190734863281250000
9.53674316406250000e-07   =   0.00000095367431640625000
4.76837158203125000e-07   =   0.00000047683715820312500
2.38418579101562500e-07   =   0.00000023841857910156250
1.19209289550781250e-07   =   0.00000011920928955078125
```

3.2.9 PROPOSAL SCALING

These are the standard deviations of the proposal. They should be roughly proportional to the standard errors of the estimate of ρ if such is known. If not, they can be easily determined as discussed in Section 6.

3.2.10 PROPOSAL GROUPING

How to specify group moves in the parmfile is discussed in Subsection 6.3. Briefly, in each matrix, the first element of the first row gives the relative probability with which this group is selected. In the remaining columns of the first row are the indexes of the parameters in that group. The first column is the same as the first row. The submatrix bounded by the first row and column is a correlation matrix. The multivariate normal to move the group is determined by this correlation matrix and the values in the `PROPOSAL SCALING` block. Within the `PROPOSAL GROUPING` block, the index of every parameter must be accounted for. Those parameters that are not moved (i.e. have a 0 to their right in the `PARAMETER START VALUES` block) are collected into a group that is assigned zero probability of being selected. If the `PROPOSAL GROUPING` block is not present, then one is synthesized. One can view EMM's interpretation of either the synthesized or user specified `PROPOSAL GROUPING` block in the file `detail.dat`, presuming `print=1` in the `ESTIMATION DESCRIPTION` block.

3.3 Directory Structure

The directory structure of the EMM distribution is as follows:

3.3.1 snpsrc

The directory `snpsrc` contains all source code for program `snp`. Classes that the user can modify are presented in `snpusr.h` and defined in `snpusr.cpp`.

3.3.2 emmsrc

The directory `emmsrc` contains all source code for program `emm`, excepting `emmusr.h` and `emmusr.cpp`, which contain headers and code for class `usrmod` and reside in their own directory.

3.3.3 svfx

This directory contains the code for stochastic volatility model applied to foreign exchange data that is used for an example in this *Guide*. For the `svfx` example, the class `usrmod` that the user must supply is presented in `emmusr.h` and defined in `emmusr.cpp`. Also present are makefiles and data for that example.

3.3.4 self

This directory contains the files `emmusr.h` and `emmusr.cpp` for fitting an SNP model model to itself by EMM. The GSM (General Scientific Models) package runs an MCMC subchain on an SNP statistical model. The easiest way to tune that chain is to fit SNP to itself using EMM. The relevant blocks from the EMM parmfile can be used directly in the GSM parmfile once the chain is tuned. The example in directory `self` is the one that appears in the *GSM User's Guide*. Also present are makefiles and data for that example.

3.3.5 elec

This directory contains the files `emmusr.h` and `emmusr.cpp` for fitting a translog electricity demand system by maximum likelihood. When it is run, set `objtype=1`.

3.3.6 var

This directory contains the files `emmusr.h` and `emmusr.cpp` for fitting a var by simulated method of moments. It's purpose is to illustrate how to code a user's objective function. When it is run, set `objtype=2`.

3.3.7 hab, lrr, pro

These directories contains the files `emmusr.h` and `emmusr.cpp` for fitting three asset pricing models by Bayesian EMM. They illustrate the use of prior information with an EMM objective function.

3.3.8 lib

The directory `lib` contains the three libraries `libscl`, `libsnp`, and `libsmm`. The source code is in `libscl/src`, the makefiles are in `libscl/gpp` or `libscl/ms`, depending on the compiler. Compilation is done within `libscl/gpp` for Linux or `libscl/ms` for Windows with the result that the library and headers reside in `libscl/gpp` or `libscl/ms` after compilation. Similarly for `libsnp`, and `libsmm`.

3.3.9 snprun

The directory `snprun` contains the makefile to build the `snp` executable and input files to run the example. Type `make` (or `nmake` for Microsoft) and the `snp` executable will be built and ready to run.

3.3.10 emmrun

The directory `emmrun` contains the makefile to build the `emm` executable and input files to run the example. Type `make` (or `nmake` for Microsoft) and the `emm` executable will be built and ready to run. This folder also contains certain key files described as follows.

control.dat: A file that contains the names of the input parmfile and the prefix for the output files. Here is an example of a one line `control.dat` file:

```
sv.parmfile.in0  sv
```

The input parmfile is named `sv.parmfile.in0` and all output files such as `detail.dat`, `rho.000.dat`, etc. are named `sv.detail.dat`, `sv.rho.000.dat`, etc. To prefix `control.dat` itself, execute `emm` with `sv.control.dat` as a command line argument, i.e.

```
emm sv.control.dat.
```

detail.dat: Voluminous detailed output from the run.

summary.dat: This file summarizes the output giving mean, mode, and standard errors, provided `kilse` $= 0$

rho.000.dat, stats.000.dat, pi.000.dat, reject.000.dat: The file `rho.000.dat` contains the MCMC chain for ρ. The file `stats.000.dat` contains the corresponding values of `stats`; Let τ denote the temperature parameter, let $\ell(\rho) = \exp(-ns_n(\rho))$, and let $p(\rho)$ denote the prior. The file `pi.000.dat` contains three items corresponding to the MCMC chain for ρ : (1) $\log \ell(\rho) + \log p(\rho)$. (2) $\log\, \ell(\rho)$. (3) $\log p(\rho)$. The file `reject.000.dat` contains a matrix whose first column contains the rejection rate for each parameter followed by the overall rejection rate. The other columns of `reject.000.dat` are discussed later. There will also be files `rho.001.dat`, `rho.002.dat`, etc. up to the limit specified by `nfile`.

emmcache.new: Written only if `max_cache_size`>0. If the run was successful, copy `sv.emmcache.new` to `sv.emmcache.dat`. If a mistake is made and a `emmcache` for a different project, different seed, different `spin` ($= N_0$), different `slen` ($= N$), etc., the `emm` will detect the error and reject the cache.

parmfile.**fit** : A copy of the input parmfile with the parameter start values replaced by the mode and scaling variables recomputed so that `incfac` and `sclfac` are 1.0. All else is the same as the input parmfile.

parmfile.**end** : A copy of the input parmfile with the parameter start values and seed replaced by the the last value of ρ and seed in the MCMC chain and scaling variables recomputed so that `incfac` and `sclfac` are 1.0. All else is the same as the input parmfile.

parmfile.**alt** : A copy of the input parmfile with the parameter start values replaced by the mode and scaling variables recomputed so that `incfac` and `sclfac` are 1.0. A `PROPOSAL GROUPING` block is inserted and `proptype` is put to 1. All else is the same as the input parmfile.

rho_mode, rho_mode, V_hat_hess, etc.: Statistics from the run in the form expected for reading with member `vecread` of class `realmat` in library `libscl`.

The user is free to modify the directory structure to suit the application, but the makefiles will need to be altered accordingly. We now proceed to the worked examples.

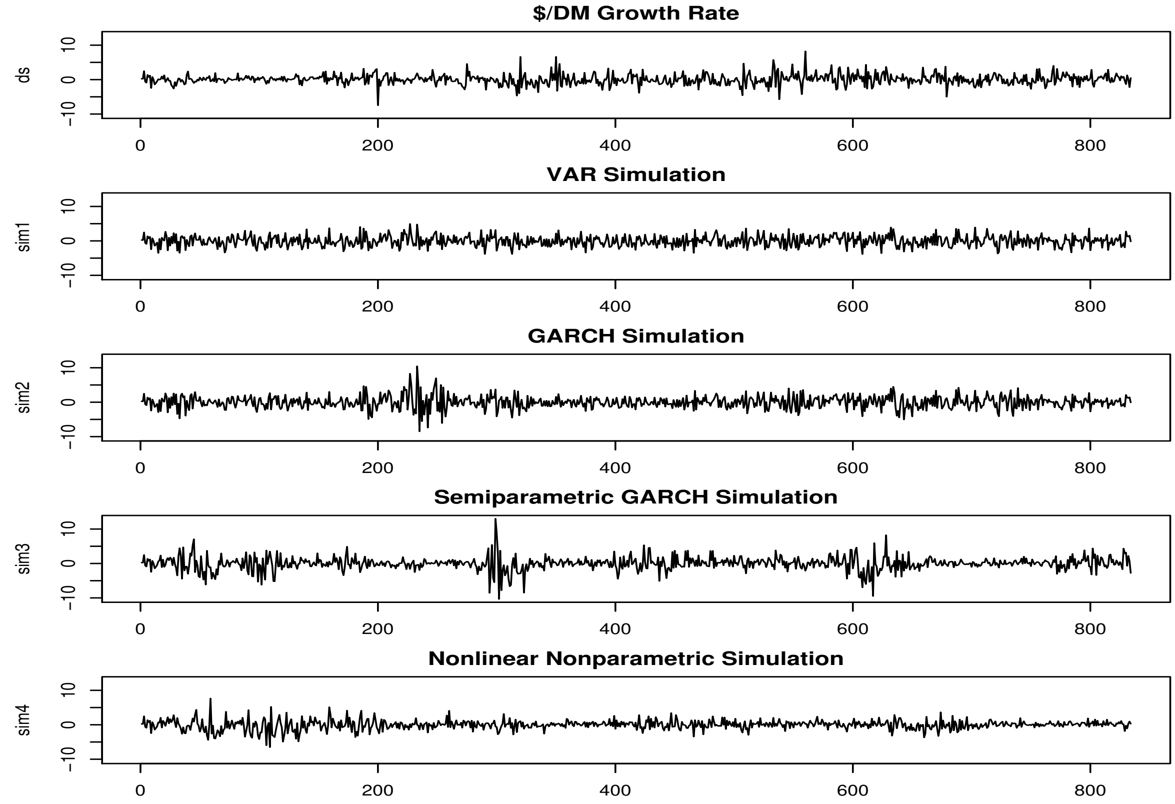

Figure 1. Changes in Weekly \$/DM Spot Exchange Rates, SNP-GARCH. The first panel is a plot of the data, which are each Friday's quote over the years 1975 to 1990 expressed as percentage change from the previous week. The second panel is a simulation from an SNP fit with $(L_u, L_g, L_r, L_p, K_z, I_z, K_x, I_x) = (1, 0, 0, 1, 0, 0, 0, 0)$; the third with (1,1,1,1,0,0,0,0), the fourth with (1,1,1,1,4,0,0,0), and the fifth with (1,1,1,1,4,0,1,0). The parameters L_v and L_w are set to zero. The parameters I_z, maxI_z, and I_x have no effect when $M = 1$. The parameter max$K_z = K_z$ in each instance that $K_x > 0$.

4 The Elementary Stochastic Volatility Model

In Section 6 we use the EMM method to estimate a simple form of the stochastic volatility model. The data set consists of 834 observations on the daily US dollar to German mark exchange rate over the years 1975 to 1990 expressed as a percentage change from the previous week. This is the same series used in the *SNP User's Guide*. The data together with simulations from SNP-GARCH fits are shown in Figure 1.

Let y_t denote the percent change. The stochastic volatility model with a leverage effect (correlation between return innovations and volatility innovations) is

$$\begin{aligned} u_{1t} &= z_{1t} && (3) \\ u_{2t} &= s\left(r\, z_{1t} + \sqrt{1-r^2}\, z_{2t}\right) && (4) \\ v_t - b_0 &= b_1(v_{t-1} - b_0) + u_{2t} && (5) \\ y_t - a_0 &= a_1(y_{t-1} - a_0) + \exp(v_t)\, u_{1t} && (6) \end{aligned}$$

where z_{1t}, z_{2t} are iid Gaussian random variables. The parameter vector is

$$\rho = (a_0, a_1, b_0, b_1, s, r)$$

Early references are Clark (1973) and Tauchen and Pitts (1983). More recent references are Gallant, Hsieh, and Tauchen (1991, 1997), Andersen (1994), and Durham (2006). See Shephard (2004) for more background and references.

There is controversy regarding the timing convention in equation (6) and the references above are not in agreement. The alternative timing convention is

$$y_t - a_0 = a_1(y_{t-1} - a_0) + \exp(v_{t-1})\, u_{1t} \tag{7}$$

which is consistent with an Euler discretization of the continuous time stochastic volatility model. See Yu (2005) for more details but be aware that his specifications do not include an autoregressive term to account for the well known slight predictability in daily returns so that his empirical results may not be relevant.

5 Fitting the Score Generator

The first step to implement the EMM estimator is to estimate the score generator model. Statistical efficiency requires that the score generator should provide a reasonably good statistical description of the data. We employ the SNP model described in Gallant and Tauchen (1992) and in the *SNP User's Guide* (Gallant and Tauchen, 2004a). Other score generators could be used, though this would require more coding.

Change directory to the folder `snprun` and make the snp executable. Instead of working through a full SNP specification search, which is described in detail for this example in the

SNP User's Guide, we show how to implement the estimator using the best score found there, namely `11114000.fit`.

The settings for the score generator are $L_u = 1$, $L_g = 1$, $L_r = 1$, $L_p = 1$, $K_z = 4$, $K_x = 0$. These settings define an AR(1) model for $\{y_t\}$ with a GARCH(1,1) conditional scale function and a time homogeneous nonparametric innovation density with fat tails accommodated via $K_z = 4$. The dependence on the past is through the linear location function and GARCH scale function. This model is optimal under the Schwarz criterion. It defines the EMM criterion function for estimation of the stochastic volatility model.

6 Worked Example

We begin with an example to show the user the basics of running the program. The purpose of these sample runs is to show the user the basic aspects of using the package. At the outset, to make runs execute quickly, the simulation size, N, and the length of the initial period to let transients die off, N_0, are set to low values. They will be set larger after the MCMC chain has been tuned. In the code N and N_0 are called `slen` and `spin` respectively.

6.1 User-Supplied Classes and Files

The user supplies the C++ class `usrmod`, a control file, and an input parameter file. The basic task of `usrmod` to implement the data generation process $\rho \mapsto \{\hat{y}_\tau(\rho), x_{\tau-1}(\rho)\}_{\tau=1}^{N}$, which takes a parameter vector ρ and generates a simulated realization from the structural model. In the example $\rho = (a_0, a_1, b_0, b_1, s, r)$.

To keep a record of our work, we will separately name each control file which means we will have to use a command line argument when we execute `emm`. The control file is called `sv.ctrl.000`. In our example, this has one line, which is

```
sv.parm.000  sv
```

Here, `sv.parm.000` is the name of the input parameter file and `sv` is the prefix to be added to all output files. The control file can have additional similar lines which allow different parmfiles for the same or different projects to be run at one time. The prefix must be different for every line or results will be overwritten.

In the example, the input parameter file `parm.000` is

```
ESTIMATION DESCRIPTION (required)
    SpotRate   Project name, pname, char*
         2.6   EMM version, defines format of this file, emmver, float
           0   Objfun type, 0 EMM, 1 MLE, 2 usr, type, int
           0   Proposal type, 0 group_move, 1 cond_move, 2 usr, proptype, int
           1   Write detailed output if print=1, int
         457   Seed for simulations, iseed, int
        1000   Number of MCMC simulations per file, lchain, int
           5   Number of MCMC simulation files, nfile, int
         1.0   Rescale proposal scaling by this value, sclfac, float
         1.0   Rescale parameter increments by this value, incfac, float
         1.0   Rescale objfun by this value, temperature, float
           1   Sandwich variance not computed if kilse=1, int
           1   The stride used to write MCMC simulations, stride, int
           0   Draw from prior if draw_from_prior=1, int
           0   Max cache size, max_cache_size, int
DATA DESCRIPTION (required) (mod and obj constructors see realmat data(M,n))
           1   Dimension of the data, M, int
         834   Number of observations, n, int
dmark.dat      File name, any length, no embedded blanks, dsn, string
4              Read these white space separated fields, fields, intvec
MODEL DESCRIPTION (required)
           6   Number of model parameters, num_mod_parms, int
           8   Number of model functionals, num_mod_funcs, int
MODEL PARMFILE (required) (constructor sees as vector<string> pfvec, alvec)
__none__       File name, code __none__ if none, mod_parmfile, string
#begin additional lines
        5000   Number of observations in simulated data, slen (=N), int
         500   Initial simulations to eliminate transients, spin, int
#end additional lines
OBJFUN PARMFILE (required) (constructor sees as vector<string> pfvec, alvec)
11114000.fit   File name, code __none__ if none, obj_parmfile, string
#begin additional lines
#end additional lines
PARAMETER START VALUES (required)
   0.05    1
   0.0     1
   1.5     1
   0.0     1
   0.1     1
   0.0     1
PROPOSAL SCALING (required)
   1.9531250000000000e-03
   1.9531250000000000e-03
   1.9531250000000000e-03
   1.9531250000000000e-03
   1.9531250000000000e-03
   1.9531250000000000e-03
```

This file was described in detail in Subsection 3.2 and there is little more about it that needs to be said here except where the numbers came from. The parameter start values are set so the stochastic volatility model will reproduce the mean and variance of the data. We could have done much better but we'll see if `emm` can recover from our laziness.

The proposal scalings are set to a power of two near 0.001. This is just a guess. Once again we are being lazy. When in doubt, err by setting these values too small.

Looking at the header `emmusr.h` in Subsection 3.1, we see that there are five member functions that need to be written: the constructor, the private member that does the simulation, the member that breaks a simulation up into bootstrap samples, the member that checks that a proposed ρ satisfies the model's support conditions, and the prior. Specifically, code must be supplied in `emmusr.cpp` to implement these lines from `emmusr.h`

```
sv_usrmod
  (const scl::realmat& dat, INTEGER len_mod_parm, INTEGER len_mod_func,
   const std::vector<std::string>& mod_pfvec,
   const std::vector<std::string>& mod_alvec,
   std::ostream& detail);
bool gen_bootstrap(std::vector<scl::realmat>& bs);
bool support(const scl::realmat& rho);
libsmm::den_val prior
  (const scl::realmat& rho, const scl::realmat& stats);
```

Everything else has been coded in the header `emmusr.h`.

The job of the constructor is to initialize the private members of usrmod by parsing `mod_pfvec` and `mod_alvec` We will not write anything to the output stream `detail`. Here is the constructor

```
emm::sv_usrmod::sv_usrmod
  (const realmat& dat, INTEGER len_mod_parm, INTEGER len_mod_func,
   const std::vector<std::string>& mod_pfvec,
   const std::vector<std::string>& mod_alvec,
   std::ostream& detail)
: data(dat), lrho(6), lstats(8), variable_seed(740726)
{
  vector<string>::const_iterator usr_ptr = mod_alvec.begin();
  ++usr_ptr;
  slen = atoi((usr_ptr++)->substr(0,12).c_str());
  spin = atoi((usr_ptr++)->substr(0,12).c_str());

  if (lrho != len_mod_parm) {
    error("Error, usrmod, constructor, len_mod_parm is set wrong in parmfile");
  }

  if (lstats != len_mod_func) {
    error("Error, usrmod, constructor, len_mod_parm is set wrong in parmfile");
  }
}
```

We will compute four stats for the generated data, min, max, mean, and standard deviation, and the same four for the latent volatility factor; a total of eight. Otherwise the code that implements `sv_simulate` below is a straightforward implementation of Equations 4 through 6 with precaution taken to force class `mcmc` of `mcmc` to reject the simulation when the `exp` function overflows by returning false when it happens. The documentation for the matrix class `realmat` is in its header `realmat.h` which is in the `libscl` distribution. Here is the code.

```
bool emm::sv_usrmod::sv_simulate(INT_32BIT& seed, INTEGER len,
  realmat& sim, realmat& stats, realmat& latent)
{
  if (sim.get_rows() != 1 && sim.get_cols() != len) {
    sim.resize(1,len);
  }
  if (latent.get_rows() != 1 && latent.get_cols() != len) {
    latent.resize(1,len);
  }

  REAL a0 = rho[1];
  REAL a1 = rho[2];
  REAL b0 = rho[3];
  REAL b1 = rho[4];
  REAL s  = rho[5];
  REAL r  = rho[6];

  REAL rr2 = sqrt(1.0 - pow(r,2));

  REAL vlag = 0.0;
  REAL ylag = 0.0;
  errno = 0;

  for (INTEGER t=1; t<=spin; ++t) {
    REAL z1 = unsk(seed);
    REAL z2 = unsk(seed);
    REAL u1 = z1;
    REAL u2 = s*(r*z1 + rr2*z2);
    REAL v = b0 + b1*(vlag - b0) + u2;
    REAL y = a0 + a1*(ylag - a0) + u1*exp(v); // or u1*exp(vlag)
    vlag = v;                                 // see User's Guide
    ylag = y;
  }

  if (errno == ERANGE) return false;

  REAL ymin = REAL_MAX;
  REAL ymax = -REAL_MAX;
  REAL ymean = 0.0;
  REAL ysdev = 0.0;
  REAL vmin = REAL_MAX;
  REAL vmax = -REAL_MAX;
  REAL vmean = 0.0;
  REAL vsdev = 0.0;

  for (INTEGER t=1; t<=len; ++t) {
    REAL z1 = unsk(seed);
    REAL z2 = unsk(seed);
    REAL u1 = z1;
    REAL u2 = s*(r*z1 + rr2*z2);
    REAL v = b0 + b1*(vlag - b0) + u2;
    REAL y = a0 + a1*(ylag - a0) + u1*exp(v); // or u1*exp(vlag)
    vlag = v;                                 // see User's Guide
    ylag = y;
    sim[t] = y;
    latent[t] = v;
    ymin = y < ymin ? y : ymin;
    ymax = y > ymax ? y : ymax;
```

```
    ymean += y;
    ysdev += pow(y,2);
    vmin = v < vmin ? v : vmin;
    vmax = v > vmax ? v : vmax;
    vmean += v;
    vsdev += pow(v,2);
  }

  if (errno == ERANGE) return false;

  if (lstats != 8) error("Error, emmusr, wrong size for stats");
  if (stats.size() != lstats) stats.resize(lstats,1);

  ymean = ymean/REAL(len);
  ysdev = sqrt( (ysdev - REAL(len)*pow(ymean,2))/REAL(len) );

  vmean = vmean/REAL(len);
  vsdev = sqrt( (vsdev - REAL(len)*pow(vmean,2))/REAL(len) );

  stats[1] = ymin;
  stats[2] = ymax;
  stats[3] = ymean;
  stats[4] = ysdev;
  stats[5] = vmin;
  stats[6] = vmax;
  stats[7] = vmean;
  stats[8] = vsdev;

  return true;
}
```

We have two support conditions to check. The absolute value of r, which is `rho[6]`, must be less than one. Also, we will require that s, which is `rho[5]`, be positive. Here is the code that checks them. We could also force the autoregressive parameters a_1 and b_1 to have absolute value less than one, but this usually is not necessary for EMM as discussed in Tauchen (1998).

```
bool emm::sv_usrmod::support(const realmat& parm)
{
  if (parm[5] < 0.0) return false;
  if (fabs(parm[6]) > 1.0) return false;
  return true;
}
```

We have no reason to reject a completed simulation other than `exp` overflow, which we have already checked for during the simulation itself, so the prior is trivial. Here it is.

```
den_val emm::sv_usrmod::prior(const realmat& parm, const realmat& stats)
{
  return den_val(true, 0.0);
}
```

If we had wanted to reject simulations for which, say, `vmax` was too large, we could code something like this

```
den_val emm::sv_usrmod::prior(const realmat& parm, const realmat& stats)
{
  if (stats[6] > 1.0e+10) return den_val(false, -REAL_MAX);
  return den_val(true, 0.0);
}
```

`REAL_MAX` is the largest value that type `REAL` can hold. Putting `-REAL_MAX` as the second member of `struct den_val` has no effect if `den_val`'s first member is checked before the second is used, as it should be. If one forgets to do so, `-REAL_MAX` approximates the log of 0. The documentation is in header `scltypes.h` which is in the `libscl` distribution. With the prior written thus, the estimation would be subject to the constraint that log volatility never exceeds 1.0e+10. By using stats and prior in this fashion, it is easy to estimate models subject to potentially quite complicated nonlinear constraints.

6.2 Running the Program

The EMM program runs alongside the SNP package, which is described in the *SNP User's Guide*. The `makefile` supplied with the example will build the executable for the example, though it might need to be edited to suit the user's preferences and system. To build it, change directory to `emmrun`, copy makefile.gpp to makefile, and enter the command `make`. The executable program built by the `makefile` is `emm`. We now execute `emm` by typing

```
./emm sv.ctrl.000
```

We get a warning messages.

```
Warning, emm_objfun, 11114000.fit value for toler invalid, reset to root EPS
```

The warning is because the value of `toler` in the SNP parmfile, while correct for SNP, is usually not correct for use in computing $\mathcal{I}$ in class `asymptotics` so program `emm` resets it. This message can be disregarded.

EMM writes the following set of files. There would be more if `max_cache_size` > 0 and `kilse`=0 .

```
sv.V_hat_hess.dat  sv.pi.004.dat      sv.rho.001.dat     sv.stats.001.dat
sv.detail.dat      sv.pi.005.dat      sv.rho.002.dat     sv.stats.002.dat
sv.parmfile.alt    sv.reject.000.dat  sv.rho.003.dat     sv.stats.003.dat
sv.parmfile.fit    sv.reject.001.dat  sv.rho.004.dat     sv.stats.004.dat
sv.pi.000.dat      sv.reject.003.dat  sv.rho.005.dat     sv.stats.005.dat
sv.pi.001.dat      sv.reject.004.dat  sv.rho_mean.dat    sv.summary.dat
sv.pi.002.dat      sv.reject.005.dat  sv.rho_mode.dat    sv.diagnostics.dat
sv.pi.003.dat      sv.rho.000.dat     sv.stats.000.dat
```

The most informative diagnostic is a plot of the MCMC chain for ρ in the files `rho.000.dat` through `rho.005.dat` together with a plot of the first row of files `pi.000.dat` through `pi.005.dat`. The first row is

$$\pi = -\texttt{temperature} \times n \times s_n(\rho, \tilde{\theta}_n) + \log(\texttt{prior}).$$

The next two rows are $-n \times s_n(\rho, \tilde{\theta}_n)$ and $\log(\texttt{prior})$. Figure 2 plots these files at every tenth point.

The file `sv.reject.000.dat` looks like this:

	Col 1	Col 2	Col 3	Col 4
Row 1	0.039548	0.17700	7.00000	177.000
Row 2	0.072368	0.15200	11.00000	152.000
Row 3	0.17365	0.16700	29.00000	167.000
Row 4	0.0	0.16900	0.0	169.000
Row 5	0.042781	0.18700	8.00000	187.000
Row 6	0.020270	0.14800	3.00000	148.000
Row 7	0.058000	1.00000	58.00000	1000.00

The fourth column gives the number of moves for each parameter with the last row being the total. The third column gives the number of rejections. The second column is the proportion that each parameter was moved; i.e. the elements of the fourth column divided by the last element of the fourth column. The first column gives the rejection rates, by parameter and in total; i.e. the elements of the third column divided by the corresponding elements of the fourth column.

It would be more useful to have the aggregate of the files `reject.000.dat` through `reject.005.dat`. There is a program `reject.cpp` in directory `utility` with usage "`reject sv.reject.*.dat > reject.out`" that does this. The file `reject.out` looks like this:

	Col 1	Col 2	Col 3	Col 4
Row 1	0.011976	0.16700	12.00000	1002.00
Row 2	0.052156	0.16617	52.00000	997.000
Row 3	0.13126	0.16633	131.000	998.000
Row 4	0.0	0.17183	0.0	1031.00
Row 5	0.038974	0.16250	38.00000	975.000
Row 6	0.030090	0.16617	30.00000	997.000
Row 7	0.043833	1.00000	263.000	6000.00

Rejection rates are usually smaller when one is far from the mode than when one is near it. One way to judge this is by inspecting plots such as Figure 2. The R code, `plot.r`, that produced them is included in the distribution. The lines in the plots are too smooth. They should be choppier. The rejection rates need to be increased by increasing the scale of the proposal.

We continue the chain by copying `sv.parmfile.fit` to `sv.parm.001` and changing `sclfac` from 1.0 to 2.0 in `sv.parm.001`. We copy `sv.ctrl.000` to `sv.ctrl.001` and change `sv.parm.000` to `sv.parm.001` in `sv.ctrl.001`. Finally, we type ./emm `sv.ctrl.001` at the command prompt.

This run produced results much as above. The overall rejection rate went up to 0.067 and π moved from -164.0 to -47.29. We'll do the same as the above and increase the proposal scaling by 4.0. Recall that `sv.parmfile.fit` readjusts the proposal scaling so that `sclfac` is 1.0 before we change it to 4.0. Therefore we are now at 8.0 times our original proposal scaling.

The results are shown in Figure 3. The aggregate rejection rates from files `reject.000` through `reject.005` are

```
               Col  1      Col  2      Col  3      Col  4

Row  1       0.20858     0.16700     209.000     1002.00
Row  2       0.15246     0.16617     152.000     997.000
Row  3       0.11323     0.16633     113.000     998.000
Row  4       0.25606     0.17183     264.000     1031.00
Row  5       0.30872     0.16250     301.000     975.000
Row  6      0.088265     0.16617    88.00000     997.000
Row  7       0.18783     1.00000     1127.00     6000.00
```

which are reasonable except for the last, which is r.

We double the scale of the last parameter and run again. The results are shown in Figure 4. The aggregate rejection rates from files `reject.000.dat` through `reject.005.dat` are

```
               Col  1      Col  2      Col  3      Col  4

Row  1       0.22255     0.16700     223.000     1002.00
Row  2       0.15246     0.16617     152.000     997.000
Row  3       0.10521     0.16633     105.000     998.000
Row  4       0.34045     0.17183     351.000     1031.00
Row  5       0.32615     0.16250     318.000     975.000
Row  6       0.12538     0.16617     125.000     997.000
Row  7       0.21233     1.00000     1274.00     6000.00
```

The autocorrelation functions are shown in Figure 5. They appear reasonable but do indicate that longer chains will be required for reliable inference.

We increase the number of MCMC simulations per file, `lchain`, to 5000 and run again.

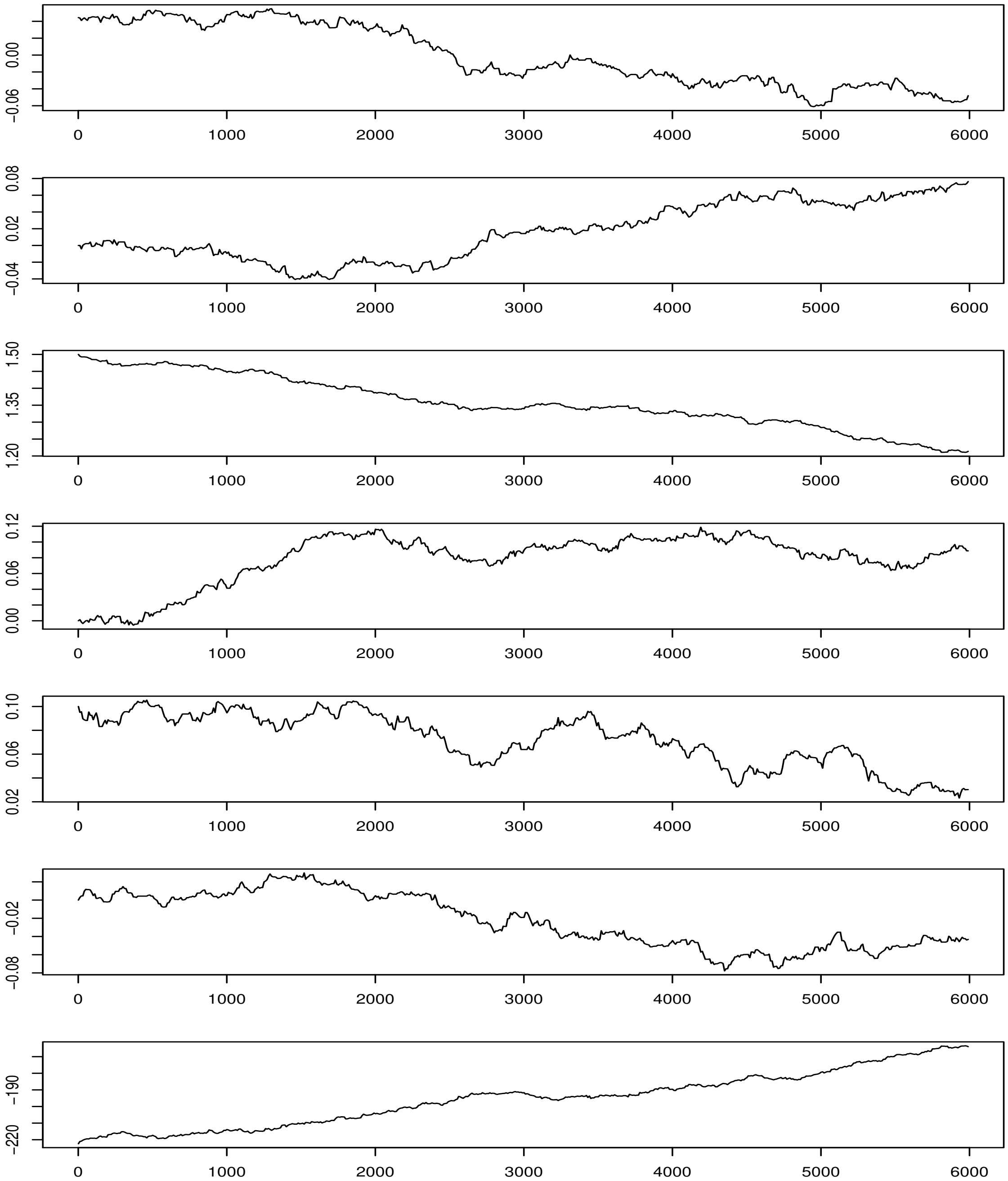

Figure 2. MCMC Chain from Parameter File sv.parm.000. The panels are from top to bottom a_0, a_1, b_0, b_1, s, r, and π. Every tenth point is plotted. $R = 6000$.

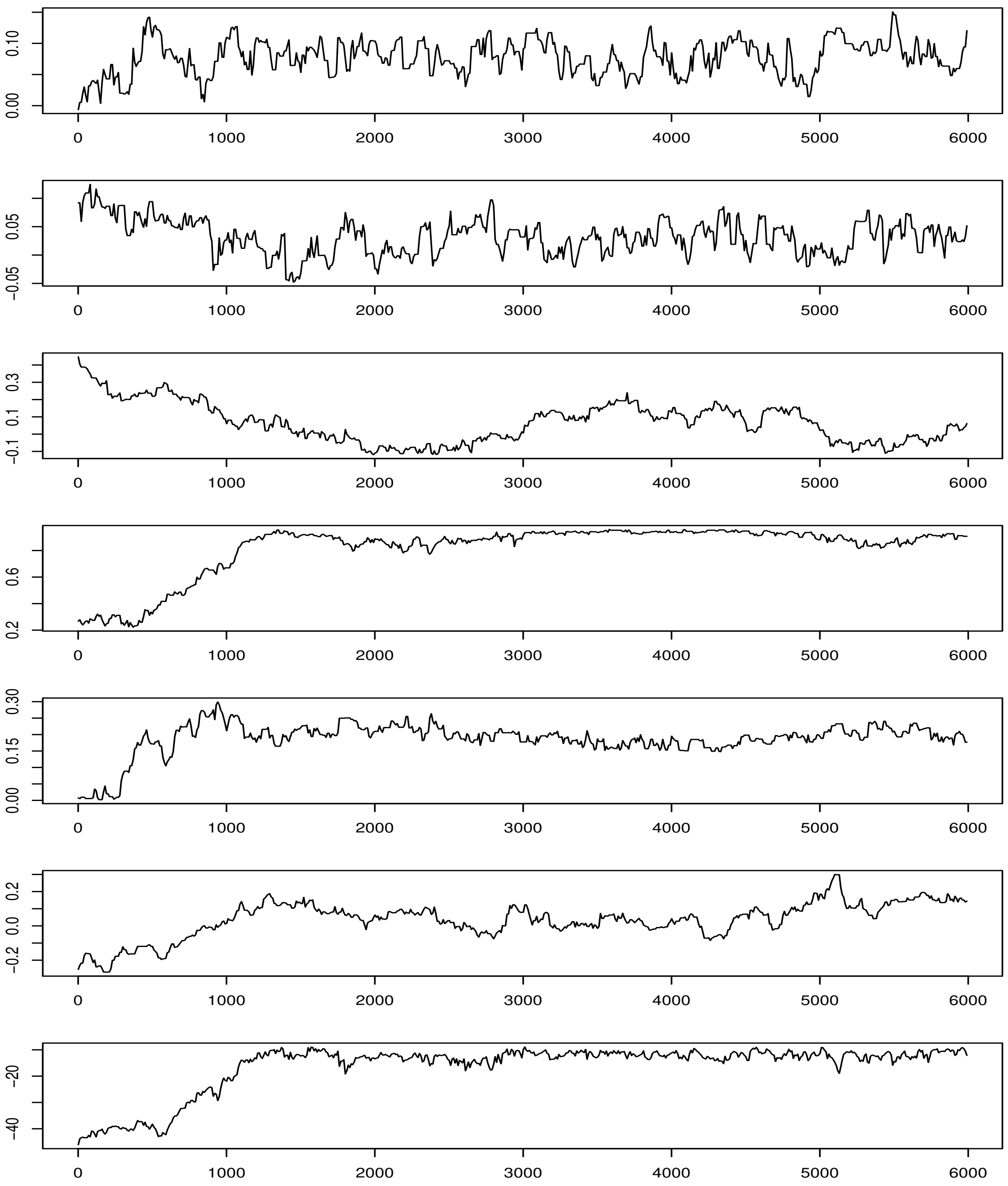

Figure 3. MCMC Chain from Parameter File sv.parm.002. The panels are from top to bottom a_0, a_1, b_0, b_1, s, r, and π. Every tenth point is plotted. $R = 6000$.

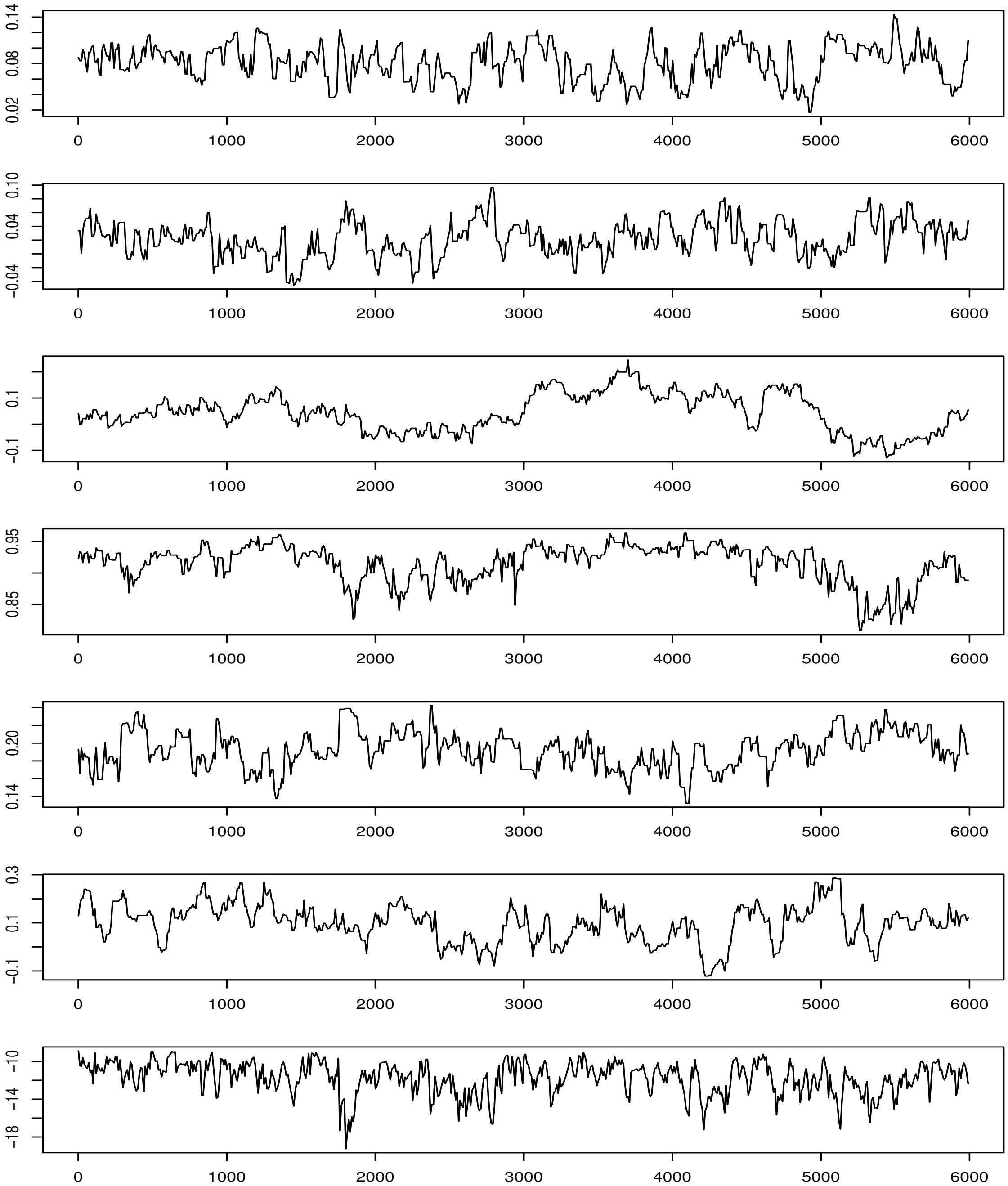

Figure 4. MCMC Chain from Parameter File sv.parm.003. The panels are from top to bottom a_0, a_1, b_0, b_1, s, r, and π. Every tenth point is plotted. $R = 6000$.

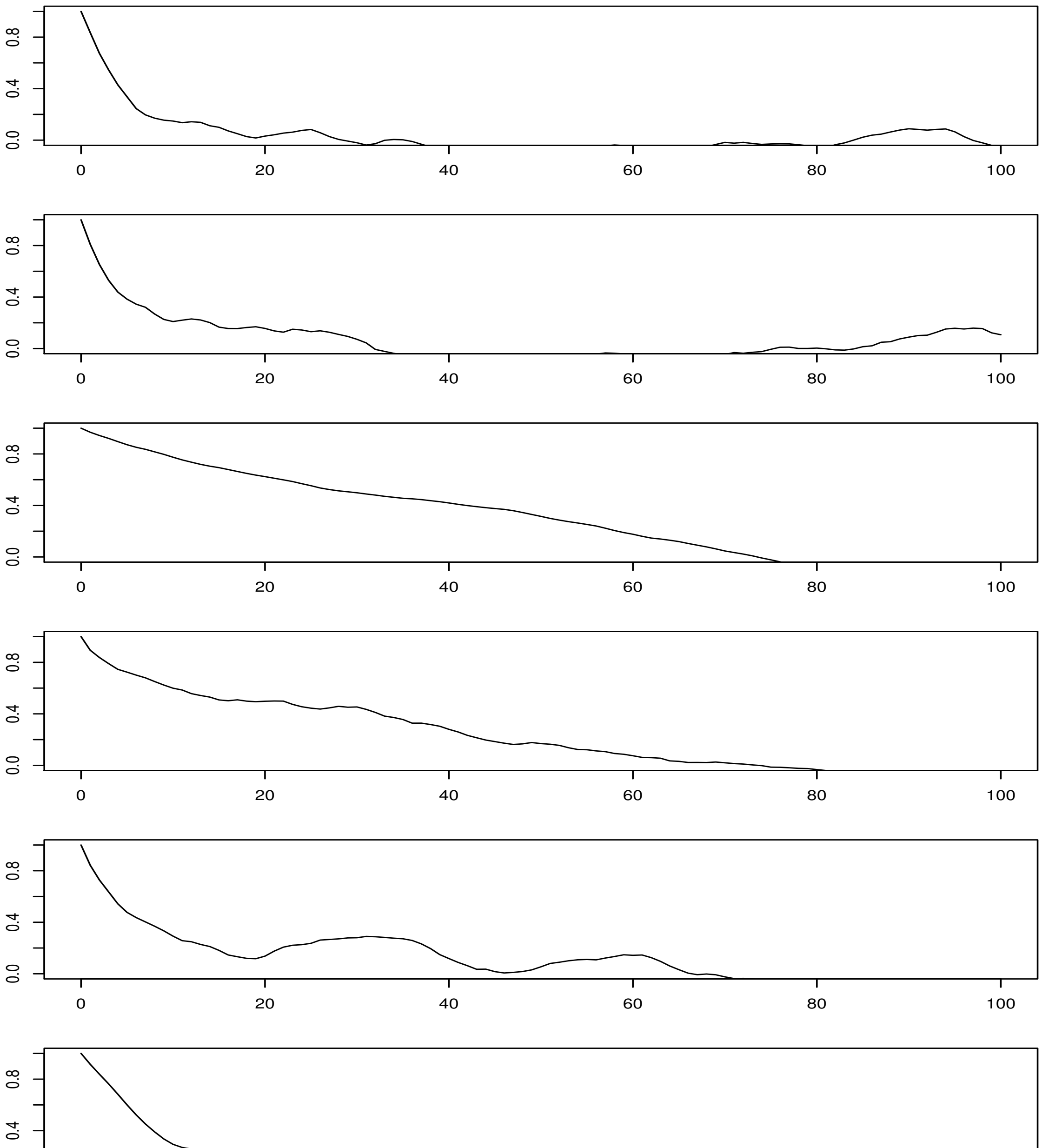

Figure 5. Autocorrelation of Chain from Parameter File sv.parm.003. The panels are from top to bottom a_0, b_0, b_1, c_1, s, and r. Every tenth point is sampled; lag 100 of sampled chain corresponds to lag 1000 of original chain. $R = 6000$.

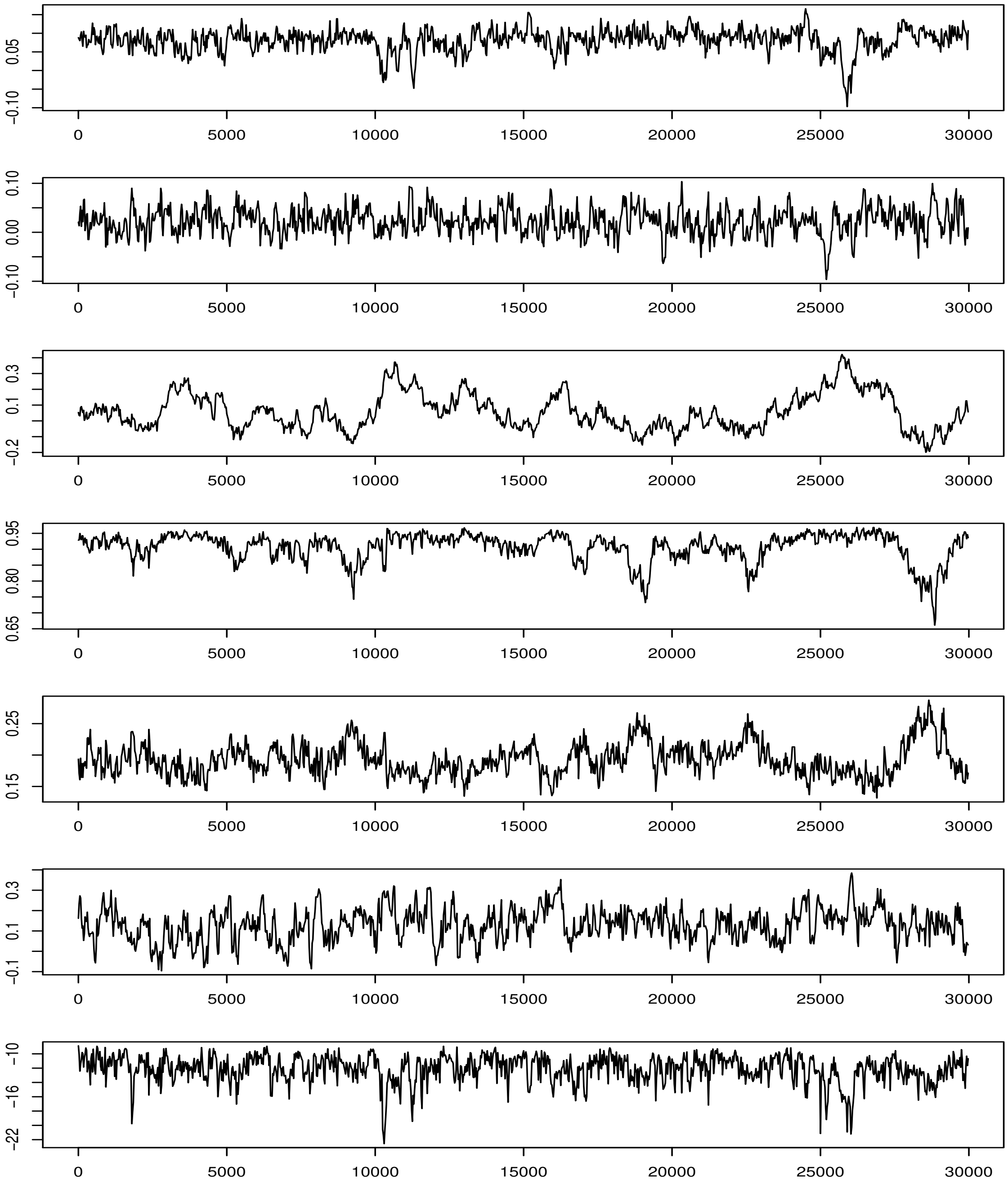

Figure 6. MCMC Chain from Parameter File sv.parm.004. The panels are from top to bottom a_0, a_1, b_0, b_1, s, r, and π. Every twenty-fifth point is plotted. $R = 30000$.

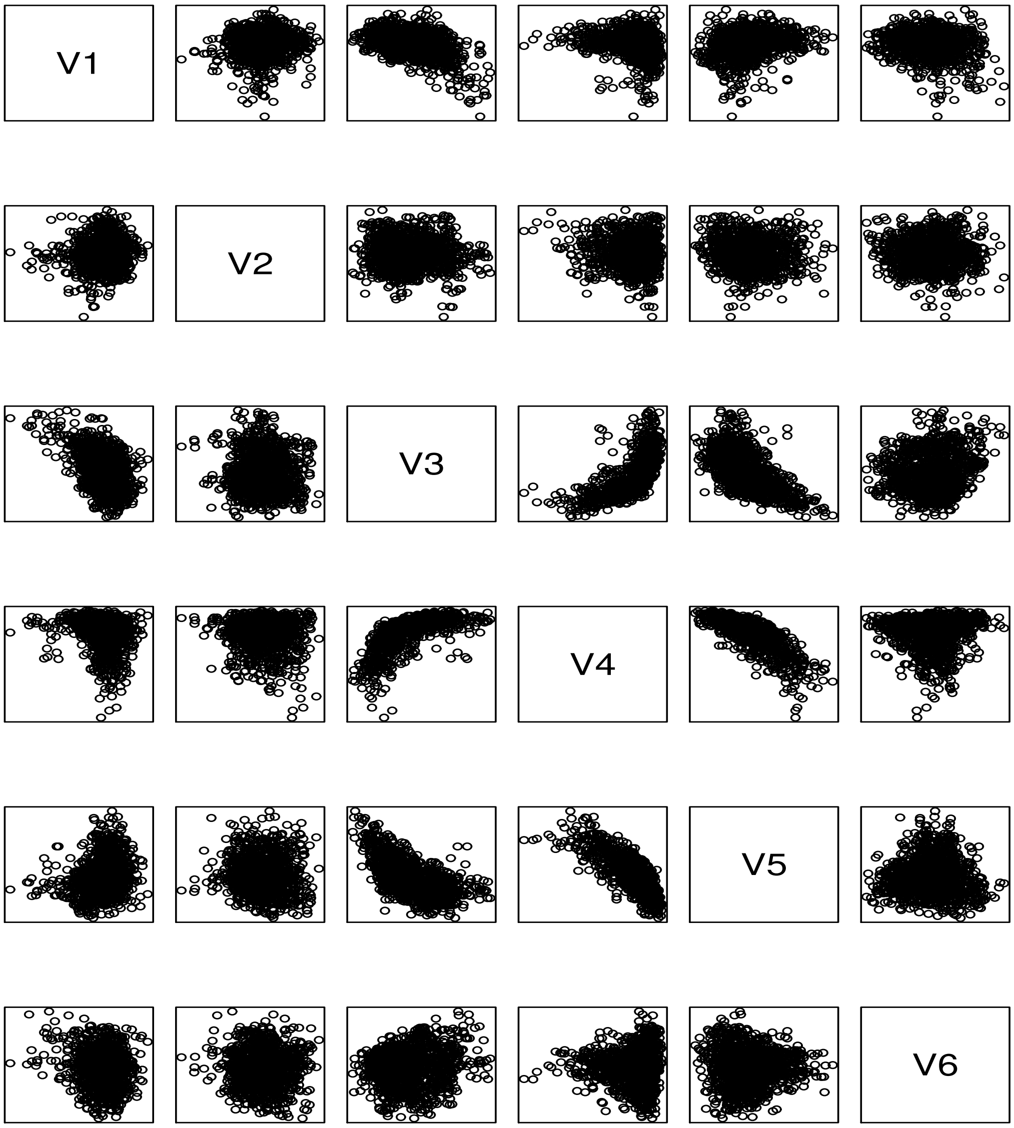

Figure 7. Scatter Plots of Chain from Parameter File sv.parm.004. The variables V1 through V6 are a_0, a_1, b_0, b_1, s, and r respectively. Every twenty-fifth point is plotted. $R = 30000$.

6.3 Group Move Proposal

The longer chain is shown in Figure 6, The chain looks like it is starting to stabilize. The rejection rates are about the same as above. We will next illustrate how the group move proposal feature of the EMM package is used.

We need to look the correlation matrix and a plot of all possible pairs of scatter plots, Figure 7, to see if group moves are actually needed. We are using R for the graphics and the plotting code is included with the distribution. R is public domain software that runs on nearly any platform that is available at `www.r-project.org`.

The correlation matrix gotten with R by sampling at the rate of every twenty-fifth point and putting all correlations that are less than 0.6 to zero is

```
    V1 V2         V3         V4         V5 V6
V1   1  0  0.0000000  0.0000000  0.0000000  0
V2   0  1  0.0000000  0.0000000  0.0000000  0
V3   0  0  1.0000000  0.6738761 -0.6400874  0
V4   0  0  0.6738761  1.0000000 -0.7926057  0
V5   0  0 -0.6400874 -0.7926057  1.0000000  0
V6   0  0  0.0000000  0.0000000  0.0000000  1
```

There is a relationship among variables V3, V4, and V5, which are parameters b_1, s, and r. The pairwise plots of these three indicate that it is not a strictly linear, tightly packed relationship. The chain looks to be excellent. There really is no need for grouping here.

The fact that we do not need grouping is no accident. The form of the stochastic volatility model used was chosen to minimize correlations. We wrote

$$v_t = b_0 + b_1(v_{t-1} - b_0) + u_{2t}$$

instead of

$$v_t = b_0 + b_1 v_{t-1} + u_{2t}$$

so that b_0 and b_1 could move independently of each another. Similarly for y_t. We wrote

$$u_{2t} = s\left(r\,z_{1t} + \sqrt{1-r^2}\,z_{2t}\right)$$

instead of

$$u_{2t} = r_{21}z_{1t} + r_{22}z_{2t}$$

so that the scale parameters s and r could move independently of each other.

Nonetheless, we will construct a group-move proposal to illustrate how it is done. We have to construct a `PROPOSAL GROUPING` block to put in the parmfile. It looks like this

```
PROPOSAL GROUPING (optional) (frequencies are relative)
 0.1    1
   1  1.0
 0.1    2
   2  1.0
 0.2      3     4     5
   3   1.0   0.7  -0.6
   4   0.7   1.0  -0.8
   5  -0.6  -0.8   1.0
 0.1    6
   6  1.0
```

Each sub-block is a matrix, which defines a group. The number in the (1,1) position is the relative frequency with which that group is to be sampled. Continuing down the first column are the indexes of the variables in the group; continuing along the first row are these same indexes. Filling in the rest of the matrix is the correlation matrix for this group. The values in the `PROPOSAL SCALING` block are used as standard deviations to convert these correlation matrices to variance matrices. Note that every variable must be listed and be in exactly one group. If some variables are fixed by coding 0's to the right of the values in the `PARAMETER START VALUES` block, `emm` will pull them out of the groups and put them in a separate group that is moved with relative frequency 0.0.

6.4 Putting Parameters on a Grid

We shall also take this opportunity to put the parameters on a grid by adding a `PARAMETER INCREMENTS` block to the parameter file. As mentioned in Subsection 3.2, putting parameter increments equal to the parameter scaling allows moves of two increments up or two down and putting it twice as large allows one move up or down. This is what we shall do.

Also, we put `max_cache_size=100000` in the `ESTIMATION DESCRIPTION` block because if we did not set it to a positive value, then putting parameters on a grid is pointless. If one neglects this step, then `max_cache_size` will be automatically set to 50000.

The complete parmfile (sv.parm.005) looks like this.

```
PARMFILE HISTORY (optional)
#
# This parmfile was written by EMM Version 2.5 using the following line from
# control.dat, which was read as char*, char*
# ---------------------------------------------------------------------------
#     sv.parm.004                sv
```

```
# ---------------------------------------------------------------------------
#
ESTIMATION DESCRIPTION (required)
    SpotRate   Project name, pname, char*
         2.6   EMM version, defines format of this file, emmver, float
           0   Objfun type, 0 EMM, 1 MLE, 2 usr, objtype, int
           0   Proposal type, 0 group_move, 1 cond_move, 2 usr, proptype, int
           1   Write detailed output if print=1, int
         457   Seed for MCMC simulations, iseed, int
        5000   Number of MCMC simulations per file, lchain, int
           5   Number of MCMC simulation files beyond the first, nfile, int
         1.0   Rescale proposal scaling by this value, sclfac, float
         1.0   Rescale parameter increments by this value, incfac, float
         1.0   Rescale objfun by this value, temperature, float
           1   Sandwich variance not computed if kilse=1, int
           1   The stride used to write MCMC simulations, stride, int
           0   Draw from prior if draw_from_prior=1, int
      100000   Max cache size, max_cache_size, int
DATA DESCRIPTION (required) (mod and obj constructors see realmat data(M,n))
           1   Dimension of the data, M, int
         834   Number of observations, n, int
dmark.dat      File name, any length, no embedded blanks, dsn, string
4              Read these white space separated fields, fields, intvec
MODEL DESCRIPTION (required)
           6   Number of modal parameters, len_mod_parm, int
           8   Number of model functionals, len_mod_func, int
MODEL PARMFILE (required) (constructor sees as vector<string> pfvec, alvec)
__none__       File name, code __none__ if none, mod_parmfile, string
#begin additional lines
        5000   Number of observations in simulated data, slen (=N), int
         500   Initial simulations to eliminate transients, spin, int
#end additional lines
OBJFUN PARMFILE (required) (constructor sees as vector<string> pfvec, alvec)
11114000.fit   File name, code __none__ if none, obj_parmfile, string
#begin additional lines
#end additional lines
PARAMETER START VALUES (required)
   8.14895773998386974e-02    1
   2.60176173224806462e-02    1
   7.28064108382113773e-02    1
   9.34458288546352378e-01    1
   1.82976626342656323e-01    1
   1.54595167239619968e-01    1
PROPOSAL SCALING (required)
   1.56250000000000000e-02
   1.56250000000000000e-02
   1.56250000000000000e-02
   1.56250000000000000e-02
   1.56250000000000000e-02
   3.12500000000000000e-02
PARAMETER INCREMENTS (optional) (fractional powers of two recommended)
   1.56250000000000000e-02
   1.56250000000000000e-02
   3.12500000000000000e-02
   3.12500000000000000e-02
   3.12500000000000000e-02
   3.12500000000000000e-02
PROPOSAL GROUPING (optional) (frequencies are relative)
 0.1    1
   1  1.0
 0.1    2
   2  1.0
```

```
0.2     3     4     5
  3   1.0   0.7  -0.6
  4   0.7   1.0  -0.8
  5  -0.6  -0.8   1.0
0.1    6
  6  1.0
```

From the file `sv.detail.dat` we can see what the proposal looks like and the cache hit rate.

```
    **********************************************************************
    *                                                                    *
    *                      grid_group_move proposal                      *
    *                                                                    *
    **********************************************************************

Probability select group 0 is 0.2
Group 0 density function is:
prob      support in increments
0.0912128 -2
0.4087872 -1
0.0000000  0
0.4087872  1
0.0912128  2

Probability select group 1 is 0.2
Group 1 density function is:
prob      support in increments
0.0912128 -2
0.4087872 -1
0.0000000  0
0.4087872  1
0.0912128  2

Probability select group 2 is 0.4
Group 2 density function is:
prob      support in increments
0.0000000 -1 -1 -1
0.0088811 -1 -1  0
0.3337195 -1 -1  1
0.0001222 -1  0 -1
0.0799762 -1  0  0
0.0007093 -1  0  1
0.0000036 -1  1 -1
0.0000006 -1  1  0
0.0000000 -1  1  1
0.0000000  0 -1 -1
0.0036870  0 -1  0
0.0575158  0 -1  1
0.0153848  0  0 -1
0.0000000  0  0  0
0.0153848  0  0  1
0.0575158  0  1 -1
0.0036870  0  1  0
0.0000000  0  1  1
0.0000000  1 -1 -1
0.0000006  1 -1  0
0.0000036  1 -1  1
0.0007093  1  0 -1
0.0799762  1  0  0
0.0001222  1  0  1
0.3337195  1  1 -1
0.0088811  1  1  0
```

```
 0.0000000  1  1  1

 Probability select group 3 is 0.2
 Group 3 density function is:
 prob      support in increments
 0.0912128 -2
 0.4087872 -1
 0.0000000  0
 0.4087872  1
 0.0912128  2


 Cache hit rate = 0.277744
 Cache hit rate = 0.321636
 Cache hit rate = 0.321402
 Cache hit rate = 0.334883
 Cache hit rate = 0.364927
 Cache hit rate = 0.405152
```

As seen from the cache hit rate, we reduced our run time by about 35 per cent.

The chain is shown in Figure 8. From the pairs shown in Figure 9 one can see the grid. The aggregate rejection rates from files `reject.000.dat` through `reject.005.dat` are

```
                Col   1      Col   2      Col   3      Col   4

 Row  1       0.32152      0.20320      1960.00      6096.00
 Row  2       0.25463      0.20173      1541.00      6052.00
 Row  3       0.52102      0.33940      5305.00      10182.0
 Row  4       0.59444      0.32507      5797.00      9752.00
 Row  5       0.60024      0.32787      5904.00      9836.00
 Row  6       0.19142      0.19433      1116.00      5830.00
 Row  7       0.37250      1.00000      11175.0      30000.0
```

If we were using the EMM program with a prior to do Bayesian inference, then these rejection rates are too low. For Bayesian inference it is better if all of them were above 50% so that the posterior gets adequately explored. For maximum likelihood inference one might prefer smaller rejection rates to try to keep in the region where a quadratic approximation to the likelihood is reasonably accurate. More on this later.

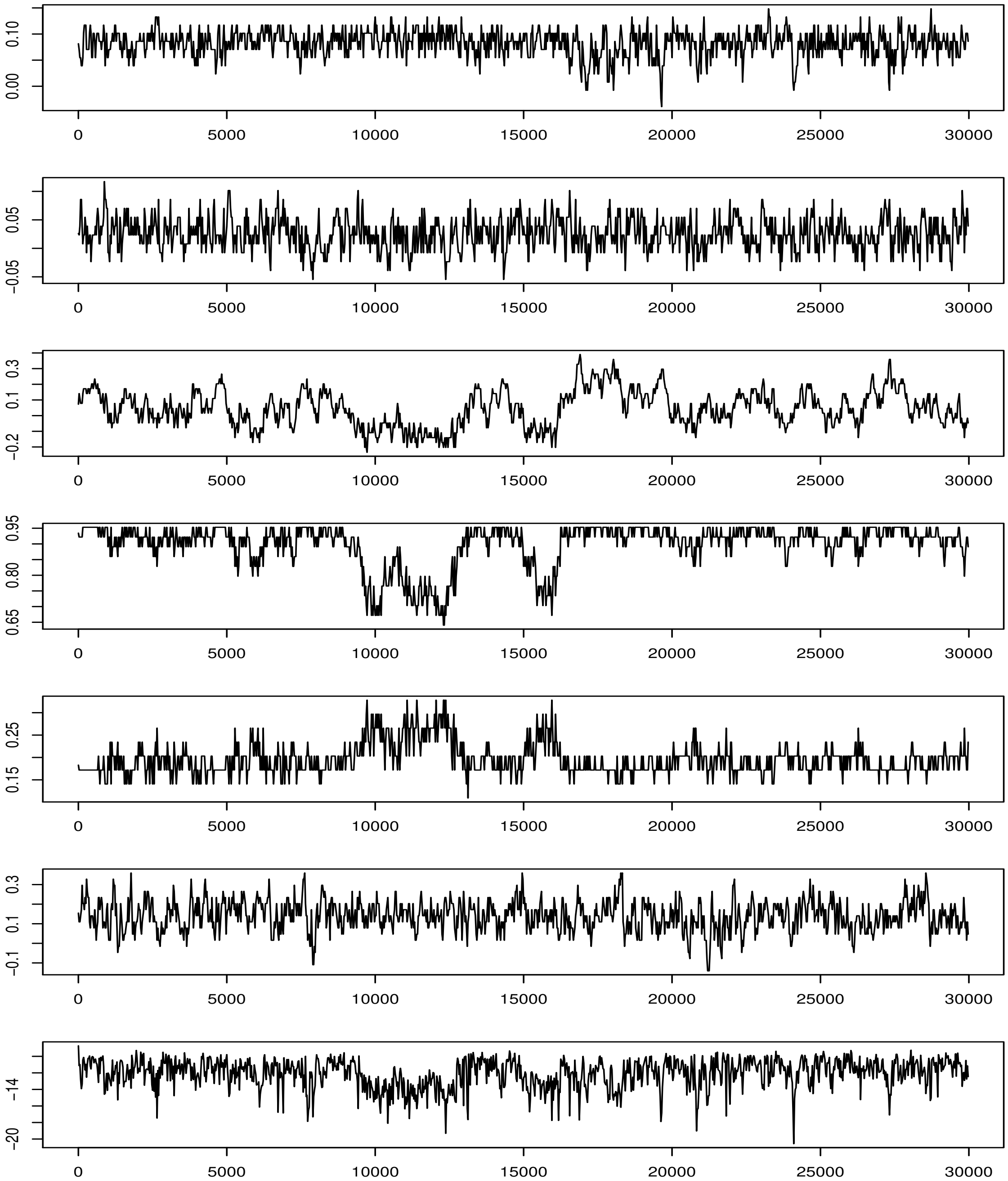

Figure 8. MCMC Chain from Parameter File sv.parm.005. The panels are from top to bottom a_0, a_1, b_0, b_1, s, r, and π. Every twenty-fifth point is plotted. $R = 30000$.

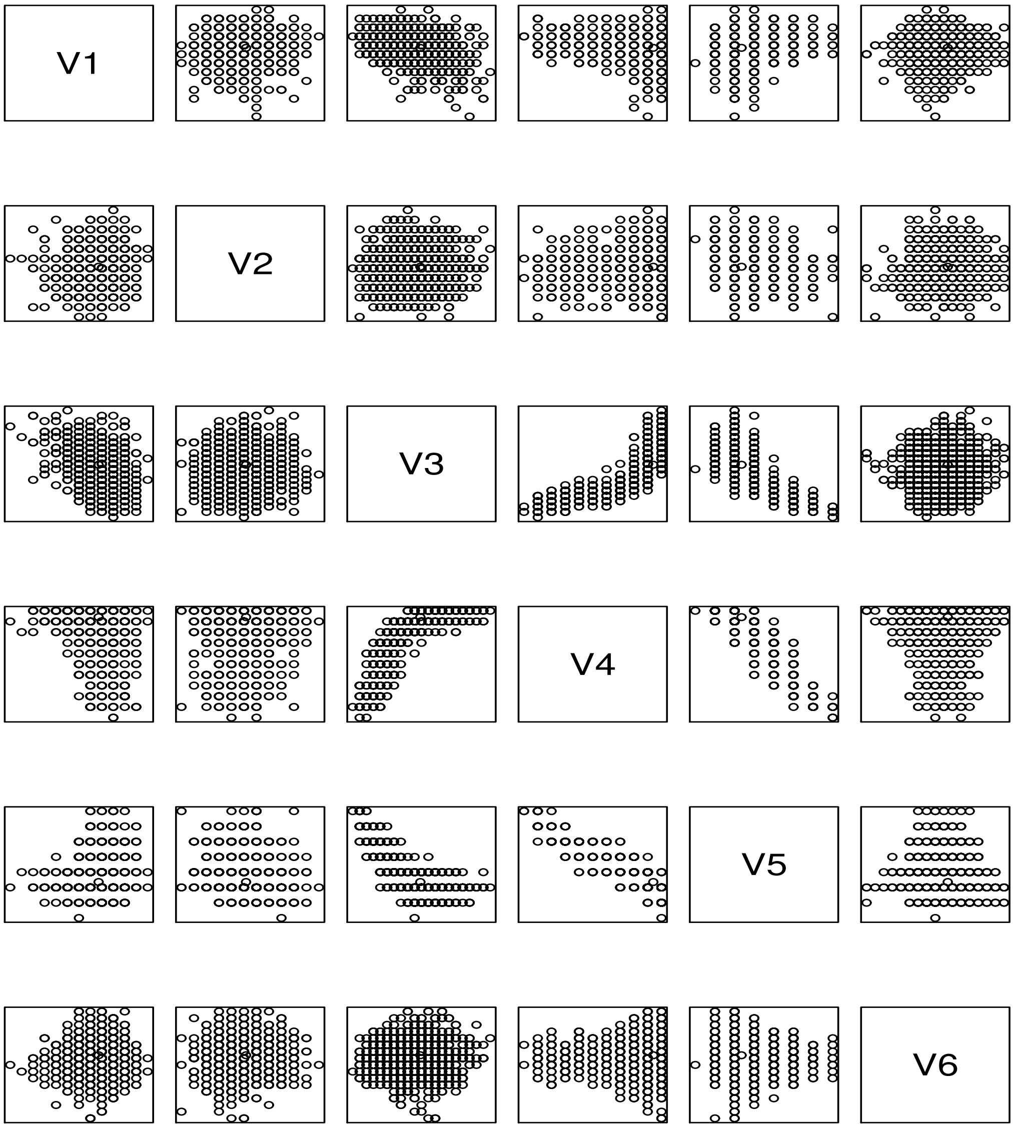

Figure 9. Scatter Plots of Chain from Parameter File sv.parm.005. The variables V1 through V6 are a_0, a_1, b_0, b_1, s, and r respectively. Every twenty-fifth point is plotted. $R = 30000$.

6.5 The Effect of Temperature

To see the effect of temperature, we shall rerun the chain for sv.parm.005 with a temperature of 2.0, calling it sv.parm.006. To make the consequences obvious in Figure 10, the first part of the chain is from the temperature 1.0 chain and the second half is from the temperature 2.0 chain. The aggregate rejection rate increases because the effect of doubling temperature is the same as doubling the sample size which makes the parameter scaling effectively twice as large.

	Col 1	Col 2	Col 3	Col 4
Row 1	0.42962	0.20343	2622.00	6103.00
Row 2	0.35885	0.20203	2175.00	6061.00
Row 3	0.70446	0.33960	7177.00	10188.0
Row 4	0.80326	0.32530	7839.00	9759.00
Row 5	0.80453	0.32810	7919.00	9843.00
Row 6	0.25317	0.19460	1478.00	5838.00
Row 7	0.50140	1.00000	15042.0	30000.0

The standard stochastic volatility model that we are using here does not fit data from financial markets well (Gallant, Hsieh, and Tauchen, 1997). One can do much better by including two stochastic volatility factors, one of which is mean reverting to fatten tails and the other of which is persistent to capture volatility clustering (Chernov, Gallant, Ghysels, and Tauchen, 2003). A hint of this can be seen from the chain for b_1 in the fourth panel of Figure 8 where the chain makes occasional excursions into a less persistent regime. As seen from the fourth panel of Figure 8, an effect of increasing temperature is to damp the movements of the chain. Although there are no excursions to a less persistent region in the high temperature half of Figure 8, they do occur if one runs the chain long enough. Their durations are shorter, however.

From the point of view of getting the MCMC chain to accurately compute the same asymptotic distribution as would be gotten from computations using analytic derivatives, keeping the movements of the chain close to the mle is beneficial because it confines the chain to the region where the likelihood is more accurately approximated by a quadratic. On the other hand, by not letting the chain make large excusions, we can mislead ourselves because we may miss noticing that the likelihood is really bimodal.

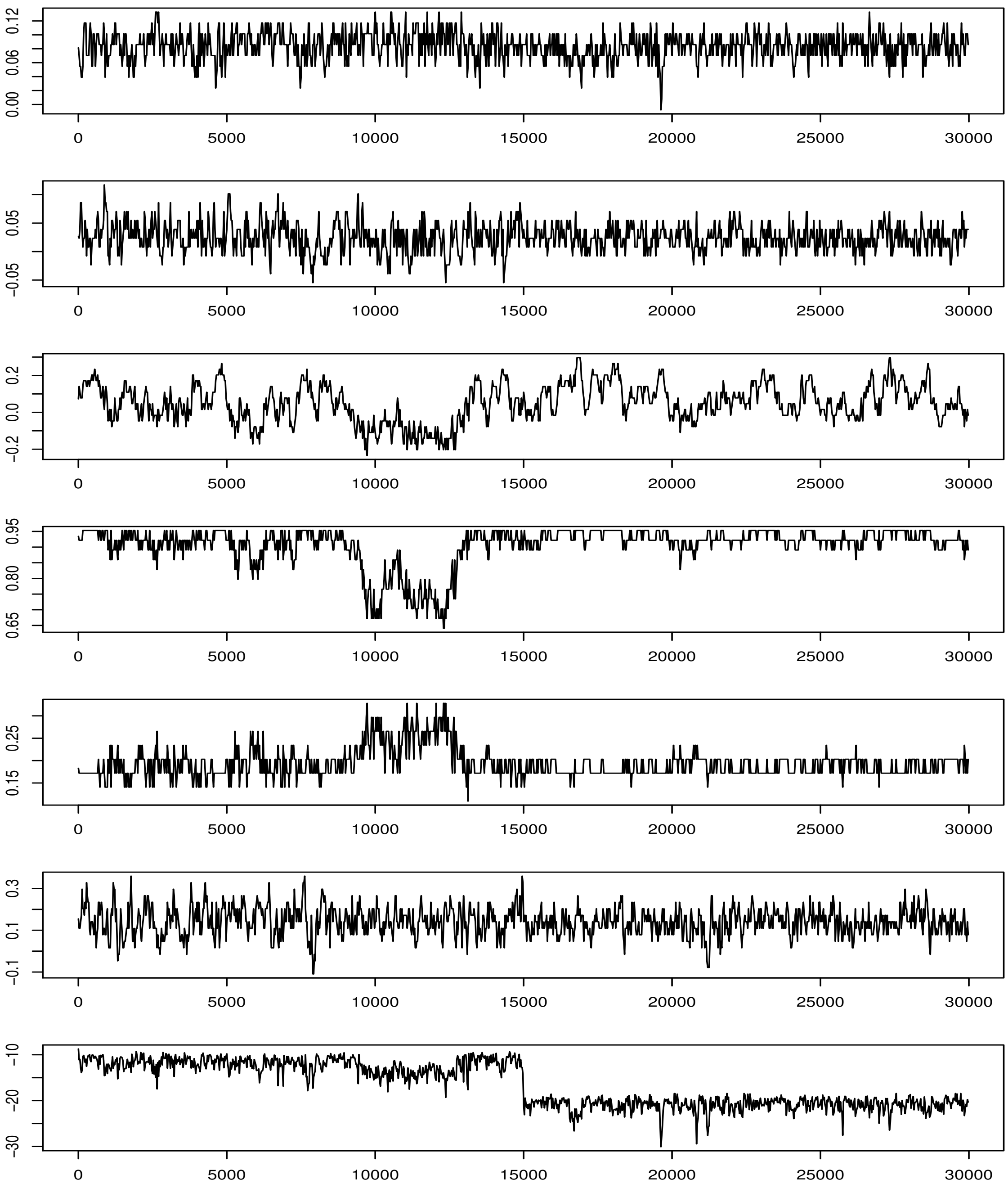

Figure 10. MCMC Chains from Parameter File sv.parm.005 and sv.parm.006. The last half of the chain is from sv.parm.006 which is the same as sv.parm.005 except that temperature is 2.0 instead of 1.0. The first half of the chain is from sv.parm.005. The panels are from top to bottom a_0, a_1, b_0, b_1, s, r, and π. Every twenty-fifth point is plotted. $R = 30000$.

6.6 Interpreting the Output

We shall continue by using `sv.parmfile.fit` that resulted from `sv.parm.005` by copying it to `sv.parm.007`. We increase the temperature to 2.5 to reflect the considerations discussed in Subsection 6.5. We remove the group move proposal because it is not needed. We adjust parameter scaling to compensate for the change in temperature and set the parameter increments to the same values so that the proposal puts probability on two moves up and two down. To get a chain that is long enough and uncorrelated enough to compute accurate averages, in particular the estimate of the Hessian $\mathcal{J}$, we increase the number of MCMC simulations per file, `lchain`, to 20000, increase `nfile` to 9, and set `stride` to 10. To improve the accuracy with which the EMM objective function is computed, we increase the simulation size `slen` to 10000. For this run, the aggregate rejection rates are

```
               Col  1     Col  2     Col  3     Col  4

Row  1        0.24488    0.16767    8212.00    33535.0
Row  2        0.20729    0.16728    6935.00    33456.0
Row  3        0.11208    0.16814    3769.00    33627.0
Row  4        0.16870    0.16624    5609.00    33248.0
Row  5        0.11132    0.16574    3690.00    33148.0
Row  6        0.13895    0.16495    4584.00    32991.0
Row  7        0.16400    1.00000    32799.0     200000
```

The MCMC chain is shown in Figure 11.

The primary output file is `sv.summary.dat` which looks like this:

```
  rhomean     rhomode      sesand      sehess      seinfo
 0.067607    0.066406    0.050294    0.024447    0.016227
 0.036087    0.035156     0.04446    0.028485    0.019562
  0.09738    0.097656     0.33038    0.097826    0.030964
  0.93709     0.94043    0.076428    0.023172   0.0073957
  0.18006     0.18066    0.057519    0.018716   0.0067218
  0.12418     0.11719     0.18173    0.085317    0.041642

The log posterior (log prior - objfun) at the mode is     -8.6733.

For EMM, if usrmod.prior returns only 0 or 1, objfun is a chi-square
on ltheta - 1 - lrho degrees of freedom.  See SNP parmfile for ltheta.

The degrees of freedom for seinfo are 536.
```

The file shows the chi-square statistic which will be correct if the prior is a zero-one indicator function in whose support is an open set containing the true value. This is the case in our application because member `prior` of class `sv_usrmod` always returns `den_val.positive=true` and member `support` only imposes stability conditions and an identification rule. Printed is the negative of the normalized value of the optimized objective function. For EMM (`objtype`=0) the objective function is

$$n m_n'(\rho, \tilde{\theta}_n)(\tilde{\mathcal{I}}_n)^{-1} m_n(\rho, \tilde{\theta}_n),$$

where

$$m_n(\rho, \tilde{\theta}_n) = \frac{1}{N} \sum_{t=1}^{N} (\partial/\partial\theta) \ln f \Big[\hat{y}_t(\rho) \mid \hat{x}_{t-1}(\rho), \tilde{\theta}_n \Big],$$

$$\tilde{\mathcal{I}}_n = \frac{1}{n} \sum_{t=1}^{n} \Big[(\partial/\partial\theta) \ln f_t(\tilde{y}_t \mid \tilde{x}_{t-1}, \tilde{\theta}_n) \Big] \Big[(\partial/\partial\theta) \ln f_t(\tilde{y}_t \mid \tilde{x}_{t-1}, \tilde{\theta}_n) \Big]'.$$

and $\tilde{\theta}_n$ maximizes the likelihood of the auxiliary model $f(y|x, \theta)$ as discussed in Subsection 1.2.

Under correct specification of the structural model, the normalized value of the optimized EMM objective function is asymptotically $\mathcal{X}^2$ with degrees of freedom equal to the length of θ minus the length of ρ minus one to account for the SNP normalization rule that `A(1,1)=1`. In our instance, `ltheta` is ten and `lrho` is six leaving three degrees of freedom. The p-value is about 0.02.

The file also shows the mode, which is the suggested estimate for EMM, the mean, and three sets of standard errors: sandwich $V = \mathcal{J}^{-1}\mathcal{I}\mathcal{J}^{-1}$, information matrix $\mathcal{I}^{-1}$, and Hessian $\mathcal{J}^{-1}$. If one is confident that the SNP fit is a good approximation to the true data generating process, then the standard errors from the Hessian should be used. Otherwise the sandwich standard errors should be used. However, because the sandwich estimate involves numerical differentiation and, in the case of EMM, several nonlinear optimizations, accuracy is a concern.

Sandwich standard errors are sensitive to the MCMC chain tuning parameters. Experience to date suggests that when the information and Hessian standard errors are roughly the same order of magnitude then the standard errors are reliable.

Another indicator of reliability are kernel density plots of the marginals of the MCMC chain for ρ as shown in Figure 12. These plots should look like the normal density. In our instance there is a small departure in panel four, which is the panel for b_1, caused by the excursions to a less persistent regime seen in panel four of Figure 11. This phenomenon is discussed in Subsection 6.5. Increasing the temperature further with corresponding decrease in proposal scale might correct this. What one is trying to do is sample where the peak of the objective function is well approximated by a quadratic without being driven to a scaling that renders the computations unstable. The computation of $\mathcal{I}$ is especially finicky because it involves numerous nonlinear optimizations to recompute SNP's θ.

Also produced are files that allow access to the components of the computations reported in `sv.summary.dat` to full machine precision. Their contents are obvious from the labels.

```
sv.V_hat_hess.dat      sv.rho_mean.dat
sv.V_hat_info.dat      sv.rho_mode.dat
sv.V_hat_sand.dat
```

Although in theory either the mean or the mode can be used as an estimate of rho, in most instances the mode is preferable. This is because the mode is what optimizes the EMM objective function and the parameters of the mode will generate the simulation that produced the mode. The mean may not even be suitable for generating a simulation because the parameter values may not be in the model's support. It is also of interest to examine kernel density plots of `stats`, which are the statistics $s(\rho)$ computed from the simulations $\rho \mapsto \{\hat{y}_\tau(\rho)\}_{\tau=1}^{N} \mapsto s(\rho)$ for each ρ in the MCMC chain. They are shown in Figure 13.

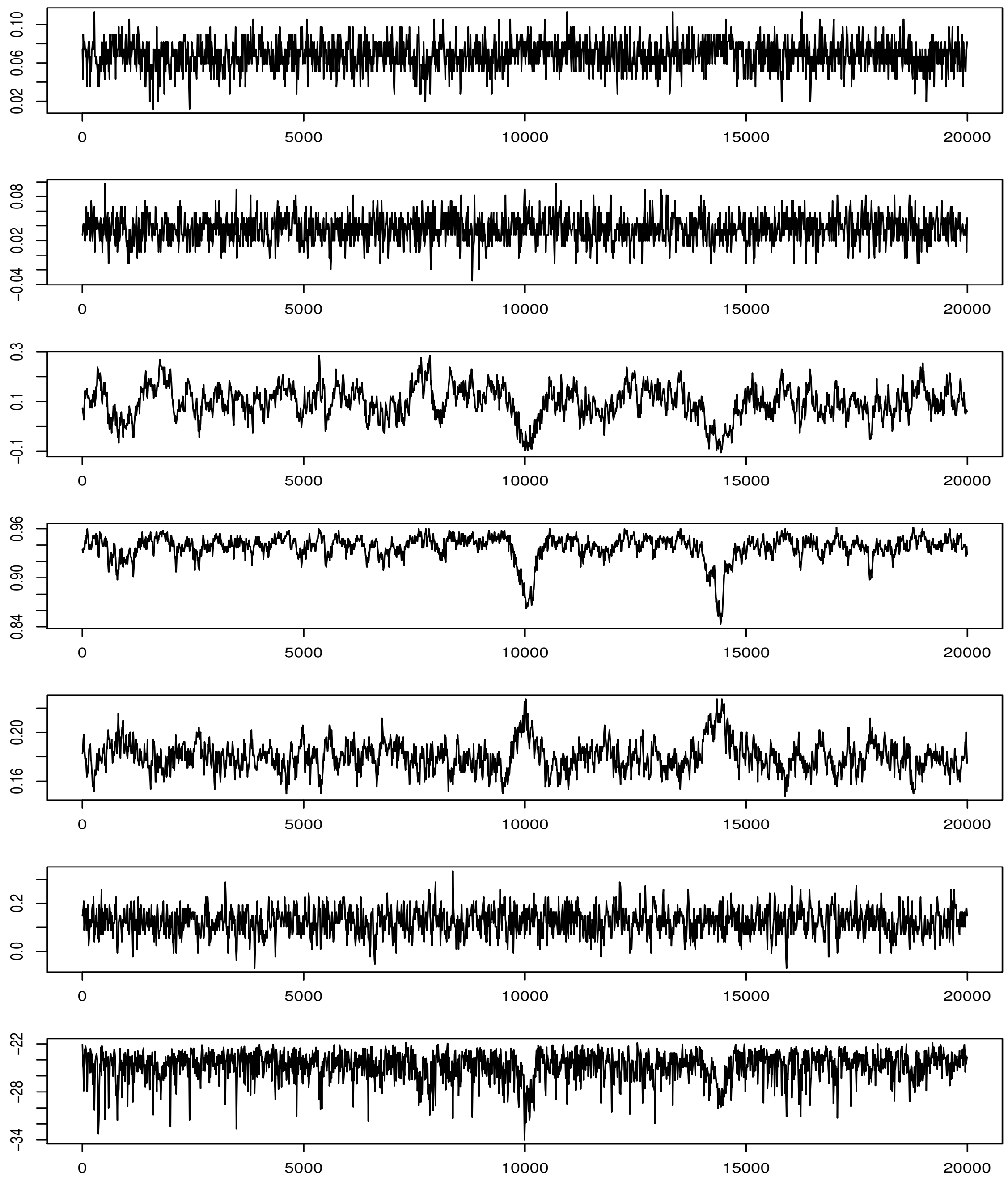

Figure 11. MCMC Chain from Parameter File sv.parm.007. The panels are from top to bottom a_0, a_1, b_0, b_1, s, r, and π. Every hundredth point is plotted. $R = 200000$.

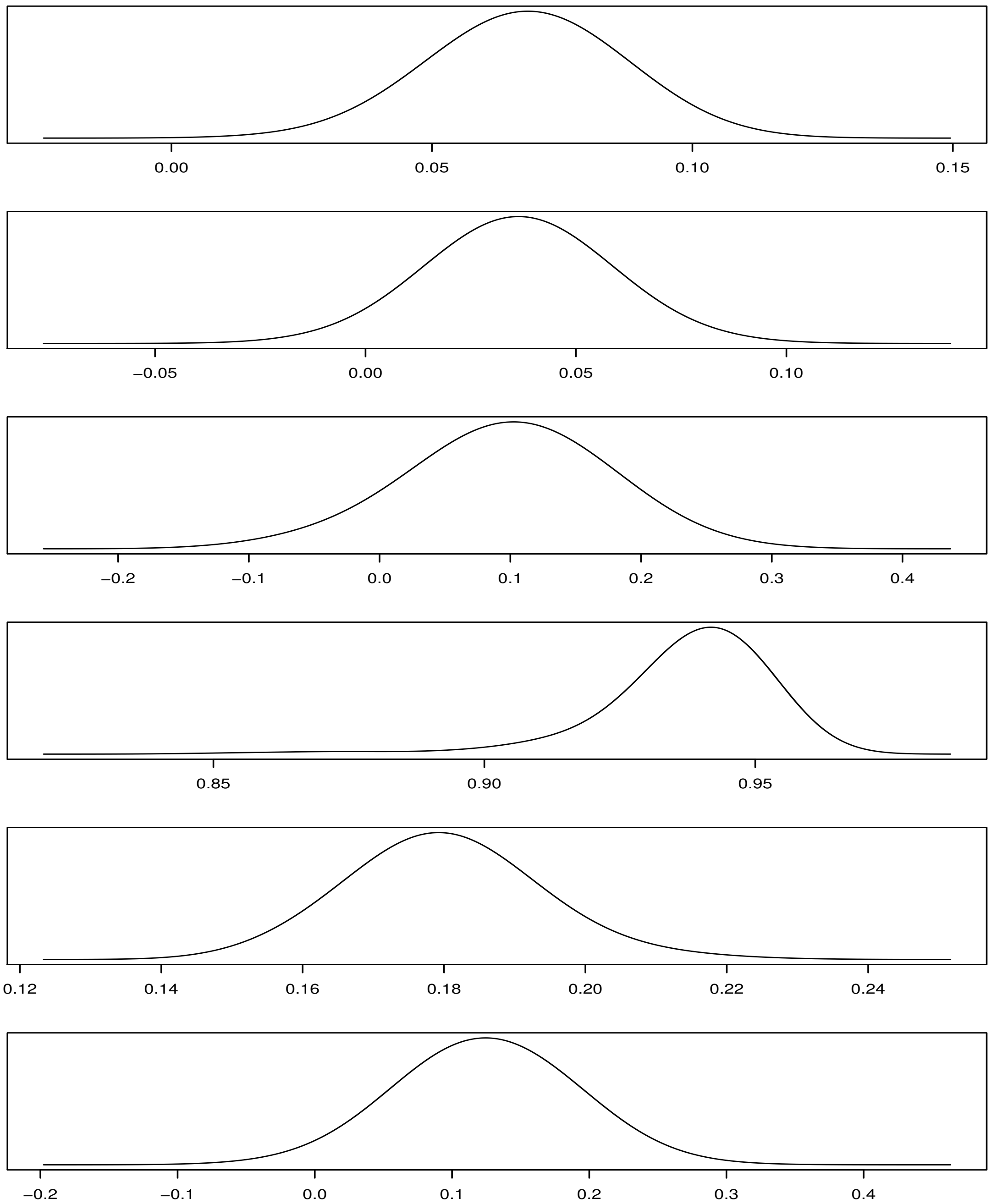

Figure 12. Kernel Density Estimates from Chain of Parameter File sv.parm.007. The variables from top to bottom are a_0, a_1, b_0, b_1, s, and r. Kernel is computed from every hundredth point. $R = 200000$.

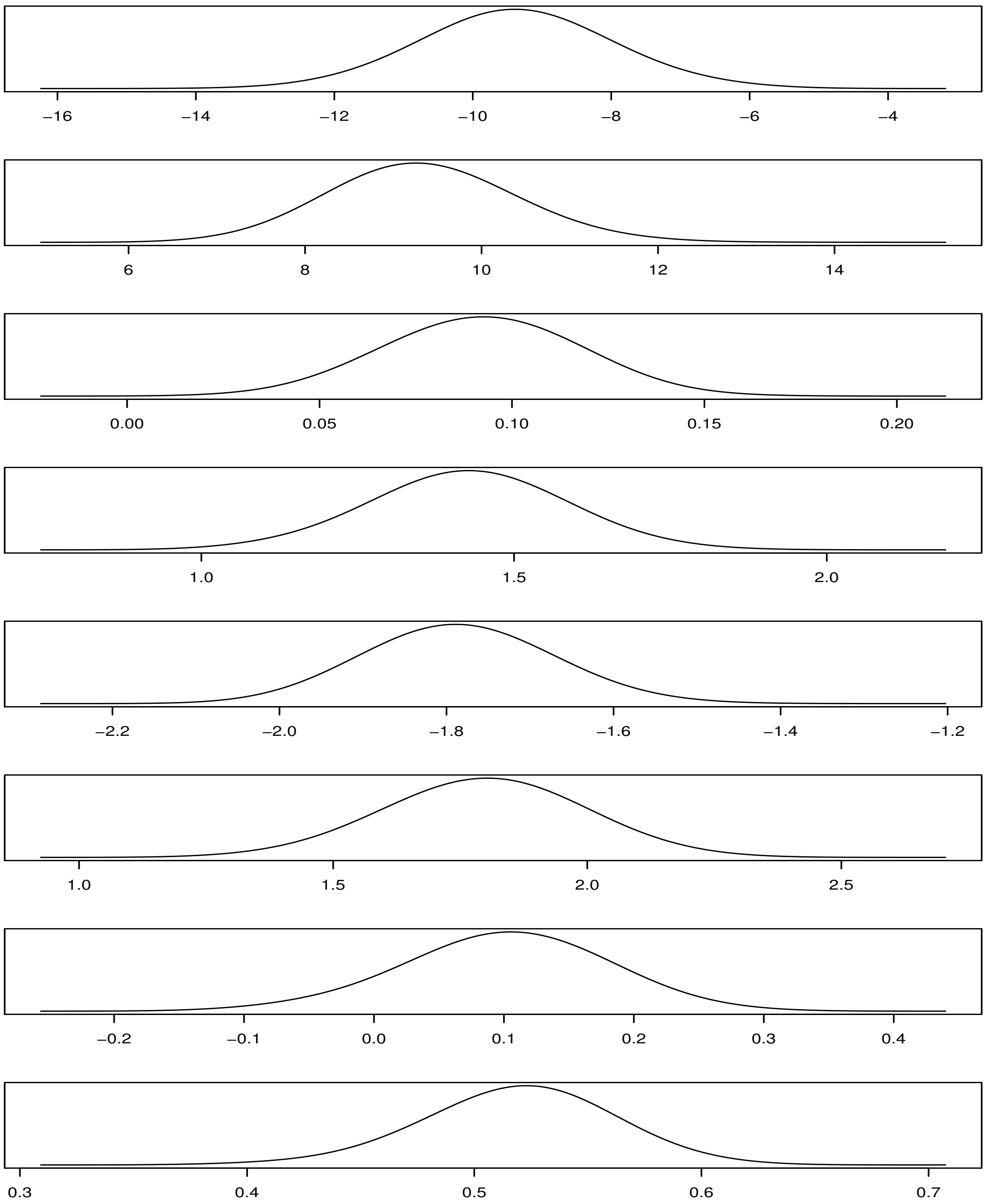

Figure 13. Kernel Density Estimates of Stats from Parameter File sv.parm.007. The variables from top to bottom are the minimum, maximum, mean, and standard deviation of y_t followed by the same for v_t. Kernel is computed from every hundredth point. $R = 200000$.

6.7 Score Diagnostics

The file `sv.diagnostics.dat` contains the normalized mean SNP score vector under the model, which is

$$\sqrt{n}\, m_n'(\tilde{\rho}_n, \tilde{\theta}_n) = \frac{1}{N} \sum_{\tau=1}^{N} \sqrt{n}(\partial/\partial\theta) \ln f\Big(\hat{y}_\tau(\tilde{\rho}_n) \,|\, \hat{x}_{\tau-1}(\tilde{\rho}_n),\, \tilde{\theta}_n\Big)\,,$$

along with the unadjusted standard errors of the normalized scores, and the corresponding quasi-t-statistics. The unadjusted standard deviations are the square roots of the diagonal elements of $\tilde{\mathcal{I}}_n$, and the quasi-t-ratios are the ratios of normalized scores to the unadjusted standard deviations. Here is an example of file `sv.diagnostics.dat` from `sv.parm.007`:

```
Score diagnostics:
           normalized         standard
Index      mean score            error      t-statistic     descriptor
    1         1.36398          1.96687          0.69348     a0[1]    1
    2        -1.93021          1.96554         -0.98203     a0[2]    2
    3        -2.14192          1.93646         -1.10610     a0[3]    3
    4        -2.41360          1.97982         -1.21910     a0[4]    4
    5         0.00000          0.00000          0.00000     A(1,1)   0 0
    6         0.09779          1.33933          0.07301     b0[1]
    7        -0.25862          1.02792         -0.25160     B(1,1)
    8        -0.90908          2.17472         -0.41802     R0[1]
    9         0.18832          2.60930          0.07217     P(1,1)   s
   10         2.63979          6.82532          0.38676     Q(1,1)   s
```

The quasi-t-ratios are not actually asymptotically $N(0,1)$ because they take account only of the randomness in $\tilde{\theta}_n$, while treating $\hat{\rho}_n$ as if it were the fixed value ρ_0. Interestingly, the unadjusted standard errors are biased upwards (Newey, 1985) and (Tauchen, 1985) so the quasi-t-ratios are downward biased relative to 2.0. The quasi-t-ratios are still useful for assessing how well the model fits along various dimensions. A quasi-t-statistic above 2.0 still indicates failure to fit the corresponding score. The quasi-t-ratios are particularly useful for assessing the underlying causes of a statistically significant chi-square statistic.

6.8 More is Better?

We copied `sv.parm.007` to `sv.parm.008`, changed `slen` ($= N$) to from 10000 to 20000 and left all else the same. Here is the result:

```
  rhomean     rhomode      sesand      sehess      seinfo
 0.074536    0.074219    0.040543    0.023101    0.016755
 0.039306    0.035156      0.0455    0.028555    0.019418
 0.091753    0.089844     0.33007    0.095939    0.029408
  0.93609     0.93848     0.08281    0.021899   0.0061191
```

```
     0.17976      0.18066    0.064532    0.018885   0.0061137
     0.11755      0.10156     0.17811    0.084962    0.042845

The log posterior (log prior - objfun) at the mode is     -8.7582.

For EMM, if usrmod.prior returns only 0 or 1, objfun is a chi-square
on ltheta - 1 - lrho degrees of freedom.  See SNP parmfile for ltheta.

The degrees freedom for seinfo are 526.
```

There is not much change. The MCMC chain looks the same as Figure 11. It would seem that with MCMC methods `slen` can be smaller than with methods that rely on nonlinear optimization. Experience acquired with earlier versions of EMM that relied on nonlinear optimizers rather than MCMC methods suggests that 50000 is pushing the limits of stability and 100000 is more reasonable.

6.9 Criterion Difference

Confidence intervals can also be obtained by inverting the criterion difference test based on the asymptotic chi-square distribution of the optimized objective function (Gallant and Tauchen, 1996b, 1997). By way of explanation, let

$$q_i(\rho_i) = n \max_{\rho,\, \rho_i \text{fixed}} \; m_n(\rho, \tilde{\theta}_n)(\tilde{\mathcal{I}}_n)^{-1} m_n(\rho, \tilde{\theta}_n),$$

denote the profile objective function with every parameter other than ρ_i optimized out. The set $\{\rho_i : q_i(\rho_i) - q_i(\hat{\rho}_i) \leq \mathcal{X}^2_{1-\alpha}\}$ is a level-α confidence interval for parameter ρ_i obtained by inverting the criterion difference test. (Technically, this set is a confidence region, because it could be disconnected, though we shall continue to use the more intuitive term of confidence interval.) The interval is the set of all values of ρ_i for which the difference $\hat{\rho}_i - \rho_i$ is statistically insignificant under a one-degree of freedom chi-square test. Simply put the interval is the set of acceptable null hypotheses for ρ_i. This confidence interval has better properties than conventional Wald-type confidence intervals computed from the standard errors in `sv.summary.dat`. It reflects asymmetries in the objective function and is invariant under nonlinear reparametrization.

The easiest way to find the boundary

$$q_i(\rho_i) - q_i(\hat{\rho}_i) = \mathcal{X}^2_{1-\alpha}$$

is to compute $q_i(\rho_i)$ for values near where the standard error for ρ_i in `sv.summary.dat` suggests the solution ought to be and interpolate. We will illustrate for $i = 4$, which is the autoregressive coefficient b_1 of the volatility equation. The values we shall use are $\rho_4 =$ 0.60, 0.65, 0.71, 0.96, 0.97, 0.98. The standard error provided good guidance for the three points on the right. The three on the left had to be found by trial and error. One needs to bracket the interval by getting points to the left and right of $\hat{\rho}_i$ that satisfy $q_i(\rho_i) - q_i(\hat{\rho}_i) > \mathcal{X}^2_{1-\alpha}$. In this case the profile objective function $q_4(\rho_4)$ is not well approximated by a quadratic so the guidance provided by the standard error was not very helpful at the left end. The reasons for this are the model's deficiencies that were discussed in Subsection 6.5. A tedious succession of fits on the left was required to find the point $\rho_4 = 0.60$.

The trial points are coded in parmfiles which we have labeled `sv60.parmfile.in0` through `sv98.parmfile.in0`. The control file looks like this for the six points displayed in Table 1; the middle point is from the `sv.summary.dat` file in Subsection 6.6.

```
sv60.parmfile.in0  sv60
sv65.parmfile.in0  sv65
sv71.parmfile.in0  sv71
sv96.parmfile.in0  sv96
sv97.parmfile.in0  sv97
sv98.parmfile.in0  sv98
```

Here is what sv97.parmfile.in0 looks like.

```
PARMFILE HISTORY (optional)
#
# This parmfile was written by EMM Version 2.5 using the following line from
# control.dat, which was read as char*, char*
# ---------------------------------------------------------------------------
#     sv.parm.007                   sv
# ---------------------------------------------------------------------------
#
ESTIMATION DESCRIPTION (required)
    SpotRate   Project name, pname, char*
         2.6   EMM version, defines format of this file, emmver, float
           0   Objfun type, 0 EMM, 1 MLE, 2 usr, objtype, int
           0   Proposal type, 0 group_move, 1 cond_move, 2 usr, proptype, int
           1   Write detailed output if print=1, int
         457   Seed for MCMC simulations, iseed, int
       20000   Number of MCMC simulations per file, lchain, int
           9   Number of MCMC simulation files beyond the first, nfile, int
         1.0   Rescale proposal scaling by this value, sclfac, float
         1.0   Rescale parameter increments by this value, incfac, float
         2.5   Rescale objfun by this value, temperature, float
           1   Sandwich variance not computed if kilse=1, int
          10   The stride used to write MCMC simulations, stride, int
           0   Draw from prior if draw_from_prior=1, int
       75000   Max cache size, max_cache_size, int
DATA DESCRIPTION (required) (mod and obj constructors see realmat data(M,n))
           1   Dimension of the data, M, int
```

```
         834   Number of observations, n, int
dmark.dat      File name, any length, no embedded blanks, dsn, string
4              Read these white space separated fields, fields, intvec
MODEL DESCRIPTION (required)
           6   Number of modal parameters, len_mod_parm, int
           8   Number of model functionals, len_mod_func, int
MODEL PARMFILE (required) (constructor sees as vector<string> pfvec, alvec)
__none__       File name, code __none__ if none, mod_parmfile, string
#begin additional lines
       10000   Number of observations in simulated data, slen (=N), int
         500   Initial simulations to eliminate transients, spin, int
#end additional lines
OBJFUN PARMFILE (required) (constructor sees as vector<string> pfvec, alvec)
11114000.fit   File name, code __none__ if none, obj_parmfile, string
#begin additional lines
#end additional lines
PARAMETER START VALUES (required)
   6.64062500000000000e-02    1
   3.51562500000000000e-02    1
   9.76562500000000000e-02    1
   9.70000000000000000e-01    0
   1.80664062500000000e-01    1
   1.17187500000000000e-01    1
PROPOSAL SCALING (required)
   7.81250000000000000e-03
   7.81250000000000000e-03
   7.81250000000000000e-03
   1.95312500000000000e-03
   1.95312500000000000e-03
   1.56250000000000000e-02
PARAMETER INCREMENTS (optional) (fractional powers of two recommended)
   7.81250000000000000e-03
   7.81250000000000000e-03
   7.81250000000000000e-03
   1.95312500000000000e-03
   1.95312500000000000e-03
   1.56250000000000000e-02
```

The summary file that results from sv97.parmfile.in0 looks like this.

```
          rhomean     rhomode      sesand      sehess      seinfo
         0.059575    0.058594                 0.02885
         0.044627    0.042969                0.027549
          0.25409     0.25391                 0.06729
          0.96973     0.96973                       0
           0.1427     0.14355                0.012104
          0.15042     0.14844                0.094205

     The log posterior (log prior - objfun) at the mode is      -12.79.

     For EMM where objfun.prior returns only 0 or 1 this is a chi-square
     on ltheta - 1 - lrho degrees freedom.  See SNP parmfile for ltheta.
```

Notice that even though we coded $\rho_4 = 0.97$, the coded value got put on the grid so that the actual value to which $q_4(\rho_4)$ corresponds is $\rho_4 = 0.96973$; a value with more significant digits is in `sv97.rho_mode.dat`. Zero is not on the grid. The first positive point on the grid is half the increment and the first negative point is minus half the increment. There are no sandwich standard errors in the summary file because `kilse` = 1.

Table 1. Criterion Differences for ρ_4.

ρ_4	$q_4(\rho_4)$	$q_4(\rho_4) - q_4(\hat{\rho}_4)$
0.97949	18.629	9.9557
0.96973	12.790	4.1167
0.95996	9.9702	1.2969
0.93709	8.6733	0.0
0.71973	11.939	3.2657
0.64941	12.289	3.6157
0.60059	12.544	3.8707

Notes: The 95% critical point of a χ^2 on 1 df. is 3.841. $R = 5000$.

We fit a quadratic to the first three values in the first and third columns of Table 1 and solved for the point where the quadratic equals 3.841 to get the upper end of the the criterion difference confidence interval. Similarly for the lower using the last three values. The interval we obtained is

$$(0.606, 0.969)$$

The confidence interval computed from the sandwich standard error in file `sv.summary.dat` of Subsection 6.6 is

$$(0.774, 1.11).$$

This interval does not reflect the sharp rise of the profile objective function on the right nor its slow rise on the left.

As remarked above, the criterion difference confidence intervals reflect asymmetries in the objective function and are to be preferred. They are also safer from a numerical analysis point of view because they only require that the mode be accurately determined by the MCMC chain, which requires neither careful tuning to try and get $\mathcal{I}$ accurately determined nor excessive length to get $\mathcal{J}$ accurately computed.

6.10 Running on a Parallel Machine

The parallel version of EMM, which is `emm_mpi`, is similar to the serial version, which is `emm`, but with some quirks caused by restrictions imposed by the LAM implementation of MPI for which the code was written. These are that path names must be absolute, that command line parameters should not be used, and that subnodes cannot print anything.

The way the absolute path name requirement is handled is to supply a header `pathname.h` that contains the absolute path name and builds it into the code at compile time. This header is generated automatically by the makefile `makefile.mpi` included with the distribution. It assumes that the build occurs in the same directory in which data, parmfiles, etc. are found.

The command line requirement is met by always using the file `control.dat` rather than entering a file name on the command line. Also, for the parallel version, only the first line of `control.dat` is read and processed.

The no print requirement is handled by always coding `print` = 0 on the SNP parmfile. If this is not done, at best an unintelligible mess will be printed to standard output, at worst the program will crash.

For our example, here is `pathname.h` which was generated automatically by the makefile:

```
#define PATHNAME "/home/arg/r/emm_develop/test_mpi"
```

This is `control.dat`:

```
sv.parm.007  sv
```

Running on a parallel machine requires initiation of MPI prior to execution. This is handled by a shell script `emm_mpi.lam_7.0.sh` included with the distribution:

```
#! /bin/sh

# This shell script works for an 8 box cluster with 2 mono core CPUs
# per box running LAM Version 7.0.  The host node is named n0 and the
# subnodes are named n1, n2, n3, n4, n5, n6, n7.

echo n0  > lamhosts
echo n1 >> lamhosts
echo n2 >> lamhosts
echo n3 >> lamhosts
echo n4 >> lamhosts
echo n5 >> lamhosts
echo n6 >> lamhosts
echo n7 >> lamhosts

test -f emm_mpi.err  && mv -f emm_mpi.err  emm_mpi.err.bak
test -f emm_mpi.out  && mv -f emm_mpi.out  emm_mpi.out.bak

rm -f core core.*

lamboot -v lamhosts

RC=$?

case $RC in
```

```
    0) ;;
    1) exit 1;;
    esac

  make -f makefile.mpi.lam_7.0 >emm_mpi.out 2>&1 && \
    mpirun -v -O -D -s h N N \
    ${PWD}/emm_mpi >>emm_mpi.out 2>emm_mpi.err

  RC=$?

  case $RC in
    0) exit 0 ;;
    esac
  exit 1;
```

Also included with the distribution are shell scripts and makefiles for Version 7.1 of LAM and for Version 2.1 of OpenMPI.

The results of a run are a set of files similar to those for the serial version. Instead of files being named like `sv.rho.001.dat` they are named like `sv.rho.015.001`. This would be `ifile`=001 produced by processor 15. (There are 16 processors numbered 0 to 15 because each of the eight nodes has two processors.) Files from one processor can be meaningfully concatenated. Files from different processors start at the same value of ρ (which should be the mode of the posterior to prevent initial transients) but use a different seed. The files `sv.emmcache.new`, `sv.rho_mode.dat`, etc. are jointly produced by all processors.

Here are the aggregate rejection counts from running `sv.parm.007` on this parallel machine

```
                Col   1      Col   2      Col   3      Col   4

    Row  1     0.25140     0.16661       125661       499842
    Row  2     0.20994     0.16672       105002       500155
    Row  3     0.11462     0.16668      57317.0       500047
    Row  4     0.16104     0.16658      80478.0       499746
    Row  5     0.11315     0.16671      56591.0       500144
    Row  6     0.14017     0.16672      70106.0       500161
    Row  7     0.16504     1.00000       495135      3000000
```

and here is the file `sv.summary.dat`

```
   rhomean     rhomode      sesand       sehess       seinfo
  0.068852    0.066406    0.053952    0.024436     0.014774
  0.035004    0.035156    0.049452    0.028573      0.01835
  0.079053    0.097656     0.48568     0.10306     0.022898
   0.93329     0.94043     0.14514    0.027875    0.0057576
   0.18233     0.18066    0.079283    0.020861    0.0064226
   0.12357     0.11719     0.18682    0.083871     0.039562

  The log posterior (log prior - objfun) at the mode is     -8.6733.

  For EMM, if usrmod.prior returns only 0 or 1, objfun is a chi-square
```

```
on ltheta - 1 - lrho degrees of freedom.  See SNP parmfile for ltheta.
The degrees of freedom for seinfo are 113310.
```

The run produced three million MCMC trials with a stride of ten, which leaves us with 300,000 trials fragmented into $15 \times 10 = 150$ files. The program `combine` in subdirectory utility with usage

```
../utility/combine prefix processors nfiles stride
```

will combine the chains from each processor into one file using a stride to reduce length. In the command line, prefix refers to the prefix in `control.dat`, which is `sv` in our case, processors is as above, which is 15 in our case, `nfiles` is the value from the parmfile named in `control.dat`, which is `sv.parm.007` with `nfiles`=9 in our case, and `stride` is set by the user. Here are our choices

```
../utility/combine sv 15 9 10
```

which will reduce the MCMC chain to length 30,000 and put 15 chains into a subdirectory `combined_files`. The concatenated chain and its autocorrelations sampled at every tenth point are shown in Figures 14 and 15. The net sampling rate in these figures is every thousandth point of the original chain of length 3,000,000. Notice the excursions of b_1 into a less persistent region in the fourth panel of Figure 14.

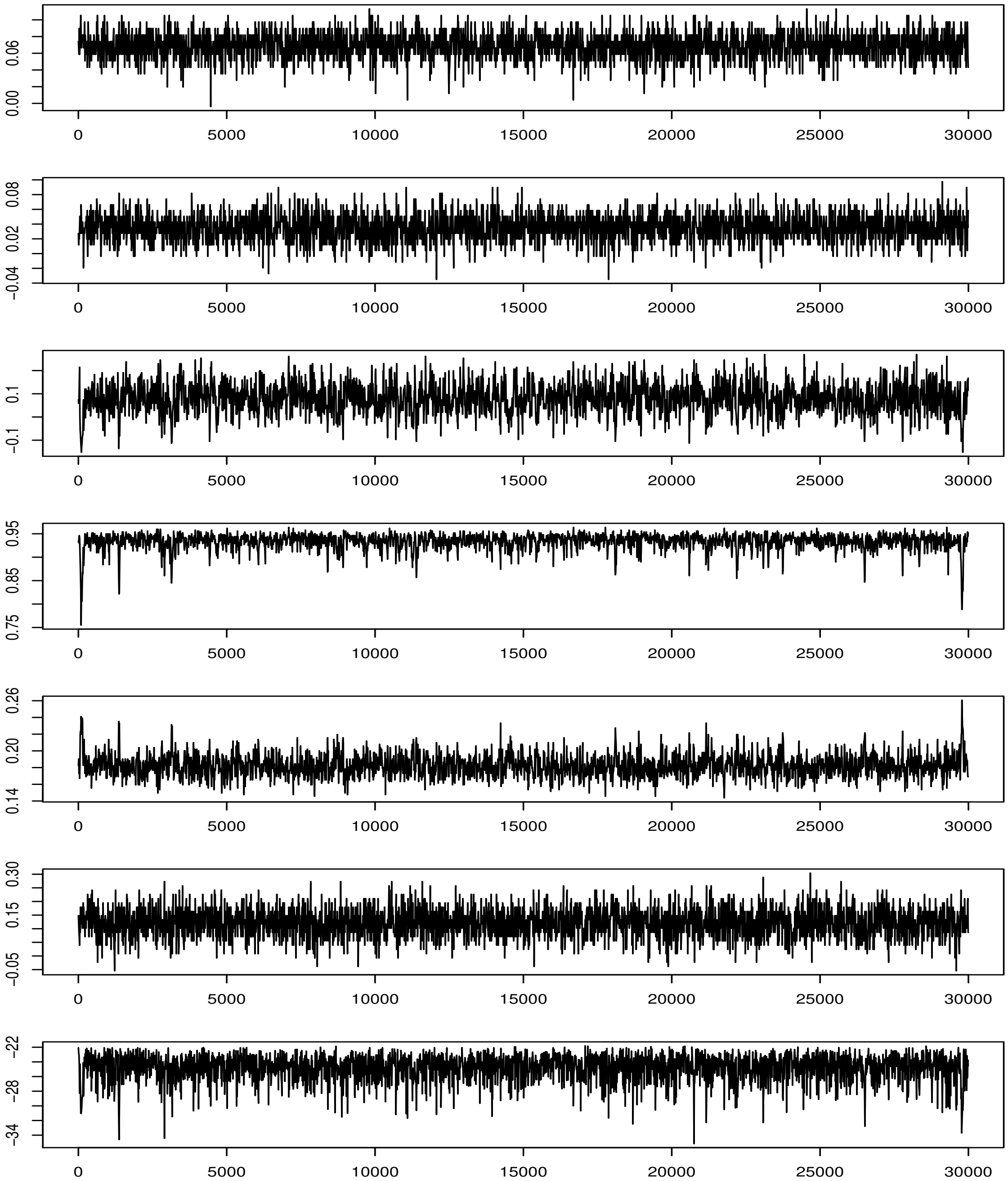

Figure 14. Parallel Machine MCMC Chain from Parameter File sv.parm.007. The panels are from top to bottom a_0, a_1, b_0, b_1, s, r, and π. Every thousandth point is plotted. $R = 3,000,000$.

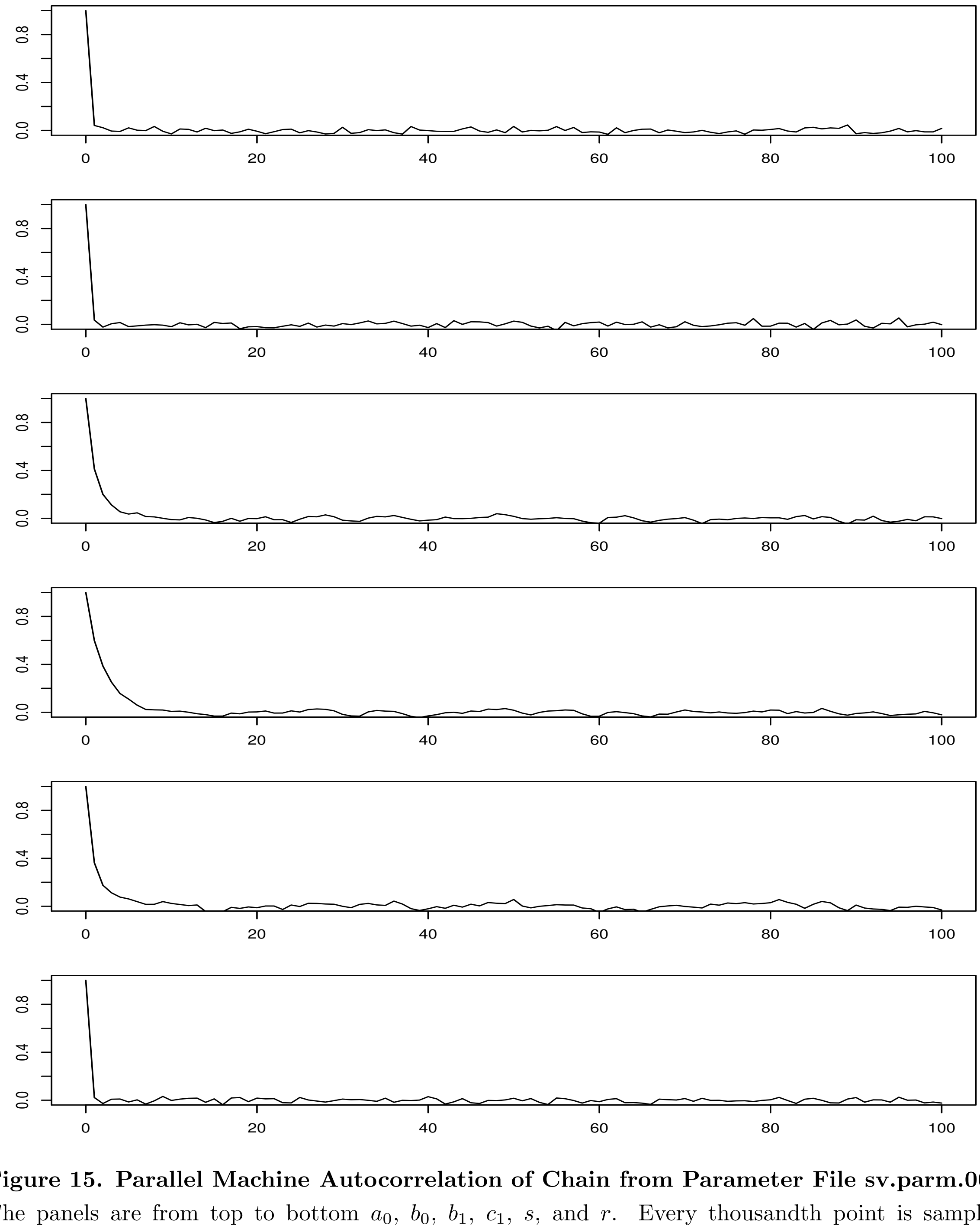

Figure 15. Parallel Machine Autocorrelation of Chain from Parameter File sv.parm.007. The panels are from top to bottom a_0, b_0, b_1, c_1, s, and r. Every thousandth point is sampled. $R = 3,000,000$.

7 Maximum Likelihood Estimation

Implementing a `usrmod` for maximum likelihood is much the same as implementing a `usrmod` for EMM. Recall, that the header `libsmm/src/libsmm_base.h` that defines the interface is

```
#include "libscl.h"

namespace libsmm {

  /* Now in libscl

  struct den_val {
    bool    positive;
    REAL    log_den;
            den_val() : positive(false), log_den(-REAL_MAX) { }
            den_val(bool p, REAL l) : positive(p), log_den(l) { }
  };

  */

  class usrmod_base {
  public:
    virtual INTEGER len_rho() = 0;
    virtual INTEGER len_stats() = 0;
    virtual bool gen_sim(scl::realmat& sim, scl::realmat& stats) = 0;
      //Same seed every call
    virtual void get_rho(scl::realmat& rho) = 0;
    virtual void set_rho(const scl::realmat& rho) = 0;
    virtual bool support(const scl::realmat& rho) = 0;
    virtual den_val prior(const scl::realmat& rho,
      const scl::realmat& stats) = 0;
    virtual void write_usrvar(const char* filename) { return; }
    virtual ~usrmod_base() {}
    virtual bool gen_bootstrap(std::vector<scl::realmat>& bs)
      //New seed each call
      {return false;}
    virtual void set_data(const scl::realmat& dat) {}
    virtual libsmm::den_val likelihood(scl::realmat& predicted,
      scl::realmat& residuals)
      {return den_val(false,-REAL_MAX);}
  };

}
```

The code for all member functions except `set_data` and `likelihood` has been discussed previously. Nothing changes here except that for maximum likelihood the argument `sim` of `gen_sim` is never used and so code to fill it does not need to be supplied. Also, for maximum likelihood, as with EMM, $\mathcal{I} = \mathcal{J}$ as discussed in Subsection 1.3. Therefore one can set `kilse`=1 in the `ESTIMATION DESCRIPTION` block of the parmfile in which case `gen_bootstrap` is never called and does not need to be coded. However, as with EMM one can code `gen_boostrap` and compute sandwich standard errors anyway. To illustrate, we shall code `gen_boostrap` and return bootstrap samples generated from model residuals.

Maximum likelihood uses the data and `rho` to compute the likelihood and therefore needs a copy of both `rho` and the data, usually stored as `realmat`s in the private part of class `usrmod`. The purpose of the member function `set_data(const realmat& dat)` is to replace `data` with `dat`. However, because `set_data` is only called if `kilse`=0, data must be set by the `usrmod` constructor as well. If this step is overlooked, the EMM program will crash.

The purpose of `likelihood` is to return the likelihood computed from `rho` and the data as a `den_val(p,l)`, where `p` is true if `rho` and the data satisfy the support conditions that pertain to the application and `l` is the log likelihood $\mathcal{L}(\rho)$.

The log likelihood has the property that $-2\mathcal{L}(\rho^o)$ is distributed as a chi square, where ρ^o denotes the true value of the parameter ρ. According to the theory discussed in Subsection 1.3, any statistical objective function with this property can be returned as `l`. In particular, if $m'(\rho)[W(\rho)]^{-1}m(\rho)$ is a GMM objective function based on moment equations $m(\rho) = \frac{1}{n}\sum_{t=1}^{n} m(y_t, \rho)$ and a weighting matrix $W(\rho)$ scaled so that $W(\rho)$ converges to a constant as the sample size n increases and such that the chi square property is satified, then $-\frac{1}{2}n\, m'(\rho)(W)^{-1}m(\rho)$ can be returned as `l`. For example, if $m(y_t, \rho^o)$ is iid, then

$$W(\rho) = \frac{1}{n}\sum_{t=1}^{n} \left[m(y_t, \rho) - m(\rho)\right]\left[m(y_t, \rho) - m(\rho)\right]',$$

is a suitably scaled weighting matrix for which the chi square property is satisfied. Note that we have subtracted off the mean in computing W. This is because the estimation method is MCMC and one does not want an absurd value of a proposed ρ to be accepted simply because it makes $W(\rho)$ ridiculously large. Also, as mentioned earlier and as discussed by Gallant and Hong (2007), it is perfectly acceptable to code a meaningful `prior` and to view the MCMC chain produced by the EMM program as a simulation whose stationary distribution is a Bayesian posterior.

We will illustrate with the electricity demand system described in detail in Chapter 5 of Gallant (1987). Briefly it is as follows. Given a vector of prices divided by expenditure x, an expenditure "share" is computed as

$$s = a + Bx$$

where a is a vector with last element -1 and B is a symmetric matrix. With these normalization conventions, an expenditure "share" actually has all elements negative and does not sum to one. When using conventional quasi Newton optimizers, imposing the constraint on a and B that this be so for all x_t observed in the data is actually quite difficult. However, as we shall see, it is trivial when using the EMM package. The "share" s is presumed to be the location parameter of a logistic normal distribution. If an observed share vector y and the "share" vector s both have dimension d and $y_{(1)}$ and $s_{(1)}$ represent a vector containing the first $d-1$ elements of these vectors, then the logistic normal density can be characterized by saying that

$$\log(y_{(1)}/y_d) \sim N_{d-1}\Big[\log(s_{(1)}/s_d), \Sigma\Big]$$

where $N_M(\mu, \Sigma)$ is the multivariate normal of dimension M and the log function is applied to a vector element by element. We shall parameterize Σ by means of its Cholesky factors: $\Sigma = RR'$, where R is upper triangular. In our example, $d = 3$ and $M = d - 1 = 2$.

Here is the emmusr.h that declares elec_usrmod:

```
#include "libsnp.h"
#include "libsmm.h"
#include "emm_base.h"
#include "snp.h"

namespace emm {

  class elec_usrmod;

  typedef elec_usrmod usrmod_type;

  class elec_usrmod : public libsmm::usrmod_base {
  private:
    scl::realmat data;
    scl::realmat rho;
    INTEGER blen;
    INTEGER lrho;
    INTEGER lstats;
    INT_32BIT variable_seed;
    scl::realmat a;
    scl::realmat B;
    scl::realmat R;
    void set_parms();
  public:
    elec_usrmod
      (const scl::realmat& dat, INTEGER len_mod_parm, INTEGER len_mod_func,
       const std::vector<std::string>& mod_pfvec,
       const std::vector<std::string>& mod_alvec,
       std::ostream& detail);
    INTEGER len_rho() {return lrho;}
    INTEGER len_stats() {return lstats;}
    bool gen_sim(scl::realmat& sim, scl::realmat& st)
      {st.resize(1,1,0.0); return true;}
```

```
    bool gen_bootstrap(std::vector<scl::realmat>& bs);
    void get_rho(scl::realmat& parm) { parm = rho; }
    void set_rho(const scl::realmat& parm) {rho = parm; set_parms();}
    void set_data(const scl::realmat& dat) { data = dat; }
    bool support(const scl::realmat& rho);
    libsmm::den_val prior(const scl::realmat& rho,
      const scl::realmat& stats);
    libsmm::den_val likelihood(scl::realmat& yhat, scl::realmat& zhat);
    void write_usrvar(const char* filename)
    {
      scl::realmat yhat, zhat;
      if (likelihood(yhat,zhat).positive) {
        vecwrite(filename, yhat);
      }
    }
  };
}
```

Here is the emmusr.cpp that defines elec_usrmod:

```
#include "libsmm.h"
#include "emm.h"

using namespace scl;
using namespace libsmm;
using namespace emm;
using namespace std;

emm::elec_usrmod::elec_usrmod
  (const realmat& dat, INTEGER len_mod_parm, INTEGER len_mod_func,
   const std::vector<std::string>& mod_pfvec,
   const std::vector<std::string>& mod_alvec,
   std::ostream& detail)
: data(dat), rho(), blen(23), lrho(11), lstats(1),
  variable_seed(740726), a(), B(), R()
{
  vector<string>::const_iterator usr_ptr = mod_alvec.begin();
  ++usr_ptr;
  blen = atoi((usr_ptr++)->substr(0,12).c_str());

  if (lrho != len_mod_parm) {
    error("Error, usrmod, constructor, len_mod_parm is set wrong in parmfile");
  }

  if (lstats != len_mod_func) {
    error("Error, usrmod, constructor, len_mod_func is set wrong in parmfile");
  }

  if (blen < 2*lrho+1) {
    blen = 2*lrho+1;
    warn("Warning: usrmod, constructor, blen increaed to 2*lrho+1");
  }

}

void emm::elec_usrmod::set_parms()
{
  a.resize(3,1);
  B.resize(3,3);
  R.resize(2,2);

  a[1] = rho[1];
```

```
  a[2] = rho[2];

  B(1,1) = rho[3];
  B(1,2) = rho[4];
  B(2,2) = rho[5];
  B(1,3) = rho[6];
  B(2,3) = rho[7];
  B(3,3) = rho[8];

  R(1,1) = rho[9];
  R(1,2) = rho[10];
  R(2,2) = rho[11];

  a[3] = -1.0;

  B(2,1) = B(1,2);
  B(3,1) = B(1,3);
  B(3,2) = B(2,3);

  R(2,1) = 0.0;
}

den_val emm::elec_usrmod::likelihood(realmat& yhat, realmat& zhat)
{
  if (!support(rho)) return den_val(false,-REAL_MAX);;

  INTEGER r = data.get_rows();
  INTEGER n = data.get_cols();

  if (r != 5) error("Error, elec_usrmod, likelihood, bad data");
  if (rho.get_rows() != 11) error("Error, elec_usrmod, likelihood, bad parm");

  realmat y = data("1:2","");
  realmat x = data("3:5","");

  realmat s = B*x;
  for (INTEGER t=1; t<=n; ++t) {
    s(1,t) += a[1];
    s(2,t) += a[2];
    s(3,t) += a[3];
  }

  for (INTEGER t=1; t<=n; ++t) {
    for (INTEGER i=1; i<=s.size(); ++i) {
      if (s[i] >= 0.0) return den_val(false,-REAL_MAX);
    }
  }

  yhat.resize(2,n);
  zhat.resize(2,n);

  realmat ehat(2,n);

  for (INTEGER t=1; t<=n; ++t) {
    REAL bot = log(-s(3,t));
    yhat(1,t) = log(-s(1,t)) - bot;
    yhat(2,t) = log(-s(2,t)) - bot;
    ehat(1,t) = y(1,t) - yhat(1,t);
    ehat(2,t) = y(2,t) - yhat(2,t);
  }

  realmat P = inv(R);
```

```
  zhat = P*ehat;

  REAL q = 0.0;

  for (INTEGER t=1; t<=n; ++t) {
    q += pow(zhat(1,t),2) + pow(zhat(2,t),2);
  }

  q *= (-0.5);

  REAL detR = R(1,1)*R(2,2);

  q -= REAL(n)*log(detR);

  const REAL pi = 3.14159265358979312e+00;

  q -= REAL(n)*log(sqrt(2.0*pi));

  return den_val(true,q);
}

bool emm::elec_usrmod::support(const realmat& parm)
{
  if (parm[9] <= 0.0) return false;
  if (parm[11] <= 0.0) return false;

  return true;
}

den_val emm::elec_usrmod::prior(const realmat& rho_in, const realmat& stats)
{
  return den_val(true, 0.0);
}

bool emm::elec_usrmod::gen_bootstrap(vector<realmat>& bs)
{
  if (!support(rho)) return false;

  realmat yhat, zhat;

  den_val dv = likelihood(yhat, zhat);

  if (!dv.positive) return false;

  INTEGER len = 2*lrho+1;
  INTEGER len_vec = bs.size();

  if (len_vec != len) {
     bs.resize(len);
  }

  INTEGER n = data.get_cols();
  realmat sim(5,n);
  realmat x, z, e, s;

  for (INTEGER i=0; i<len; ++i) {
    for (INTEGER t=1; t<=n; ++t) {
      x = data("3:5",t);
      INTEGER u = iran(variable_seed, n-1);
      ++u;
      z = zhat("",u);
```

```
        e = R*z;
        s = B*x;
        s[1] += a[1];
        s[2] += a[2];
        s[3] += a[3];
        for (INTEGER j=1; j<=s.size(); ++j) {
          if (s[j] >= 0.0) return false;
        }
        REAL bot = log(-s[3]);
        sim(1,t) = log(-s[1]) - bot + e[1];
        sim(2,t) = log(-s[2]) - bot + e[2];
        sim(3,t) = x[1];
        sim(4,t) = x[2];
        sim(5,t) = x[3];
      }
      bs[i] = sim;
    }

    return true;
  }
```

All is as described earlier but attention needs to be called to the constructor and member functions (methods) `set_parms`, `gen_sim`, `gen_bootstrap`, and `likelihood`.

Note that the constructor for `elec_usrmod` sets the data. As mentioned earlier, this is essential.

The member function `set_parms` maps the parameter ρ into the vector a and matrices B and R.

As coded here, the method `gen_bootstrap` uses draws from the empirical distribution of standardized model residuals to generate bootstrap samples.

The method `gen_sim` of `elec_usrmod` doesn't do anything but set `stats` to the same dimension as coded in the parameter file and fill it with zeros.

The method `likelihood` of `elec_usrmod` computes

$$\log n_{d-1}\Big(\log(y_{(1)}/y_d)\,\Big|\,\log(s_{(1)}/s_d),\,\Sigma\Big)$$

for each datum in `data` and accumulates them in `q`. Note in particular the statement `if (s[i] >= 0.0) return den_val(false,-REAL_MAX);` that imposes the constraint that s have all entries negative for all x_t.

This code is not sensitive to tuning parameters and runs fast. Start values were taken from Chapter 5 of Gallant (1987). Herewith follows the parameter file.

```
PARMFILE HISTORY (optional)
```

```
#
# This parmfile was written by EMM Version 2.5 using the following line from
# control.dat, which was read as char*, char*
# ---------------------------------------------------------------------------
#     el.parm.000                  el
# ---------------------------------------------------------------------------
#
#    a[1] = rho[1];
#    a[2] = rho[2];
#
#    B(1,1) = rho[3];
#    B(1,2) = rho[4];
#    B(2,2) = rho[5];
#    B(1,3) = rho[6];
#    B(2,3) = rho[7];
#    B(3,3) = rho[8];
#
#    R(1,1) = rho[9];
#    R(1,2) = rho[10];
#    R(2,2) = rho[11];
#
#    a[3] = -1.0;
#
#    B(2,1) = B(1,2);
#    B(3,1) = B(1,3);
#    B(3,2) = B(2,3);
#
#    R(2,1) = 0.0;
#
#    s = a + Bx
#    e = Rz
#
#    y[1] = log(s[1]/s[3]) + e[1]
#    y[2] = log(s[2]/s[3]) + e[2]
#
ESTIMATION DESCRIPTION (required)
    electric   Project name, pname, char*
         2.6   EMM version, defines format of this file, emmver, float
           1   Objfun type, 0 EMM, 1 MLE, 2 usr, objtype, int
           0   Proposal type, 0 group_move, 1 cond_move, 2 usr, proptype, int
           1   Write detailed output if print=1, int
         457   Seed for MCMC simulations, iseed, int
       20000   Number of MCMC simulations per file, lchain, int
           9   Number of MCMC simulation files beyond the first, nfile, int
      0.0625   Rescale proposal scaling by this value, sclfac, float
         1.0   Rescale parameter increments by this value, incfac, float
         1.0   Rescale objfun by this value, temperature, float
           0   Sandwich variance not computed if kilse=1, int
           1   The stride used to write MCMC simulations, stride, int
           0   Draw from prior if draw_from_prior=1, int
           0   Max cache size, must be 10 or more, max_cache_size, int
DATA DESCRIPTION (required) (mod and obj constructors see realmat data(M,n))
           5   Dimension of the data, M, int
         224   Number of observations, n, int
electric.dat   File name, any length, no embedded blanks, dsn, string
   1 2 3 4 5   Read these white space separated fields, fields, intvec
MODEL DESCRIPTION (required)
          11   Number of parameters, len_stat_parm, int
           1   Number of functionals, len_stat_func, int
MODEL PARMFILE (required) (constructor sees as vector<string> pfvec, alvec)
    __none__   File name, code __none__ if none, mod_parmfile, string
#begin additional lines
```

```
          23   Number of bootstrap repetitions, blen, int
#end additional lines
OBJFUN PARMFILE (required) (constructor sees as vector<string> pfvec, alvec)
    __none__   File name, code __none__ if none, obj_parmfile, string
#begin additional lines
#end additional lines
PARAMETER START VALUES (required)
  -2.92727122000000000e+00    1      a[1]
  -1.53786463000000000e+00    1      a[2]
  -1.28362479000000000e+00    1      B(1,1)
   0.81889299000000000e+00    1      B(1,2)
  -1.04835591000000000e+00    1      B(2,2)
   0.36106759000000000e+00    1      B(1,3)
   0.03049767000000000e+00    1      B(2,3)
  -0.46735947000000000e+00    1      B(3,3)
   0.26562000000000000e+00    1      R(1,1)
   0.30397000000000000e+00    1      R(1,2)
   0.29659000000000000e+00    1      R(2,2)
PROPOSAL SCALING (required)
   5.00000000000000000e-01    a[1]
   1.25000000000000000e-01    a[2]
   5.00000000000000000e-01    B(1,1)
   1.25000000000000000e-01    B(1,2)
   1.25000000000000000e-01    B(2,2)
   6.25000000000000000e-02    B(1,3)
   6.25000000000000000e-02    B(2,3)
   6.25000000000000000e-02    B(3,3)
   1.25000000000000000e-01    R(1,1)
   1.25000000000000000e-01    R(1,2)
   6.25000000000000000e-02    R(2,2)
```

And here is the file `summary.dat`.

```
     rhomean     rhomode      sesand       sehess      seinfo
     -3.0226     -2.9273     0.19681        0.251     0.33523
     -1.5537     -1.5379    0.084392     0.089628     0.10291
     -1.3046     -1.2836     0.17087      0.19372     0.24106
     0.80819     0.81889    0.081372      0.08216    0.087983
      -1.066     -1.0484    0.078783     0.080535    0.085266
     0.34557     0.36107    0.033553     0.030485    0.028781
    0.037249    0.030498     0.04201     0.037316    0.034982
    -0.46676    -0.46736    0.015915     0.016947    0.019363
     0.26891     0.26562    0.017964     0.012963    0.010089
      0.3129     0.30397    0.025988     0.022697       0.021
     0.30375     0.29659    0.014154     0.013614    0.014135

The log posterior (log prior - objfun) at the mode is      137.89.

The degrees of freedom for seinfo are 972.
```

Compare to page 368 of Gallant (1987). The match is pretty close but comparison is made tedious because the variables are ordered differently.

8 References

Aldrich, Eric M. and A. Ronald Gallant (2010), “Habit, Long Run Risks, Prospect? A Statistical Inquiry,” Working paper, Duke University, Fuqua School of Business, Durham

NC.

Ahn, D-H., R. F. Dittmar, and A. R. Gallant (2002), “Quadratic Term Structure Models: Theory and Evidence,” *The Review of Financial Studies*, 15, 243–288.

Andersen, Torben G. (1994), “Stochastic Autoregressive Volatiltiy: A Framework for Volatility Modeling,” *Mathematical Finance* 4, 74–102.

Andersen, Torben G. and Jesper Lund (1997a), “Estimating Continuous-time Stochastic Volatility Models of the Short Term Interest Rate,” *Journal of Econometrics*, 77, 343–378.

Andersen, Torben G. and Jesper Lund (1997b), “Stochastic Volatility and Mean Drift in the Short Term Interest Rate Diffusion: Implications for the Yield Curve,” Manuscript, Northwestern University.

Andrews, D. W. K. (1991), “Heteroskedasticity and Autocorrelation Consistent Covariance Matrix Estimation,” *Econometrica* 59, 307–346.

Bansal, R., A. R. Gallant, R. Hussey & G. Tauchen (1993), Computational aspects of nonparametric simulation estimation. In D. A. Belsley (ed.) *Computational Techniques for Econometrics and Economic Analysis*, pp. 3–22. Boston: Kluwer Academic Publishers.

Bansal, R., A. R. Gallant, R. Hussey, and G. Tauchen (1995), “Nonparametric Estimation of Structural Models for High-Frequency Currency Market Data,” *Journal of Econometrics* 66, 251–287.

Bansal, Ravi, A. Ronald Gallant, and George Tauchen (2007), “Rational Pessimism, Rational Exuberance, and Asset Pricing Models,” *Review of Economic Studies* 74, 1005–1033.

Bansal, R., and A. Yaron. (2004). “Risks For the Long Run: A Potential Resolution of Asset Pricing Puzzles.” *Journal of Finance 59,* 1481–1509.

Bansal, Ravi and H. Zhou (2002), "The Term Structure of Interest Rates with Regime Shifts," *Journal of Finance* 57, 1997–2043.

Barberis, N., M Huang and T. Santos (2001), "Prospect Theory and Asset Prices" *Quarterly Journal of Economics* 116, 1–54.

Brandt, M. W., and P. Santa-Clara (2002), "Simulated Likelihood Estimation of Multivariate Diffusions with an Application to Interest Rates and Exchange Rates with Stochastic Volatility," *Journal of Financial Economics* 63, 161–210.

Campbell, J. Y., and J. Cochrane. (1999). "By Force of Habit: A Consumption-based Explanation of Aggregate Stock Market Behavior." *Journal of Political Economy 107,* 205–251.

Clark, P. K. (1973), "A Subordinated Stochastic Process Model with Finite Variance for Speculative Prices," *Econometrica* 41, 135–56.

Chernov, Michael, A. Ronald Gallant, Eric Ghysels, and George Tauchen (1999), "A New Class of Stochastic Volatility Models with Jumps: Theory and Estimation," Working paper, University of North Carolina, Department of Econmics, Chapel Hill NC.

Chernov, Michael, A. Ronald Gallant, Eric Ghysels, and George Tauchen (2003), "Alternative Models for Stock Price Dynamics," *Journal of Econometrics* 116, 225–257 .

Chernov, Mikhail, and Eric Ghysels (2000), "A Study Towards a Unified Approach to the Joint Estimation of Objective and Risk Neutral Measures for the Purpose of Options Valuation," *Journal Of Financial Economics* 56, 407-458.

Chernozhukov, Victor, and Han Hong (2003), "An MCMC Approach to Classical Estimation," *Journal of Econometrics* 115, 293–346.

Chung, Chae-Shick, and George Tauchen (2001), "Testing Target Zone Models Using the Efficient Method of Moments, *JBES* Invited Address, *Journal of Business and Economic Statistics,* 19, 255–277.

Dai, Qiang, and Kenneth J. Singleton (2001), “Specification Analysis of Affine Term Structure Models,” *Journal of Finance* 55, 1943–1978.

Duffie, Darrell, and Kenneth J. Singleton (1993), “Simulated Moments Estimation of Markov Models of Asset Prices,” *Econometrica* 61, 929–952.

Durham, Garland (2006), “Monte Carlo Methods for Estimating, Smoothing, and Filtering One- and Two-Factor Stochastic Volatility Models,” *Journal of Econometrics* 133, 273–305.

Elerian, O., S. Chib, and N. Shephard (2001), “Likelihood Inference for Discretely Observed Nonlinear Diffusions,” *Econometrica* 69, 959–993.

Engle, R. F. (1982), “Autoregressive Conditional Heteroskedasticity with Estimates of the Variance of United Kingdom Inflation,” *Econometrica* 50, 987–1007.

Gallant, A. Ronald (1987), *Nonlinear Statistical Models*, New York: John Wiley and Sons.

Gallant, A. Ronald, and Han Hong (2007), “A Statistical Inquiry into the Plausibility of Recursive Utility,” *Journal of Financial Econometrics,* forthcoming.

Gallant, A. Ronald, Han Hong, and Ahmed Khwaja (2010), “Bayesian Estimation of a Dynamic Oligopolistic Game with Serially Correlated Unobserved Cost Variables,” Working paper, Duke University, Fuqua School of Business, Durham NC.

Gallant, A. Ronald, David A. Hsieh, and George E. Tauchen (1991), “On Fitting a Recalcitrant Series: The Pound/Dollar Exchange Rate, 1974–83,” in William A. Barnett, James Powell, George E. Tauchen, eds. *Nonparametric and Semiparametric Methods in Econometrics and Statistics, Proceedings of the Fifth International Symposium in Economic Theory and Econometrics,* Cambridge: Cambridge University Press, Chapter 8, pp. 199–240.

Gallant, A. Ronald, David A. Hsieh, and George Tauchen (1997), “Estimation of Stochastic Volatility Models with Diagnostics,” *Journal of Econometrics* Vol 81, No. 1, pp. 159–192.

Gallant, A. Ronald and Jonathan R. Long (1997), "Estimating Stochastic Differential Equations Efficiently by Minimum Chi-Square", *Biometrika*, 84, 125–141.

Gallant, A. Ronald, and Robert E. McCulloch (2009), "On the Determination of General Scientific Models with Application to Asset Pricing," *Journal of the American Statistical Association* 104, 117–131.

Gallant, A. Ronald, and Douglas W. Nychka (1987), "Seminonparametric Maximum Likelihood Estimation," *Econometrica* 55, 363–390.

Gallant, A. Ronald, Peter E. Rossi, and George Tauchen (1993), "Nonlinear Dynamic Structures," *Econometrica* 61, 871–907.

Gallant, A. Ronald, and George Tauchen (1989), "Seminonparametric Estimation of Conditionally Constrained Heterogeneous Processes: Asset Pricing Applications," *Econometrica* 57, 1091–1120.

Gallant, A. Ronald, and George Tauchen (1992), "A Nonparametric Approach to Nonlinear Time Series Analysis: Estimation and Simulation," in David Brillinger, Peter Caines, John Geweke, Emanuel Parzen, Murray Rosenblatt, and Murad S. Taqqu eds. *New Directions in Time Series Analysis, Part II.* New York: Springer-Verlag, 71-92.

Gallant, A. Ronald, and George Tauchen (1996a), "Which Moments to Match," *Econometric Theory* 12, 657–681.

Gallant, A. Ronald, and George Tauchen (1996b), "Specification Analysis of Continuous Time Models in Finance," in P. E. Rossi (ed.), *Modeling Stock Market Volatility: Bridging the Gap to Continuous Time* (New York: Academic Press, 1996b).

Gallant, A. Ronald, and George Tauchen (1997), "Estimation of Continuous Time Models for Stock Returns and Interest Rates," *Macroeconomic Dynamics*, Vol. 1, No. 1, 135–168.

Gallant, A. Ronald, and George Tauchen (1998), "Reprojecting Partially Observed Systems with Application to Interest Rate Diffusions," *Journal of the American Statistical Association* 93, 10–24.

Gallant, A. Ronald, and George Tauchen (2001), "Efficient Method of Moments" Manuscript, Duke University.

Gallant, A. Ronald, and George Tauchen (2002a), "Simulated Score Methods and Indirect Inference for Continuous-time Models," Chapter in preparation for the Handbook of Financial Econometrics.

Gallant, A. Ronald, and George Tauchen (2002b), "SNP: A Program for Nonparametric Time Series Analysis, Version 9.0, User's Guide," Manuscript, Duke University. Available along with code and worked example by anonymous ftp at site ftp.econ.duke.edu in directory pub/arg/snp_cpp.

Gallant, A. Ronald, and George Tauchen (2009), "Simulated Score Methods and Indirect Inference for Continuous-time Models," in Yacine Aït-Sahalia and Lars Peter Hansen, eds. (2009), *Handbook of Financial Econometrics,* Elsevier/North-Holland, Amsterdam.

Gamerman, D., and H. F. Lopes (2006), *Markov Chain Monte Carlo: Stochastic Simulation for Bayesian Inference (2nd Edition),* Chapman and Hall, Boca Raton, FL.

Gourieroux, C., A. Monfort, and E. Renault (1993), "Indirect Inference," *Journal of Applied Econometrics* 8, S85–S118.

Hansen, L. P. (1982), Large Sample Properties of Generalized Method of Moments Estimators. *Econometrica* 50, 1029–1054.

Jacquier, E., Polson, N. G., and P. E. Rossi (1994), "Bayesian Analysis of Stochastic Volatility Models," *Journal of Business and Economic Statistics*, 12, 371–388.

Kloeden, Peter E. and Eckhard Platen (1992), *Numerical Solution of Stochastic Differential Equations*. New York: Springer-Verlag.

McFadden, D. (1989), A Method of Simulated Moments for Estimation of Discrete Response Models Without Numerical Integration. *Econometrica* 57, 995–1026.

Nelson, D. (1991), "Conditional Heteroskedasticity in Asset Returns: A New Approach," *Econometrica* 59, 347–370.

Newey, W. (1985), "Conditional Moment Specification Testing," *Econometrica* 53, 1047–1071.

Pakes, A. and D. Pollard (1989), "Simulation and the Asymptotics of Optimization Estimators," *Econometrica* 57, 1027–1058.

Politis, D. N., and J. P. Romano (1994), "The Stationary Bootstrap," *Journal of the American Statistical Association* 89, 1303–1313.

Schwarz, G. (1978), "Estimating the Dimension of a Model," *Annals of Statistics* 6, 461–464.

Shephard, N. (2004), *Stochatic Volatility: Selected Readings* Oxford University Press.

Smith, A. A. (1993), "Estimating Nonlinear Time Series Models Using Simulated Vector Autoregressions," *Journal of Applied Econometrics*, 8, S63–S84.

Tauchen, George (1985), "Diagnostic Testing and Evaluation of Maximum Likelihood Models," *Journal of Econometrics* 30, 415–443.

Tauchen, George (1997), "New Minimum Chi-Square Methods in Empirical Finance," in *Advances in Econometrics, Seventh World Congress*, eds. D. Kreps, and K. Wallis, Cambridge UK: Cambridge University Press, 279–317.

Tauchen, George (1998), The Objective Function of Simulation Estimators Near the Boundary of the Unstable Region of the Parameter Space," *Review of Economics and Statistics* 389–398.

Tauchen, George, and Robert Hussey (1991), "Quadrature-Based Methods for Obtaining Approximate Solutions to Nonlinear Asset Pricing Models," *Econometrica* 59, 371–396.

Tauchen, George and Mark Pitts (1983), "The Price Variability-Volume Relationship on Speculative Markets," *Econometrica* 51, 485–505.

Yu, Jun, (2005), "On Leverage in a Stochastic Volatility Model," *Journal of Econometrics* 127, 165–178.