MLE: Maximum Likelihood Estimation by MCMC

Version 1.1

User's Guide [1]

A. Ronald Gallant
Penn State University
Department of Economics
University Park PA 16802 USA

May 2013
Last Revised June 2013

[1]The code and this guide are available at http://www.aronaldg.org.

ABSTRACT

This guide shows how to use the computer package MLE. It provides instructions on how to install the software and a description of the package. It also walks the reader through two examples.

The MLE and EMM packages are similar, for many problems either would work. Both are misnomers because they cover a much broader class of estimators than their names suggest. The difference between them is that EMM presumes that a model can be simulated and MLE presumes that it cannot.

The main practical difference is that the heteroskedasticity autoregressive consistent (HAC) estimator of the information matrix that goes in the middle of a sandwich variance estimator can be computed by parametric bootstrap if a model can be simulated. A sandwich variance, $\mathcal{J}^{-1}\mathcal{I}\mathcal{J}^{-1}$, has the inverse of the Hessian of the log likelihood on either side and has a HAC estimate of the variance of the gradient of the log likelihood in the middle. In both EMM and MLE the inverse of the Hessian is estimated by the variance matrix of an MCMC chain. EMM can compute a bootstrap HAC from a simulation or a stationary bootstrap. MLE cannot and instead relies on user supplied scores.

The design of the EMM code, although facilitating bootstrap estimation of the HAC in the middle, makes it impossible to use EMM for the Metropolis within Gibbs estimator for generalized method of moments (GMM) estimation of models with latent variables proposed by Gallant, Giacomini, and Ragusa (2013). The technical reason is that EMM uses two instantiations of the user supplied model (one for the MCMC chain and another for the HAC in the middle) that would need to communicate with each other to implement Metropolis within Gibbs but cannot within EMM without offense to the principles of object oriented programming and parallel-safe design. MLE was written to fix this problem.

An acquaintance with the *EMM User's Guide* would be helpful but in principle this *Guide* is self contained, excepting the Introduction, which compares *EMM* to *MLE*. The development is based on two examples.

The first example is the consumer demand application in the EMM distribution. It illustrates the difference in the user supplied code between the EMM and MLE packages. In the latter the user provides the scores and this *Guide* illustrates how it is done.

The second is the stochastic volatility example from Gallant, Giacomini, and Ragusa (2013), which is Metropolis within Gibbs applied to a likelihood derived from moment equations.

The code and this guide are available at http://aronaldg.org/webfiles/mle.

Contents

1 Introduction

1.1 Overview

A comparison of the MLE package to the EMM package is a quick way to introduce the latter.

To use the EMM package for maximum likelihood estimation one must: (1) Write code to implement the class `emmusr` that inheirits from `emmusr_base`, fill out a parmfile named, for example, `el.parm.000` that sets the values of program `emm` control parameters, and a file named, for example, `control.dat`, whose lines have two blank separated items per line. The first item is the filename of the parmfile to be used; the second item is the prefix for all output files. For example, `control.dat` might have one line that reads `el.parm.000 el`. If `emm` is entered on the command line then the file named `control.dat` will be read. If `emm el.ctrl.000` then `el.ctrl.000` will be read instead of `control.dat`.

To use the EMM package for maximum likelihood estimation, two of the members in the class `emmusr` are the most relevant: member `likelihood`, which computes the likelihood, and member `gen_bootstrap`, which provides the bootstrap sample for computing the HAC matrix in the middle of the sandwich variance.

Everything is the same for the MLE program with the exceptions that the user written class is named `mleusr`, it inheirits from `mleusr_base`, and a few of the lines of the parmfile are different. The most important difference between `emmusr` and `mleusr` is that member `gen_bootstrap` is replaced by `get_scores`.

This is probably not a small change from the user's point of view because computation of the scores requires evaluation and differentiation of a log density at each observation and storage of the result whereas the log likelihood requires only evaluation and accumulation without need for storage. If scores are computed numerically, as in our first example, then managing store is the only practical difference. If scores are computed analytically, the differences are more substantial.

1.2 The Chernozhukov and Hong Method

The computational methods discussed here and implemented by the MLE 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

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

Quasi maximum likelihood estimation requires the computation of the estimator itself, $\hat{\rho}_n = \underset{\rho}{\text{argmin}}\, s_n(\rho)$, 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

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

An estimator of $\mathcal{I}$ is what we have termed the HAC in the middle previously.

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

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

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, one can use the mode of $\ell(\rho)$ as the estimator $\hat{\rho}_n$. The MLE package computes and reports both the mean and the mode. There is the problem, however, that a mode computed by finding the maximum of the likelihood over an MCMC chain is not as accurate as that provided by a derivative based optimizer. Nonetheless, the attraction of the mode, besides the fact that it is the maximum of the likelihood, is that the mode will satisfy the support conditions of the structural model whereas the mean may not.

The strategy used to estimate $\mathcal{I}$ from the scores is standard. See, e.g., Gallant (1987). The MLE package uses Parzen weights.

The MCMC method described here makes the imposition of support restrictions, inequality restrictions, and informative prior information exceptionally convenient.

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, member `set_rho` of `mleusr` is called before member `support`. The user should be aware of this fact when writing code for the structural model because one is not guaranteed that a ρ set by MLE will satisfy support conditions. However, one can count on member `support` being called immediately after `set_rho`. If it returns false, then no other member of the user's implementation of the structural model is called.

1.3 Using this Guide

New users should install as in Section 2, skim Section 3, and work through the `elec` example developed in Section 4. Thereafter, use Section 3 for reference purposes.

2 Building and Running MLE

2.1 Availability

C++ code and this *Guide* as a PostScript or PDF file are available at `http://www.aronaldg.org/webfiles/mle`. This code runs under Linux and MacOS. On a Windows machine one can use either Cygwin at `http:/www.cygwin.com` or MinGW at `http://www.mingw.org`. It has not been tested on other platforms.

2.2 Building and Running MLE

Download `mle.tar` from `http://www.aronaldg.org/webfiles/mle`. On a Unix machine use `tar -xf mle.tar` to expand the tar archive into a directory that will be named `mle`.

mle
- elec
- lib
  - libscl
  - libmle
- mleman
- mlerun
- mlesrc
- svsim
- utility

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

First the two libraries libscl and libmle must be built, in that order. Change directory to `lib/libscl/gpp` and type `make`. Then change directory to `lib/libmle/gpp` and type `make`.

To run the `elec` example that comes with the MLE distribution, within `emmrun` copy `makefile.gpp` to `makefile`, type `make` and then `./mle elec.ctrl.000`.

3 The Structure of the MLE 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.

3.1 User Supplied Class

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

```
#include "libscl.h"

namespace libmle {

  class usrmod_base {
  public:
    virtual INTEGER len_rho() = 0;
    virtual INTEGER len_stats() = 0;
    virtual bool get_stats(scl::realmat& stats) = 0;
    virtual void get_rho(scl::realmat& rho) = 0;
    virtual void set_rho(const scl::realmat& rho) = 0;
    virtual void set_rho_old(const scl::realmat& rho) { return; }
    virtual bool support(const scl::realmat& rho) = 0;
    virtual scl::den_val prior(const scl::realmat& rho,
      const scl::realmat& stats) = 0;
    virtual scl::den_val likelihood() = 0;
    virtual bool get_scores(scl::realmat& scores)
      {scl::error("Error, usrmod_base, get_scores"); return false;}
    virtual void write_usrvar(const char* filename) { return; }
    virtual ~usrmod_base() {}
  };

}
#endif
```

and here is the corresponding `mleusr.h`

```
#ifndef __FILE_MLEUSR_H_SEEN__
#define __FILE_MLEUSR_H_SEEN__

#include "libmle.h"
#include "mle_base.h"

namespace mle {

  class elec_usrmod;

  typedef elec_usrmod usrmod_type;

  class elec_usrmod : public libmle::usrmod_base {
  private:
    scl::realmat data;
    scl::realmat rho;
    INTEGER lrho;
    INTEGER lstats;
    INT_32BIT variable_seed;
    scl::realmat a;
    scl::realmat B;
    scl::realmat R;
    scl::realmat yhat;
    scl::realmat ehat;
    scl::realmat zhat;
    void set_parms();
    bool get_obj(scl::realmat& obj);
  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 get_stats(scl::realmat& stats);
    void get_rho(scl::realmat& parm) { parm = rho; }
    void set_rho(const scl::realmat& parm) { rho = parm; set_parms();}
    bool support(const scl::realmat& rho);
    scl::den_val prior(const scl::realmat& rho,
      const scl::realmat& stats);
    scl::den_val likelihood();
    bool get_scores(scl::realmat& scores);
    void write_usrvar(const char* filename)
    {
      std::string stem = filename;
      size_t suffix = stem.find(std::string("dat"));
      if (suffix != std::string::npos) stem = stem.substr(0, suffix);
      if (likelihood().positive) {
        std::string fn;
        fn = stem + "yhat.dat"; vecwrite(fn.c_str(), yhat);
        fn = stem + "ehat.dat"; vecwrite(fn.c_str(), ehat);
        fn = stem + "zhat.dat"; vecwrite(fn.c_str(), zhat);
      }
    }
  };
}

#endif
```

Class `elec_usrmod` gets bound to program `mle` 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 code to support MLE including a nonlinear equation solver and a nonlinear optimizer. Of special interest is class gmm that implements generalized method of moments estimation, which we will use in our second example.

The idea behind `stats` is that there is more information about a model that the user needs to know besides the value of `rho` that generated it. Member `stats` is more useful for simulation based estimation where one computes a statistic from a simulation to use with the prior for checking conditions that can only be determined by simulation. It is of less use with the MLE package. For more information regarding `stats`, see the *EMM User's Guide*

The realmat `stats` of length `lstats` gets written to a file by program `mle` as does `rho` and much else as described later.

The method `prior` is exactly what its name suggests and can be used to implement Bayesian inference. It can also be used to implement support conditions that depend on `stats` in frequentist inference. If `prior` is not to be used, one codes it to always return `scl::denval(true,0)`. An `scl::denval` is a struct whose first member is a `bool` that is true if the argument of a density is in the support of the density second member is a `REAL` that contains the log density if it is.

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 `stats` and `prior` after. The intent is to save the cost of an unnecessary computation if `rho` violates support conditions that can be cheaply determined. Be aware that `set_rho` is always called before `support`.

In this example, the real work is done by private member function `get_obj`. It computes the log density of the data for each observation and places the result in its `scl::realmat` argument `obj`. Member `likelihood` calls `get_obj` and sums the elements of `obj`. This sum is the log likelihood that `likelihood` returns as an `scl::denval`, whose first member is true if `rho` is in the support of the log likelihood and whose second member contains the log likelihood if it is.

Member `get_scores` calls `get_obj` repeatedly at different parameter values to compute the scores by numerical differentiation and returns the result in the `scl::realmat` argument `scores`. Private member `set_parms` copies the elements of `rho` into the models parameters: `a, B, R`.

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 `mle.cpp` will be required.

All other members in this example are trivial and can be coded in the header as shown.

3.2 The Input Parameter File

The MLE input parameter file contains several blocks of control information. An example, used again in Section 4, is

```
PARMFILE HISTORY (optional)
#
# This parmfile was written by MLE Version 1.1 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*
         1.1   mle version, defines format of this file, mlever, float
           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 draws, iseed, int
       20000   Number of MCMC draws per output file, lchain, int
           9   Number of MCMC output files beyond the first, nfile, int
         1.0   Rescale proposal scaling by this value, sclfac, float
         1.0   Rescale likelihood by this value, temperature, float
           0   Sandwich variance not computed if kilse=1, int
           0   Number of lags in HAC middle of sandwich variance, lhac, int
           1   The stride used to write MCMC draws, stride, int
           0   Draw from prior if draw_from_prior=1, int
DATA DESCRIPTION (required) (mod constructor sees 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 modal parameters, len_mod_parm, int
           2   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
#end additional lines
PARAMETER START VALUES (required)
  -2.92727122000000017e+00    1      a[1]
  -1.53786463000000007e+00    1      a[2]
  -1.28362478999999996e+00    1      B(1,1)
   8.18892990000000043e-01    1      B(1,2)
  -1.04835590999999995e+00    1      B(2,2)
   3.61067589999999994e-01    1      B(1,3)
   3.04976700000000010e-02    1      B(2,3)
  -4.67359469999999999e-01    1      B(3,3)
   2.65620000000000023e-01    1      R(1,1)
   3.03970000000000018e-01    1      R(1,2)
   2.96590000000000020e-01    1      R(2,2)
PROPOSAL SCALING (required)
   3.12500000000000000e-02    a[1]
   7.81250000000000000e-03    a[2]
   3.12500000000000000e-02    B(1,1)
   7.81250000000000000e-03    B(1,2)
   7.81250000000000000e-03    B(2,2)
   3.90625000000000000e-03    B(1,3)
   3.90625000000000000e-03    B(2,3)
   3.90625000000000000e-03    B(3,3)
   7.81250000000000000e-03    R(1,1)
   7.81250000000000000e-03    R(1,2)
   3.90625000000000000e-03    R(2,2)
```

A description of each block of the input file follows.

3.2.1 PARMFILE HISTORY

This block is optional. It is written by `mle` 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. In the example, lines describing the model are included in the history.

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.

mlever: Version of MLE.

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 the *EMM User's Guide*. 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 `mleusr.h` and `mleusr.cpp`. At the beginning of `mleusr.h` one would need to insert the compiler directive

```
#define USR_PROPOSAL_TYPE_IMPLEMENTED
```

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

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

Examples are in `proposal.cpp` in `libmle/src`.

print: If `print=1`, then voluminous debugging information is written to file detail.dat. Setting `print=0` suppresses printing.

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 $R =$ `lchain`*(`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.

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 likelihood 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, setting `kilse` = 1 will stop them from being computed. When `kilse` = 1, `scores` is not called and does not need to be coded unless, as the `elec` example, it is called by the user's code. 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 illustrated by example in the *EMM User's Guide* and discussed in texts such as Gamerman and Lopes (2006).

lhac : The number of lags to be used to compute the HAC information matrix in the middle of the sandwich variance estimator. Set `lhac=0` if the scores are uncorrelated, in which case the estimator is heteroskedastic consistent.

stride : MLE 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 MLE 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, briefly, 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.

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".

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.

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.

#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]`.

3.2.6 PARAMETER START VALUES

The block labeled `PARAMETER START VALUES` specifies the first value for the chain. It must satisfy the support conditions; i.e. `elec_usrmod::support` must return true, and `elec_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`.

3.2.7 PROPOSAL GROUPING

How to specify group moves in the parmfile is discussed and illustrated in the *EMM User's Guide*. We will not be using it in the examples of this *Guide*.

3.3 Directory Structure

The directory structure of the MLE distribution is as follows:

3.3.1 mlesrc

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

3.3.2 elec

This directory contains the files `mleusr.h` and `mleusr.cpp` for fitting a translog electricity demand system by maximum likelihood. It is our first example.

3.3.3 svsim

This directory contains the code for stochastic volatility estimated by means of the Metropolis within Gibbs GMM estimator proposed by Gallant, Giacomini, and Ragusa (2013) that is used for the second example in this *Guide*. For the `svfx` example, the class `usrmod` that the user must supply is presented in `mleusr.h` and defined in `mleusr.cpp`. Also present are makefiles and data for that example.

3.3.4 lib

The directory `lib` contains the two libraries `libscl` and `libmle`. The source code is in `libscl/src`, the makefiles are in `libscl/gpp`. Compilation is done within `libscl/gpp` for Linux the library and headers reside in `libscl/gpp` after compilation. Similarly for and

`libmle`. If one uses something other such as Portland Group than or in addition to GNU's suite, we suggest compiling in a directory such as `pgi`.

3.3.5 mlerun

The directory `mlerun` contains the makefile to build the `mle` executable and input files to run the `elec` example. Type `make` and the `mle` 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:

```
el.parm.000  el
```

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

```
mle el.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`.

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.

4 Maximum Likelihood Estimation of the Electricity Example

Here is the `mleusr.h` that declares `elec_usrmod`:

```
#ifndef __FILE_MLEUSR_H_SEEN__
#define __FILE_MLEUSR_H_SEEN__

#include "libmle.h"
#include "mle_base.h"

namespace mle {

  class elec_usrmod;

  typedef elec_usrmod usrmod_type;

  class elec_usrmod : public libmle::usrmod_base {
  private:
    scl::realmat data;
    scl::realmat rho;
    INTEGER lrho;
    INTEGER lstats;
    scl::realmat a;
    scl::realmat B;
    scl::realmat R;
    scl::realmat yhat;
    scl::realmat ehat;
    scl::realmat zhat;
    void set_parms();
    bool get_obj(scl::realmat& obj);
  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 get_stats(scl::realmat& stats);
  void get_rho(scl::realmat& parm) { parm = rho; }
  void set_rho(const scl::realmat& parm) { rho = parm; set_parms();}
  bool support(const scl::realmat& rho);
  scl::den_val prior(const scl::realmat& rho, const scl::realmat& stats);
  scl::den_val likelihood();
  bool get_scores(scl::realmat& scores);
  void write_usrvar(const char* filename)
  {
    std::string stem = filename;
    size_t suffix = stem.find(std::string("dat"));
    if (suffix != std::string::npos) stem = stem.substr(0, suffix);
    if (likelihood().positive) {
      std::string fn;
      fn = stem + "yhat.dat"; vecwrite(fn.c_str(), yhat);
      fn = stem + "ehat.dat"; vecwrite(fn.c_str(), ehat);
      fn = stem + "zhat.dat"; vecwrite(fn.c_str(), zhat);
    }
  }
 };
}

#endif
```

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

We are using 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 MLE 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 parametrize 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 `mleusr.cpp` that defines `elec_usrmod`:

```
#include "libmle.h"
#include "mle.h"
#include "mleusr.h"

using namespace scl;
using namespace libmle;
using namespace mle;
using namespace std;

mle::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(), lrho(11), lstats(2), a(), B(), R(),
  yhat(2,dat.ncol()), ehat(2,dat.ncol()), zhat(2,dat.ncol())
{
  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");
  }
}

void mle::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;
}

bool mle::elec_usrmod::get_obj(realmat& obj)
{
  INTEGER n = data.get_cols();

  yhat.resize(2,n,0.0);
  ehat.resize(2,n,0.0);
  zhat.resize(2,n,0.0);

  obj.resize(1,n,-REAL_MAX);

  if (!support(rho)) return false;

  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 i=1; i<=s.size(); ++i) {
    if (s[i] >= 0.0) return false;
  }

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

  const REAL pi = 3.14159265358979312e+00;

  for (INTEGER t=1; t<=n; ++t) {
    obj[t] = pow(zhat(1,t),2) + pow(zhat(2,t),2);
    obj[t] *= (-0.5);
    obj[t] -= log(detR);
    obj[t] -= log(sqrt(2.0*pi));
  }

  return true;
}

den_val mle::elec_usrmod::likelihood()
{
```

```
  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 obj;

  if (!get_obj(obj)) return den_val(false,-REAL_MAX);;

  REAL q = 0.0;

  for (INTEGER t=1; t<=n; ++t) {
    q += obj[t];
  }

  return den_val(true,q);
}

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

bool mle::elec_usrmod::get_stats(scl::realmat& stats)
{
  stats.resize(2,1);
  INTEGER n = zhat.ncol();
  stats[1] = stats[2] = 0.0;
  for (INTEGER j=1; j<=n; ++j) {
    stats[1] += zhat(1,j);
    stats[2] += zhat(2,j);
  }
  stats[1] /=n; stats[2] /= n;
  return true;
}

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

bool mle::elec_usrmod::get_scores(scl::realmat& scores)
{
  realmat rhosave = rho;

  INTEGER n = data.get_cols();

  scores.resize(lrho,n,0.0);

  REAL eps = std::pow(double(REAL_EPSILON),0.33333333);

  for (INTEGER i=1; i<=lrho; ++i) {
    REAL tmp = rho[i];

    REAL h = eps*std::fabs(tmp);
    if (h == 0.0) h = eps;

    REAL hi = tmp + h;
    rho[i] = hi;
```

```
      set_rho(rho);

      realmat f1;
      if (!get_obj(f1)) {set_rho(rhosave); return false;}

      REAL lo = tmp - h;
      rho[i] = lo;
      set_rho(rho);

      realmat f0;
      if (!get_obj(f0)) {set_rho(rhosave); return false;}

      REAL difference = hi - lo;

      for (INTEGER t=1; t<=n; ++t) {
        scores(i,t) = (f1[t] - f0[t])/difference;
      }

      rho[i] = tmp;
    }

    set_rho(rhosave);

    return true;
  }
```

All is as described earlier but attention needs to be called to member functions (methods) `set_parms`, `get_obj`, `likelihood, and get_scores`.

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

The method `get_obj` 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 returns them in its `realmat&` argument `obj`. Note in particular the statement `if (s[i] >= 0.0) return false;` that imposes the constraint that s have all entries negative for all x_t. As intermediate steps it computes the vectors `yhat`, `ehat` and `zhat`, which we write for use as diagnostics in `write_usevar` coded in `mleusr.h`.

All `likelihood` has to do is call `get_obj`, sum the elements of `obj`, and return the sum as a `denval`.

As coded here, the method `get_scores` calls `get_obj` repeatedly at various settings for `rho` in order to computed numerical derivatives according to standard formulae.

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 MLE Version 1.1 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*
         1.1   mle version, defines format of this file, mlever, float
           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 draws, iseed, int
       20000   Number of MCMC draws per output file, lchain, int
           9   Number of MCMC output files beyond the first, nfile, int
         1.0   Rescale proposal scaling by this value, sclfac, float
         1.0   Rescale likelihood by this value, temperature, float
           0   Sandwich variance not computed if kilse=1, int
           0   Number of lags in HAC middle of sandwich variance, lhac, int
           1   The stride used to write MCMC draws, stride, int
           0   Draw from prior if draw_from_prior=1, int
DATA DESCRIPTION (required) (mod constructor sees 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 modal parameters, len_mod_parm, int
           2   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
#end additional lines
```

```
PARAMETER START VALUES (required)
  -2.92727122000000017e+00    1      a[1]
  -1.53786463000000007e+00    1      a[2]
  -1.28362478999999996e+00    1      B(1,1)
   8.18892990000000043e-01    1      B(1,2)
  -1.04835590999999995e+00    1      B(2,2)
   3.61067589999999994e-01    1      B(1,3)
   3.04976700000000010e-02    1      B(2,3)
  -4.67359469999999999e-01    1      B(3,3)
   2.65620000000000023e-01    1      R(1,1)
   3.03970000000000018e-01    1      R(1,2)
   2.96590000000000020e-01    1      R(2,2)
PROPOSAL SCALING (required)
   3.12500000000000000e-02    a[1]
   7.81250000000000000e-03    a[2]
   3.12500000000000000e-02    B(1,1)
   7.81250000000000000e-03    B(1,2)
   7.81250000000000000e-03    B(2,2)
   3.90625000000000000e-03    B(1,3)
   3.90625000000000000e-03    B(2,3)
   3.90625000000000000e-03    B(3,3)
   7.81250000000000000e-03    R(1,1)
   7.81250000000000000e-03    R(1,2)
   3.90625000000000000e-03    R(2,2)
```

And here is the file `summary.dat`.

```
parm      rhomean      rhomode       sesand       sehess       seinfo
   1       -3.0226      -2.9273      0.22164        0.251      0.30973
   2       -1.5537      -1.5379     0.086409     0.089628      0.10047
   3       -1.3046      -1.2836      0.15577      0.19372      0.26437
   4       0.80819      0.81889     0.084848      0.08216     0.084678
   5        -1.066      -1.0484     0.086415     0.080535     0.093049
   6       0.34557      0.36107     0.034945     0.030485     0.030173
   7     0.037249     0.030498     0.037116     0.037316     0.040359
   8     -0.46676     -0.46736     0.014801     0.016947     0.022628
   9      0.26891      0.26562     0.018419     0.012963    0.0096423
  10       0.3129      0.30397      0.02507     0.022697     0.022151
  11      0.30375      0.29659      0.01564     0.013614      0.01377

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

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

5 GMM with Latent Variables

Before proceeding to our next example, we shall describe the method for estimating the parameters of a structural model proposed by Gallant, Giacomini, and Ragusa (2013). The estimator is implemented by a Metropolis within a Gibbs algorithm with Chernozukov and Hong's (2003) MCMC algorithm used to implement the Metropolis step and Andrieu,

Douced, and Holenstein's (2010, Subsection 4.1) modified particle filter algorithm used to implement the Gibbs step.

We require a structural model that has parameters θ, a vector. We denote the true but unknown value of the parameters by θ^o. We observe the the history $X = (X_1, X_2, ..., X_T)$, a subset of the endogenous and exogenous variable in the model. We do not observe the variables in the model that remain: $\Lambda = (\Lambda_1, \Lambda_2, ..., \Lambda_T)$. These are the latent variables. Partial histories are denoted $X_{1:t} = (X_1, X_2, ..., X_t)$ and $\Lambda_{1:t} = (\Lambda_1, \Lambda_2, ..., \Lambda_t)$.

We assume that we can draw from the transition density of the latent variables $\Lambda_{t+1} \sim P(\Lambda_{t+1} \,|\, \Lambda_t, \theta)$. The transition density is assumed to be ergodic. Note that this implies that we can also draw from the stationary density $P(\Lambda_t \,|\, \theta)$ by drawing from $P(\Lambda_{t+1} \,|\, \Lambda_t, \theta)$ with an arbitrary start Λ_0 and waiting for transients to die out.

The type of model that this setup describes that comes immediately to mind is a state space model whereby the latent variables are the driving process or shocks and the remainder of the model is the measurement equation. Although this is the typical form of a model that satisfies the requirements set forth here, we do not assume that the model has a state space representation.

We are given a set of conditional moment conditions of the form

$$\mathcal{E}[g(X_{t+1}, \Lambda_{t+1}, \theta) \,|\, \mathcal{I}_t] = 0,$$

where $g(\cdot, \cdot, \cdot)$ is M-dimensional. The information set is $\mathcal{I}_t = \{X_{-\infty}, ..., X_t, \, \Lambda_{-\infty}, ..., \Lambda_t\}$. We assume that the corresponding unconditional moment equations

$$\mathcal{E}[g(X_{t+1}, \Lambda_{t+1}, \theta)] = 0 \tag{2}$$

would identify θ if both X and Λ were observed.

The corresponding sample moment conditions are

$$g_T(X, \Lambda, \theta) = \frac{1}{\sqrt{T}} \sum_{t=1}^{T} g(X_t, \Lambda_t, \theta)$$

with weighting matrix

$$\Sigma(X, \Lambda, \theta) = \frac{1}{T} \sum_{t=1}^{T} \widetilde{g}(X_t, \Lambda_t, \theta)' \widetilde{g}(X_t, \Lambda_t, \theta) \tag{3}$$

$$\widetilde{g}(X_t, \Lambda_t, \theta) = g(X_t, \Lambda_t, \theta) - \frac{1}{\sqrt{T}} g_T(X, \Lambda, \theta) \tag{4}$$

If the moment conditions are serially correlated one will have to substitute a heteroskedastic autoregessive consistent (HAC) weighting matrix (Andrews, 1991) for that shown as (3). If a HAC matrix is used, the residuals used to compute it should be of the form shown as (4). We assume that the moment conditions normalized by the weighting matrix are asymptotically normal; i.e.,

$$Z = \left[\Sigma(X, \Lambda, \theta^o)\right]^{-1/2} g_T(X, \Lambda, \theta^o) \xrightarrow{d} N(0, I)$$

Regularity conditions such that asymptotic normality obtains are in Hansen and Singleton (1982), Gallant and White (1987), and elsewhere.

Define

$$p(X, \Lambda, \theta) = (2\pi)^{-M/2} \exp\left\{ -\frac{1}{2} g_T(X, \Lambda, \theta)' \left[\Sigma(X, \Lambda, \theta)\right]^{-1} g_T(X, \Lambda, \theta) \right\} \tag{5}$$

Our most important assumption is that the Chernozhukov and Hong (2003) result holds; that is, a sample $\left\{\theta^{(i)}\right\}_{i=1}^{R}$ from the density

$$p(\theta \,|\, X, \Lambda) \propto p(X, \Lambda, \theta) \tag{6}$$

is a sample from the asymptotic distribution of the GMM estimator for large T. Thus, all we have to do to achieve the goal of this paper is provide an algorithm that generates an MCMC chain for (6).

We do this by sampling $\left\{\theta^{(i)}, \Lambda^{(i)}\right\}$ from the density

$$p(\theta, \Lambda \,|\, X) \propto p(X, \Lambda, \theta) \tag{7}$$

using a Metropolis within Gibbs algorithm and discarding the Λ draws. The method is as follows:

1. Initialization. Choose a reasonable start $(\theta^{(0)},\ \Lambda^{(0)})$ and set $i = 1$.

2. Sample $\theta^{(i)}$ from $p(\theta \,|\, X, \Lambda^{(i-1)})$ using a Metropolis algorithm (Subsection 5.3).

3. Sample $\Lambda^{(i)}$ from $p(\Lambda \,|\, X, \theta^{(i)})$, where

$$p(\Lambda \,|\, X, \theta) \propto p(X, \Lambda, \theta), \tag{8}$$

   using a modified particle filter (Subsection 5.2).

4. Increment i and repeat from Step 2 until i exceeds some preassigned value R.

We impose an additional requirement that is a considerable convenience:

$$\int_{-\infty}^{\infty}\cdots\int_{-\infty}^{\infty}(2\pi)^{-M/2}\exp\left\{-\frac{1}{2}g_T(X,\Lambda,\theta)'\left[\Sigma(X,\Lambda,\theta)\right]^{-1}g_T(X,\Lambda,\theta)\right\}dX_1\cdots dX_T=1, \quad (9)$$

which implies

$$p(X\,|\,\Lambda,\theta)=p(X,\Lambda,\theta). \quad (10)$$

As yet we have not encountered a practical application that violates this condition. Usually all that is required is that each element of g is unbounded with respect to some element of X_t and that the residuals used to compute the weighting matrix are centered as in (4).

If the integral in (9) does not integrate to one, but one has a convenient means to compute it, then this requirement can be eliminated by using

$$p^{\#}(X\,|\,\Lambda,\theta)=\frac{\exp\left\{-\frac{1}{2}g_T(X,\Lambda,\theta)'\left[\Sigma(X,\Lambda,\theta)\right]^{-1}g_T(X,\Lambda,\theta)\right\}}{\int_{-\infty}^{\infty}\cdots\int_{-\infty}^{\infty}\exp\left\{-\frac{1}{2}g_T(X,\Lambda,\theta)'\left[\Sigma(X,\Lambda,\theta)\right]^{-1}g_T(X,\Lambda,\theta)\right\}dX_1\cdots dX_T} \quad (11)$$

instead of $p(X\,|\,\Lambda,\theta)$ to implement the particle filters in Subsections 5.1 and 5.2.

We discard the Λ draws to avoid excessive consumption of disk space and because there is little use for the joint distribution of θ and Λ in frequentist inference. Of more use is to generate countefactuals. For this one needs the ordinary particle filter algorithm (Subsection 5.1) for generating draws from the conditional distribution of Λ given X at the estimated or some other value of θ.

Sometimes one uses a penalty function in connection with MCMC using (7). In our examples we shall investigate the effect of multiplying (7) by a Jacobian term $[\det\Sigma(X,\Lambda,\theta)]^{-M/2}$.

Three algorithms are required to implement our proposal:

- A particle filter (PF) algorithm.
  - Input: θ.
  - Output: Draws $\left\{\Lambda^{(i)}\right\}_{i=1}^{R}$ from $P(\Lambda\,|\,X,\theta)$.
- A modified particle filter algorithm.
  - Input: The previous draw $\Lambda^{(i-1)}$ and a draw $\theta^{(i)}$ from $p(\theta\,|\,X,\Lambda^{(i-1)})$.
  - Output: A draw $\Lambda^{(i)}$ from $P(\Lambda\,|\,X,\theta^{(i)})$.

- A Metropolis algorithm.
  - Input: Λ.
  - Output: A draw θ from $p(\theta \,|\, X, \Lambda)$.

We present them in turn.

We previously introduced the notation $X_{1:t} = (X_1, ..., X_t)$ and $\Lambda_{1:t} = (\Lambda_1, ..., \Lambda_t)$ for partial histories. The joint density for partial histories is

$$p(X_{1:t}, \Lambda_{1:t}, \theta) = (2\pi)^{-M/2} \exp\left\{ -\frac{1}{2} g_t(X_{1:t}, \Lambda_{1:t}, \theta)' \left[\Sigma(X_{1:t}, \Lambda_{1:t}, \theta)\right]^{-1} g_t(X_{1:t}, \Lambda_{1:t}, \theta) \right\}, \quad (12)$$

which corresponds to (5). The densities $p(X_{1:t} \,|\, \Lambda_{1:t}, \theta)$ and $p(\theta \,|\, \Lambda_{1:t}, X_{1:t})$ are proportional to (12). For $p(X_{1:t} \,|\, \Lambda_{1:t}, \theta)$ the proportionality factor is assumed to be one; we do not need the proportionality factor for $p(\theta \,|\, \Lambda_{1:t}, X_{1:t})$ because we use a Metropolis algorithm to draw from it.

5.1 A Particle Filter

1. Initialization.

- Input θ (and X)
- Set T_0 to the minimum sample size required to compute $g_t(X_{1:t}, \Lambda_{1:t}, \theta)$.
- For $i = 1, \ldots, N$ sample $(\Lambda_1^{(i)}, \Lambda_2^{(i)}, \ldots, \Lambda_{T_0}^{(i)})$ from $p(\Lambda_t | \Lambda_{t-1}, \gamma)$.
- Set t to $T_0 + 1$.
- Set $\Lambda_{1:t-1}^{(i)} = (\Lambda_1^{(i)}, \Lambda_2^{(i)}, \ldots, \Lambda_{T_0}^{(i)})$

2. Importance sampling step.

- For $i = 1, \ldots, N$ sample $\tilde{\Lambda}_t^{(i)}$ from $p(\Lambda_t | \Lambda_{t-1}^{(i)})$ and set

$$\tilde{\Lambda}_{1:t}^{(i)} = (\Lambda_{0:t-1}^{(i)}, \tilde{\Lambda}_t^{(i)}).$$

- For $i = 1, \ldots, N$ compute weights $\tilde{w}_t^{(i)} = p(X_{1:t} \,|\, \tilde{\Lambda}_{1:t}^{(i)}, \theta)$.
- Scale the weights to sum to one.

3. Selection step.

   - For $i = 1, \ldots, N$ sample with replacement particles $\Lambda_{1:t}^{(i)}$ from the set $\{\tilde{\Lambda}_{1:t}^{(i)}\}$ according to the weights.

4. Repeat

   - If $t < T$, increment t and go to Importance Sampling Step;
   - else output $\left\{\Lambda_{1:T}^{(i)}\right\}_{i=1}^{N}$.

5.2 A Modified Particle Filter

1. Initialization.

   - Input $\Lambda_{1:T}^{(1)}$, θ (and X)
   - Set T_0 to the minimum sample size required to compute $g_t(X_{1:t}, \Lambda_{1:t}, \theta)$.
   - For $i = \mathbf{2}, \ldots, N$ sample $(\Lambda_1^{(i)}, \Lambda_2^{(i)}, \ldots, \Lambda_{T_0}^{(i)})$ from $p(\Lambda_t | \Lambda_{t-1}, \gamma)$.
   - Set t to $T_0 + 1$.
   - Set $\Lambda_{1:t-1}^{(i)} = (\Lambda_1^{(i)}, \Lambda_2^{(i)}, \ldots, \Lambda_{T_0}^{(i)})$

2. Importance sampling step.

   - For $i = \mathbf{2}, \ldots, N$ sample $\tilde{\Lambda}_t^{(i)}$ from $p(\Lambda_t | \Lambda_{t-1}^{(i)})$ and set

$$\tilde{\Lambda}_{1:t}^{(i)} = (\Lambda_{0:t-1}^{(i)}, \tilde{\Lambda}_t^{(i)}).$$

   - For $i = \mathbf{1}, \ldots, N$ compute weights $\tilde{w}_t^{(i)} = p(X_{1:t} \,|\, \tilde{\Lambda}_{1:t}^{(i)}, \theta)$.
   - Scale the weights to sum to one.

3. Selection step.

   - For $i = \mathbf{2}, \ldots, N$ sample with replacement particles $\Lambda_{1:t}^{(i)}$ from the set $\{\tilde{\Lambda}_{1:t}^{(i)}\}_{i=\mathbf{1}}^{N}$ according to the weights.

4. Repeat

- If $t < T$, increment t and go to Importance Sampling Step;
- else output the particle $\Lambda_{1:T}^{(N)}$.

5.3 A Metropolis Algorithm

To implement a Metropolis algorithm we require a proposal density for θ. A proposal density is a transition density of the form $T(\theta_{old}, \theta_{new})$ such as a move-one-at-a-time random walk. In the example of Section 6.1, we used the move-one-at-a-time random walk that uniformly selects an index k and then moves the element $\theta_{k,old}$ of θ_{old} to $\theta_{k,new}$ according to a normal with mean $\theta_{k,old}$ and variance σ_k, where σ_k is chosen by trial and error to achieve a rejection rate of about 50% in the Accept-Reject step of the algorithm that follows or at least about the same magnitude as the standard errors of the chain. For K below we set $K = 50$.

- Input: Λ, θ_{old} (and X)
- Propose: Draw θ_{prop} from $T(\theta_{old}, \theta)$
- Accept-Reject: Put $\theta^{(i)}$ to θ_{prop} with probability

$$\alpha = \min\left[1, \frac{p(X, \Lambda, \theta_{prop})T(\theta_{prop}, \theta_{old})}{p(X, \Lambda, \theta_{old})T(\theta_{old}, \theta_{prop})}\right]$$

  else put $\theta^{(i)}$ to θ_{old}.
- Repeat: If $i < K$ put $\theta_{old} = \theta^{(i)}$ and go to Propose; else output $\theta^{(K)}$.

6 A Stochastic Volatility Model

6.1 Model

Our second example is a stochastic volatility (SV) model:

$$\begin{aligned} X_t &= \rho X_{t-1} + \exp(\Lambda_t)\, u_t \\ \Lambda_t &= \phi \Lambda_{t-1} + \sigma e_t \\ e_t &\sim N(0,1) \\ u_t &\sim N(0,1) \end{aligned}$$

The true values of the parameters are

$$\theta_0 = (\rho_0, \phi_0, \sigma_0) = (0.9, 0.9, 0.5)$$

for the purpose of plotting the particle filter and

$$\theta_0 = (\rho_0, \phi_0, \sigma_0) = (0.25, 0.8, 0.1)$$

for illustrating estimation results. The reason for the difference is that the former generates plots that are easy to assess visually whereas the latter are more representative of, say, fits to daily S&P 500 closing prices.

The moment conditions used with this model are:

$$\begin{aligned}
g_1 &= (X_t - \rho X_{t-1})^2 - [\exp(\Lambda_t)]^2 && (13)\\
g_2 &= |X_t - \rho X_{t-1}||X_{t-1} - \rho X_{t-2}| - \left(\frac{2}{\pi}\right)^2 \exp(\Lambda_t)\exp(\Lambda_{t-1}) && (14)\\
&\vdots \\
g_{L+1} &= |X_t - \rho X_{t-1}||X_{t-L} - \rho X_{t-L-1}| - \left(\frac{2}{\pi}\right)^2 \exp(\Lambda_t)\exp(\Lambda_{t-L}) && (15)\\
g_{L+2} &= X_{t-1}(X_t - \rho X_{t-1}) && (16)\\
g_{L+3} &= \Lambda_{t-1}(\Lambda_t - \phi\Lambda_{t-1}) && (17)\\
g_{L+4} &= (\Lambda_t - \phi\Lambda_{t-1})^2 - \sigma^2 && (18)
\end{aligned}$$

Moment (16) identifies ρ independently of Λ_t; moments (13) through (16) overidentify Λ_t given ρ. Moment (17) identifies ϕ given Λ_t and moment (18) identifies σ given Λ_t.

Estimates of θ for the SV model are shown in Table 1 for three methods: Metropolis within Gibbs GMM with a Jacobian term, without a Jacobian term, and using the Flury and Shephard (2010) estimator. The Flury and Shephard estimator can be regarded as state-of-the-art. The MCMC chain generated using the method are draws from the exact posterior with a flat prior.

Applying the particle filter at the true value of θ and $N = 5000$, we obtain the estimate of Λ shown as a time series plot in Figure 1 and as a scatter plot in Figure 2 for the case when a Jacobian term is included and as Figures 3 and 4 when it is not. The plots for the

Flury and Shephard estimator are Figures 5 and 6. In the particle filter vernacular, the Metropolis within Gibbs GMM estimator is computed from a smooth whereas the Flury and Shephard estimator is computed from a filter; accordingly, the plots shown for the Metropolis within Gibbs GMM estimator are smooths whereas the plots shown of the Flury-Shephard estimator are filters.

Table 1. Parameter Estimates for the SV Model Instruments (13) through (18)

Parameter	True Value	Mean	Mode	Standard Error
		With Jacobian Term		
ρ	0.25	0.30488	0.30961	0.074778
ϕ	0.8	0.09153	0.94851	0.660790
σ	0.1	0.09023	0.06702	0.050229
		Without Jacobian		
ρ	0.25	0.30271	0.30939	0.076758
ϕ	0.8	0.15348	0.85765	0.643400
σ	0.1	0.11400	0.08435	0.070081
		Flury and Shephard Estimator		
ρ	0.25	0.30278	0.28555	0.059320
ϕ	0.8	0.17599	0.89189	0.509780
σ	0.1	0.09737	0.07839	0.064661

Data of length $T = 250$ was generated by simulating the model of Subsection 6.1 at the values shown in the column labeled true. In the first two panels the model was estimated by using the Metropolis within Gibbs methods described in Section 5 with a one-lag HAC weighting matrix using $N = 1000$ particles for Gibbs and $M = 50$ draws for Metropolis. In the third panel the estimator is the Bayesian estimator proposed by Flury and Shepard (2010) with a flat prior. It is a standard maximum likelihood particle filter estimator except that the seed changes every time a new θ is proposed with N increased as necessary to control the rejection rate of the MCMC chain. The columns labeled mean, mode, and standard deviation are the mean, mode, and standard deviations of a Metropolis within Gibbs chain of length $R = 9637$ for the first two panels and the same from an MCMC chain of length $R = 500000$ with a stride of 5 for the third.

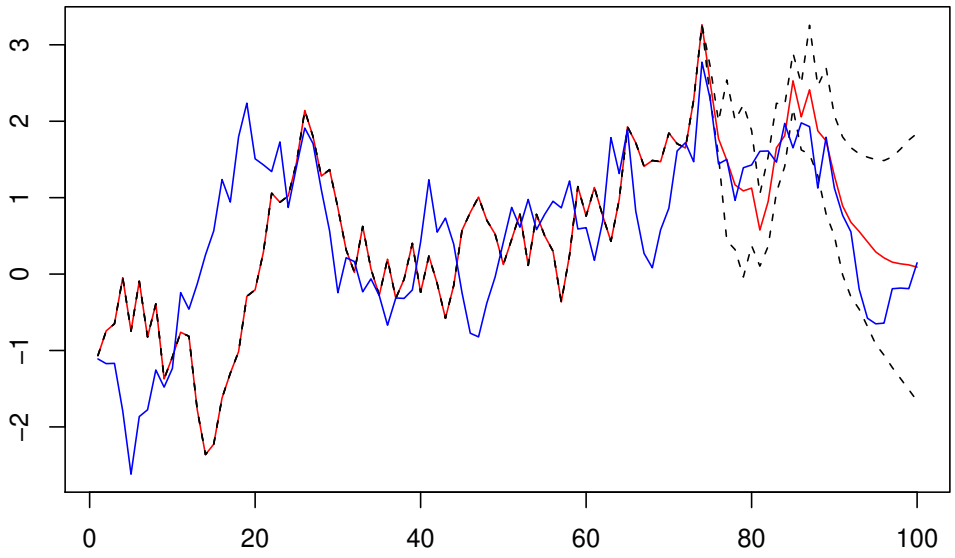

Figure 1. PF for Λ with Jacobian, Time Series Plot, SV Model. Data of length $T = 100$ was generated from a simulation of the model of Subsection 6.1 and $N = 5000$ particles computed using the algorithm described in Section 5.1 with a Jacobian term. The blue line plots the simulated Λ. The red line is the mean of the particles and the dashed black lines are plus and minus two pointwise standard errors. The moment equations were (13) through (18); a one lag HAC estimator was used for (3).

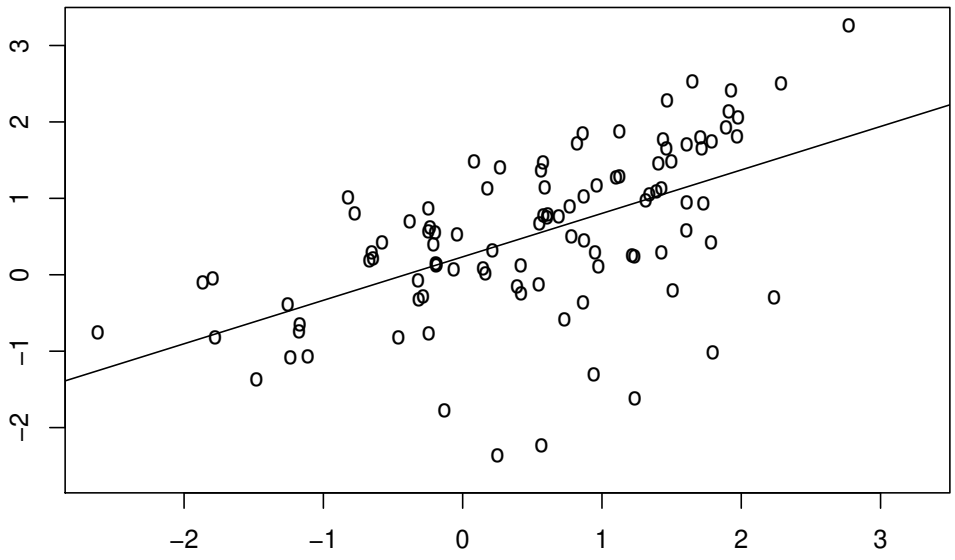

Figure 2. PF for Λ with Jacobian, Scatter Plot, SV Model. As for Figure 1 except that plotted is the mean of the particles vs. the simulated Λ.

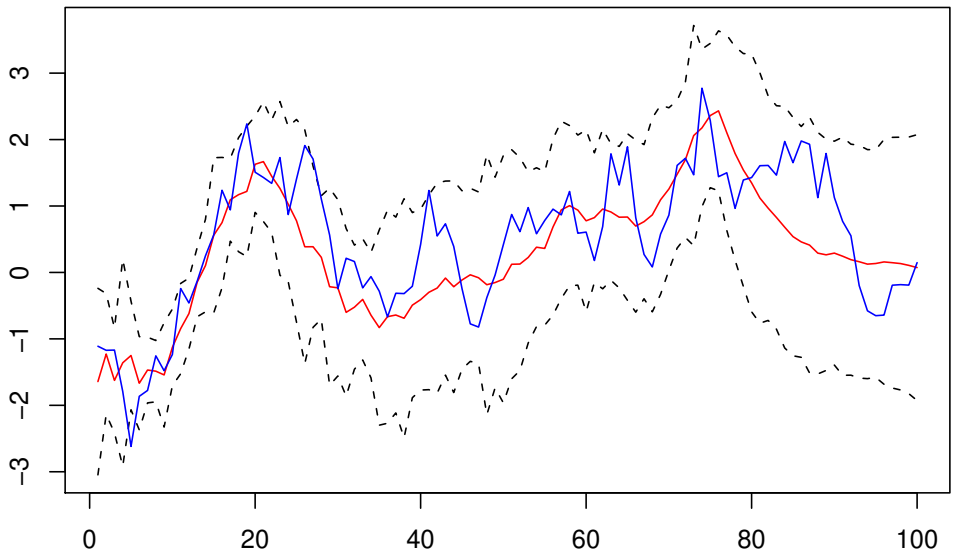

Figure 3. PF for Λ, without Jacobian, Time Series Plot. As for Figure 1 except that estimation is without a Jacobian term.

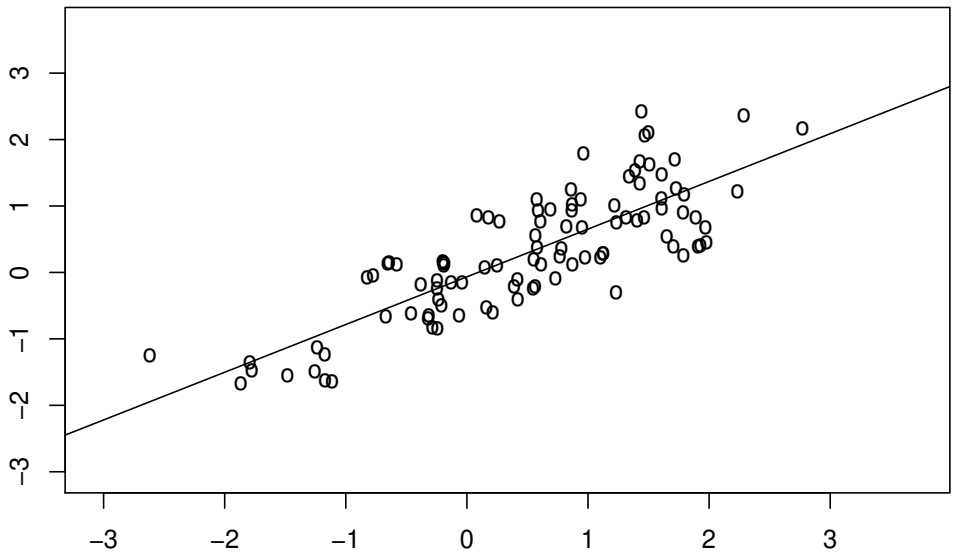

Figure 4. PF for Λ, without Jacobian, Scatter Plot, SV Model. As for Figure 3 except that plotted is the mean of the particles vs. the simulated Λ.

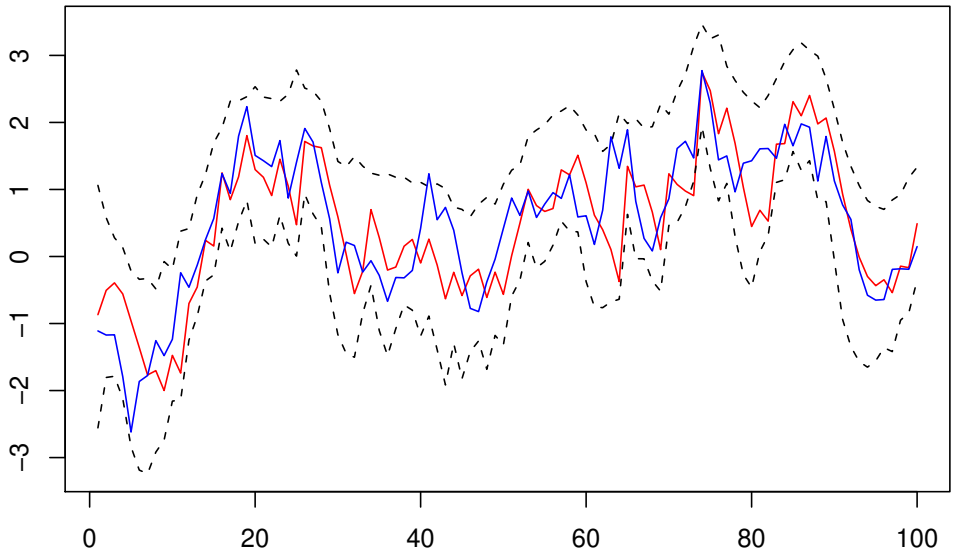

Figure 5. PE for Λ, Flurry-Shephard Method, Time Series Plot, SV Model. As for Figure 1 except that plotted is a filter, not a smooth, and weighting is by the measurement density, not GMM.

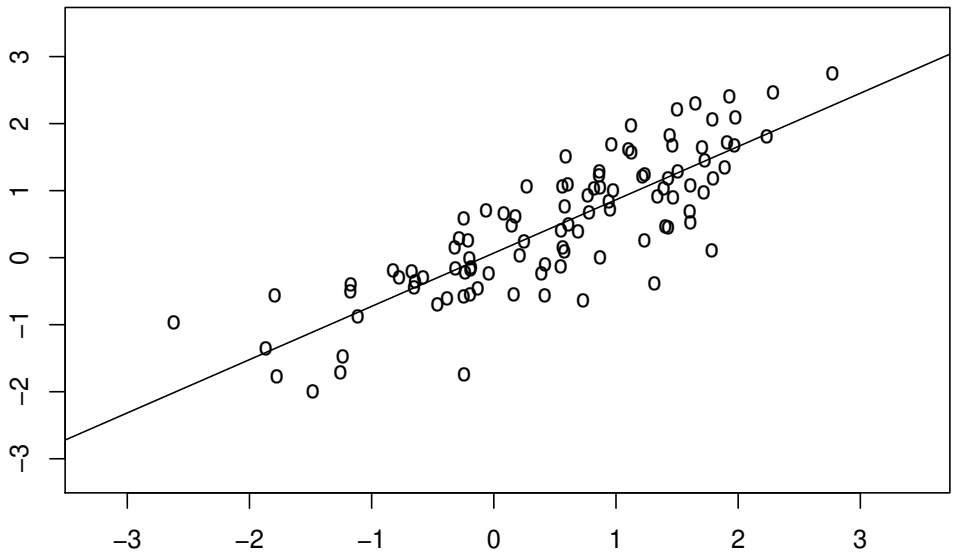

Figure 6. PF for Λ, Flurry-Shephard Method, Scatter Plot. As for Figure 5 except that plotted is the mean of the particles vs. the simulated Λ.

6.2 Code

The code implementing the model is in directory `svsim` of the distribution. One runs it by copying `makefile.gpp.lib` to `makefile` and then entering `mle.sh svsim.ctrl.dbg` on the command line. Entering `mle.sh svsim.ctrl.000` should reproduce the results shown in Subsection 6.1 but takes a long time to run, say about 72 hours. This is overkill, the tuning parameters in `svsim.parm.000` could be set to produce shorter runs without changing results much.

The particle filter is coded in `likelihood`. While `emcount` is less than `emlimit` MCMC draws are computed. When `emcount` exceeds `emlimit` the particle filter code is executed. The communication problem mentioned in Section 5 Introduction arises because likelihood is called after `mcmc` in `libmle` calls `set_rho` whereas the particle filter needs to start at the value at the preceding MCMC draw. This is where `set_rho_old` comes in. It supplies that value. This is also where the need for only one instance of `usrmod` arises: Only `mcmc` knows what `rho` and `rho_old` are so that all parts of the code must be working with the same instance of `usrmod` to guarantee these are set correctly.

The remainder of the code implements a likelihood based on a GMM criterion. It relies on `gmm` in `libscl`. The ideas behind the rest of the code can be determined by reading the portions of `libscl.h` relevant to class `gmm` and the documentation for `gmm` at the beginning of file `gmm.cpp`.

7 References

Andrieu, C., A. Douced, and R. Holenstein (2010), "Particle Markov Chain Monte Carlo Methods," *Journal of the Royal Statistical Society, Series B*, 72, 269–342.

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

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

Flury, Thomas, and Neil Shephard (2010), "Bayesian Inference Based Only on Simulated

Likelihood: Particle Filter Analysis of Dynamic Economic Models," *Econometric Theory*, forthcoming.

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

Gallant, A. Ronald, Raffaella Giacomini, and Giuseppe Ragusa (2013), "Generalized Method of Moments with Latent Variables," Working paper, http://www.aronaldg.org

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 (1990), "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 http://www.aronaldg.org.

Gallant, A. Ronald, and George Tauchen (1993), "EMM: A Program for Efficient Method of Moments Estimation, Version 2.6, User's Guide," Manuscript, Duke University. Available along with code and worked example at http://www.aronaldg.org.

Hansen, Lars Peter and Kenneth J. Singleton (1982), "Generalized Instrumental Variables Estimation of Nonlinear Rational Expectations Models," *Econometrica* 50, 1269–1286

Gallant, A. R., and H. White (1988), *A Unified Theory of Estimation and Inference for Nonlinear Dynamic Models.* Oxford: Basil Blackwell

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

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

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