User's Guide for the Bayes GMM estimator

June 26, 2017

This is a guide for the C++ program that implements the Bayesian GMM estimator of Gallant et al. (2017). The program is **heavily** relies on Gallant's MLE package.

Directory Structure

base_model: this folder contains the heart of the MCMC estimator. In principle, this part should be independent from the specific project.

initialize: reads the InputParamFile and defines the **specification class**

- *source files:* main.cpp, initialize.cpp
- *header files:* initialize.h

estimator: elements of the mcmc sampler/optimizer, generates an **mcmc class**

- *source files:* asymptotics.cpp, mcmc_class.cpp, proposal.cpp
- *header files:* estimator_base.h, estimator.h

libscl: slightly altered version of Gallant's statistical library (including the **gmm class**)

*****.example:** these foders belong to separate projects (indentified by the *** prefix).[1] Each project directory must contain three subfolders

usermodel: program codes that define the **usermodel class** (see XXX)

- *source files:* usermodel.cpp, moments.cpp, model.cpp, default_params.cpp
- *header files:* usermodel.h, moments.h, model.h, default_params.h

data: contains the data (in file `data.dat`) and the `initial_particle.dat` file containing an intial draw of particles for the conditional particle filter.

result_files: a plethora of .dat files generated by the estimator for diagnoses and further analyses

[1] In addition to the subfolders detailed below, they contain (1) the makefile that generates the executable (called `bayes_gmm`), (2) the InputParamFile detailing the specifics of the estimator (3) a python script generating summary statistics and plots from the result files.

1 Usage

Download `bayes_gmm.tar` from XXX. On a Unix machine use `tar -xf bayes_gmm.tar` to expand the tar archive into a directory that will be named `bayes_gmm`.

1.1 Sequential version

Change directory to `bayes_gmm/***_example/`,[2] and type `make` and the `bayes_gmm` executable will be built and ready to run.

This folder also contains a file called `control.dat`. This file contains the names of the InputParam Files (see later) and the prefix for the generated output files (denoted by `***`). Here is an example of a one line `control.dat` file:

```
test.param.000 test
```

The input paramfile is named `test.param.000` (must be in the same folder) and all output files such as `detail.dat`, `pi.000.dat`, etc. are named `test.detail.dat`, `test.pi.000.dat`, etc. in the corresponding `result_files` folder. The key result files are:

- `detail.dat`: Voluminous detailed output from the run.
- `summary.dat`: This file summarizes the output giving mean, mode, and standard errors
- `theta.000.dat`: contains the MCMC chain for θ
- `pi.000.dat`: Let $\ell(\theta) = \exp(-ns_n(\theta))$, and let $p(\theta)$ denote the prior. This file contains three items corresponding to the MCMC chain for θ : (1) $\log \ell(\theta) + \log p(\theta)$, (2) $\log \ell(\theta)$, (3) $\log p(\theta)$.
- `reject.000.dat`: this contains a matrix whose first column contains the rejection rate for each parameter followed by the overall rejection rate. ADD: other columns
- `paramfile.fit`: A copy of the InputParamFile with the parameter start values replaced by the mode and scaling variables recomputed so that `proposal_scale_factor` is 1.0. All else is the same as the InputParamFile.
- `theta_mode`, `theta_mode`, `V_hat_hess`, etc.: Statistics from the run in the form expected for reading with member `vecread` of class `scl::realmat` in library `libscl`.

1.2 Parallel version with OpenMPI

[2] Currently `***` stands for either `sv` (stochastic volatility model defined in Section 5.1. of Gallant et al. (2017)) or `sdf` (stochastic discount factor with latent state)

2 The InputParam File

The InputParam File (typically named as `***.param.000`) contains several blocks of control information. The structure is the same as the paramfiles for Gallant's EMM and MLE packages. The followings are mostly extracts from his MLE guide.

2.1 PARAMFILE HISTORY

This part is optional. It is written automatically by the program to the output paramfile called `***.paramfile.fit` (located in the `result_files` folder) 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[3] (e.g. describing the model) that begin with a # and these will get copied from the input parmfile to the output parmfile.

Example:

```
PARAMFILE HISTORY (optional)
#
# This paramfile was written by bayes_gmm 1.0 using the following line from
# control.dat, which was read as char*, char*
# ------------------------------------------------------------------------
#      test.param.000               test
# ------------------------------------------------------------------------
#
```

2.2 ESTIMATION DESCRIPTION

The 13 lines specify values for parameters of the estimator **(order matters!)**. Each line starts with 12 characters including the parameter value, then the following pieces of information separated by comma: (1) description, (2) variable name, (3) variable type.

`proposaltype:` Standard is the *group move* proposal which defaults to a single move proposal when the optional InputParamFile block PROPOSAL GROUPING is missing. When the PROPOSAL GROUPING block is missing, the `proposaltype=0` proposal randomly selects an element of θ 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.

[3] Specifically, the three paramfiles (`***.paramfile.fit`, `***.paramfile.alt`, `***.paramfile.end`) are being written by the function `initialize::specification_class::write_params()` which is defined in `initialize.cpp`.

`ask_print:` If the value is 1, then voluminous debugging information is written to file `***.detail.dat` in the `result_files.dat` subdirectory. Setting it ot 0 suppresses printing.

`iseed:` Seed for the MCMC chain.

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

`num_mcmc_files::` Determines how many files in addition to `theta.000.dat` are generated. The total length of the MCMC chain is `R= num_mcmc_draws*(num_mcmc_files + 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.

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

`temperature:` For Bayesian inference it is essential that temperature = 1! Otherwise, this variable controls the peakedness of the objective function (e.g. likelihood). 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.

`no_sandwich:` Computing sandwich standard errors is costly and often unnecessary, setting this variable to 1 will stop them from being computed. Even for an estimator that does require the computation of sandwich standard errors, one should set `no_sandwich = 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, `no_sandwich` can be set to 0.

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

`thin:` The program writes the MCMC chains to files of length `num_mcmc_draws` as explained above. If `thin=1`, every element of the chain is written. If `thin=2`, every other element is written and the length of an output files becomes `num_mcmc_draws/2`. Similarly for higher values of `thin`. Thin 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 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. 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). 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 `theta.000.dat`, `theta.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.

Example:

```
ESTIMATION DESCRIPTION (required)
        test   Project name, project_name, char*
         1.0   bayes_gmm version, version, float
           0   Proposal type, 0 group_move, 1 cond_move, 2 usr, proposaltype, int
           1   Write detailed output if ask_print=1, ask_print, int
  1741133992   Seed for MCMC draws, iseed, int
       10000   Number of MCMC draws per output file, num_mcmc_draws, int
           9   Number of MCMC output files beyond the first, num_mcmc_files, int
        10.0   Rescale prop scale block by this, proposal_scale_factor, float
         1.0   Rescale posterior by this val, temperature, float
           1   Sandwich variance not computed if no_sandwich=1, no_sandwich, int
           0   Number of lags in HAC middle of sandwich variance, lag_hac, int
           5   The thinning parameter used to write MCMC draws, thin, int
           0   Draw from prior if draw_from_prior=1, draw_from_prior, int
```

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.

`sample_size:` 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.

`datafilename:` The name of the file from which the data are to be read. The file must be located in the `data` directory.

`var_cols :` 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 variables (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. `var_cols` can be specified as a single digit or as a range. Thus, one can enter either `1 2 3 5` or `1:3 5`.

Example:

```
DATA DESCRIPTION (required) (model constructor sees realmat data(M,sample_size))
          12   Dimension of the data, M, int
         200   Number of observations, sample_size, int
data.dat       File name, any length, no embedded blanks, datafilename, string
1:12           Read these white space separated var_cols, var_cols, intvec
```

2.4 MODEL DESCRIPTION

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

`len_model_param:` The dimension of θ, which is the parameter vector of the model.

`len_model_func:` The dimension of `stats`, which is the vector of statistics (functionals) of the model that are computed from a simulation of the model. (**NOT USED**)

Example:

```
MODEL DESCRIPTION (required)
          26   Number of model parameters, len_model_param, int
           1   Number of model functionals, len_model_func, int
```

2.5 MODEL PARAMFILE

The vectors `model_paramfile_lines` and `model_addlines` of type `vector<string>` that are passed to the usermodel constructor (in `main.cpp`) are defined in the MODEL PARAMFILE block.

`model_paramfile:` This is the name of a file containing lines of the user's choosing. This file is read

and passed to the usermodel constructor as the `std::vector` of `std::string` `model_paramfile_lines`. If there is no such file then code `__none__` as the filename.

`#begin additional_lines`, `#end additional_lines:` Lines between these two markers are read and passed to the usermodel constructor as `model_addlines` of type `vector<string>`. The two marker lines are passed as well so that the first user line is `model_addlines[1]` and not `model_addlines[0]`.

Example:

```
MODEL PARAMFILE (required) (goes to usermodel as model_addlines)
__none__       File name, use __none__ if none, model_paramfile, string
#begin additional lines
           2   Number of observable risk factors, numb_obs_factor, int
           1   Lags for observable risk factors, lag_obs_factor, int
           8   Number of log returns, numb_returns, int
           1   Lags for HAC variance estimator (GMM objfun), lag_hac_gmm, int
         500   Number of particles, N, int
         100   Simulation size, len_simul, int
          50   Draws between particle filter updates, particle_update, int
#end additional lines
```

2.6 PARAMETER START VALUES

The block labeled PARAMETER START VALUES specifies the first value for the chain.[4] It must satisfy the support conditions; i.e. `usermodel_class::support` must return `true`, and `usermodel_class::prior` must return `scl::dev_val.positive = true` for this initial value of θ. 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 θ. To the right of this 0 or 1 the user may add text such as the name of the parameter.

Example:

```
PARAMETER START VALUES (required)
   1.90970554334998099e-01    1         1  A11
   3.24793056338963071e+00    1         2  A21
   3.60281749990822575e-02    1         3  A12
   8.48608238478818777e-02    1         4  A22
```

[4] New files `paramfile.fit`, `paramfile.end` and `paramfile.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 θ in the chain in the case of `.end`. The `paramfile.end` is used to recommence where one left off; `paramfile.fit` is used to recommence starting at the mode, which is what one usually wants to do; and `paramfile.alt` is used when switching to the conditional move proposal (`proposal_type=1`). If the number of parameters exceeds 20, then `paramfile.alt` will not be written. Once the mode has been found, it will not change.

2.7 PROPOSAL SCALING

These are the standard deviations of the proposal. Ideally, they should be roughly proportional to the standard errors of the estimate of θ if such is known. Altering their values is a way to affect the rejection probability.

Example:

```
PROPOSAL SCALING (required)
   3.12500000000000017e-03              1  A11
   3.12500000000000017e-03              2  A21
   3.12500000000000017e-03              3  A12
   3.12500000000000017e-03              4  A22
```

2.8 PROPOSAL GROUPING

How to specify group moves in the InputParam File is discussed in Subsection 6.3 of Gallant's EMM User's Guide. 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 an example (the synthesized version) in the file `***.detail.dat`, presuming `ask_print=1` in the ESTIMATION DESCRIPTION block.

3 How to define your own model

3.1 The `usermodel_class`

The class is declared in `usermodel/usermodel.h` and defined in `usermodel/usermodel.cpp`. Most importantly, it has three key linked private members: these are pointers to (1) a `moments` class, (2) a `scl::gmm` class, and (3) a `model` class[5]

`model` class :

- its likelihood member calculates the value of the GMM representation of the measurement density evaluated at the data y_t for given $t \leq$ `sample_size` and a given set of particles. For this step, it first `set_particle` and `set_sample_size`(t) of the `moments` class, then calls `scl::gmm` for the current `theta` value.
- it has two members that `draw_x0()` and `draw_xt(xlag)` (latent variable) and a wrapper `prop_yt(t, particle)` for the likelihood which are called by the particle filter

`moments` class : evaluates the moment conditions $g_t(\theta, y_t, x_t)$ (for a given period t) given `theta`, the `data`, and the `particle`. It inherits the `scl::moment_function_base` and provides a way to specify the moment conditions.

`scl::gmm` class :

- it returns $g_n' S_n^{-1} g_n$ where $g_n = \frac{1}{n} \sum_{T_{\min}}^{n} g_t(\theta, y_t, x_t)$ and $T_{\min} \leq n \leq$`sample_size` and S_n is
  - calculated from centered sample moments if `correct_for_mean==true`
  - if `regularize_W==true`, it is regularized with parameter`ridge` to make the inverse well conditioned
  - HAC corrected if `lag_hac_gmm>0`
- its `set_*()` methods are expected to update the `moments` class as well.

Moreover, corresponding to the particle filter, it has `scl::realmat` members

- `saved_particle`: used for the conditioning on the last trajectory in the conditioned PF
- `draws` (unweighted whole set), `filter` (reweighted using draws until t), `smooth` (reweighted using the whole set): $T \times N$ matrices to store the particles from the last updates. They are needed to calculate the mean path and standard deviations
- `gibbs_draws` separately collects the `theta` mcmc draws for periods when the `saved_particle` gets updated

[5] Defined in this order(!) using the previously defined objects, i.e. `scl::gmm` is defined using `moments` and `model` is defined using `moments` and `scl::gmm`.

As for the methods, there are three important ones

1. `usermodel.support(theta)`: boolean variable to determine whether the given `theta` is inside the support of the prior
2. `usermodel.prior(theta)`: calculates the log prior density evaluated at a specified θ. The default is a flat prior with $\log p = 0$.
3. `usermodel.likelihood()`: this is the **particle filter**
   - Before it is called, the `mcmc.draw()` function sets `theta_old` and `theta` (the proposed θ)
   - when it is called, the first thing to do is resetting `theta` for the `*moments` and `*model` class members and extract `sample_size` and `T0` from the `data` and `*moments` class (`get_Tmin`) members respectively
   - set the data and sample size for the `scl::gmm` class

     ```
     bool dset = gmm_objfun->set_data(&data);
     bool nset = gmm_objfun->set_sample_size(n+1);
     ```

   - calculates the likelihood of `theta`
     (a) until `counter < particle_update`
       – use `saved_particle` and `t=sample_size + 1` to call `*model.likelihood()`
       – increment `counter` and return `likelihood(theta_new)`
     (b) when `counter >= particle_update`, run the particle filter:
       – reset `theta_old` for `*moments` and `*models`
       – add `theta_old` to `gibbs_draws` and initiate `draws, smooth, filter` with the `saved_particle` (0th entry of the vector of `scl::realmats`)
       – fill the first $T_0 = T_{\min}$ elements with `*model.draw_x0` and `*model.draw_xt`
       – Importance sampling step uses `*model.prob_yt` (again `*model.likelihood`) for the weights (in the background, `scl::gmm` call with `theta_old` and `data`)
       – reset `saved_particle` with the last trajectory and return `likelihood(theta_new)` (i.e. with the old particles and new theta)

4 The `sdf` usermodel

Necessary parameters to be given

K or `numb_obs_factor`: Number of observable macro risk variables

L or `lag_obs_factor`: Number of lags for the observable macro risk variables

I or `numb_returns`: Number of returns used in the estimation

The `data.dat` file has the following format

- Let Y_t be the vector of **observable** risk factors in a demeaned format, i.e.

$$Y_t := \left[y_t^1 - \mu_y^1, y_t^2 - \mu_y^2, \ldots, y_t^K - \mu_y^K\right]' \qquad \mu_y^k := \frac{1}{T}\sum_{t=1}^{T} y_t^k, \quad \forall k \in \{1, \ldots, K\}$$

  These constitute the first KL columns.

- The remaining columns contain the log returns $\log R_t^i$ for various assets $i \in \mathcal{I}$.
- (?) Maybe after that could come the conditioning variables

The statistical model is the following

$$\begin{bmatrix} Y_{t+1} \\ X_{t+1} \end{bmatrix} = \begin{bmatrix} A_y & 0 \\ 0 & A_x \end{bmatrix} \begin{bmatrix} Y_t \\ X_t \end{bmatrix} + \begin{bmatrix} C_y & 0 \\ 0 & C_x \end{bmatrix} \varepsilon_{t+1}$$

$$\lambda_t = \lambda_0 + \begin{bmatrix} \lambda_y & \lambda_x \end{bmatrix} \begin{bmatrix} Y_t \\ X_t \end{bmatrix} \qquad \text{with} \quad \lambda_t \in \mathbb{R}^{\dim(\varepsilon)}$$

$$\log\left(\frac{S_{t+1}}{S_t}\right) = -\delta_0 - \begin{bmatrix} \delta_y & \delta_x \end{bmatrix} \begin{bmatrix} Y_t \\ X_t \end{bmatrix} - \frac{|\lambda_t|^2}{2} - \lambda_t \cdot \varepsilon_{t+1}$$

That implies the following moment conditions

$$\mathbf{0}_{K\times KL} = \mathbf{m}_1(y_{t+1}, y_t, \theta) = E\left[\left(Y_{t+1} - \sum_{l=1}^{L} A_{y,l} Y_{t+1-l}\right)\begin{bmatrix} Y_t' & Y_{t-1}' & \dots & Y_{t-L}' \end{bmatrix}\right]$$

$$\mathbf{0}_{K\times K} = \mathbf{m}_2(y_{t+1}, y_t, \theta) = E\left[\left(Y_{t+1} - \sum_{l=1}^{L} A_{y,l} Y_{t+1-l}\right)\left(Y_{t+1} - \sum_{l=1}^{L} A_{y,l} Y_{t+1-l}\right)'\right] - C_y C_y'$$

$$0 = \mathbf{m}_3(x_{t+1}, x_t, \theta) = E\left[\left(X_{t+1} - A_x X_t\right) X_t\right]$$

$$0 = \mathbf{m}_4(x_{t+1}, x_t, \theta) = E\left[\left(X_{t+1} - A_x X_t\right)^2\right] - {C_x}^2$$

$$0 = \mathbf{m}_5^i(z_{t+1}, z_t, x_{t+1}, x_t, \theta) = E\left[\exp\left(-\begin{bmatrix}\delta_0 & \delta_y & \delta_x\end{bmatrix}\begin{bmatrix}1\\ Y_t\\ X_t\end{bmatrix} - \frac{1}{2}\begin{bmatrix}1\\ Y_t\\ X_t\end{bmatrix}' \Lambda'\Lambda \begin{bmatrix}1\\ Y_t\\ X_t\end{bmatrix}\right.\right.$$
$$\left.\left. - \left(\Lambda\begin{bmatrix}1\\ Y_t\\ X_t\end{bmatrix}\right)' \begin{bmatrix} C_y & 0\\ 0 & C_x\end{bmatrix}^{-1}\begin{bmatrix}\left(Y_{t+1} - \sum_{l=1}^{L} A_{y,l} Y_{t+1-l}\right)\\ X_{t+1} - A_x X_t\end{bmatrix} + \log R_{t+1}^i\right) - 1\right]$$

where $\Lambda := \begin{bmatrix}\lambda_0 & \lambda_y & \lambda_x\end{bmatrix}$. If Y_t includes the risk-free interest rate, then $\delta_0 = \delta_y = \delta_{y,\neq r} = 0$.

The θ vector is of length $K^2L + \frac{K(K+1)}{2} + 2 + K(K+2) + (K+2)$ with

$$\theta := \begin{bmatrix}\text{vec}\left(A_{y,1}\right)' & \dots & \text{vec}\left(A_{y,L}\right)' & \text{vec}\left(C_y\right) & \rho & \sigma & \text{vec}\left(\Lambda\right)' & \delta_0 & \delta_y' & \delta_x\end{bmatrix}$$

The `base_model/initialize/main.cpp` file

1. The `main` is called with command arguments with possibly multiple rows, e.g. the file `control.dat` might contain

   ```
   svsim.param.000 svsim_0
   svsim.param.001 svsim_1
   ```

   and the program reads rows iteratively, at each round `argp[1]` becomes `paramfile`, `prefix`

2. Create **specification class** (called `specification`) and take the rows of `paramfile` to pass them into `specification.set_params()`. This initializes the private members of `specification_class`

   - the `(estblock, datablock, modelblock)` triple using the first three blocks of the InputParamFile (see above)
   - set the value of `theta` (and `theta_fixed`) using PARAMETER START VALUES block
   - set `proposal_scale` as a product of `estblock.proposal_scale_factor` and the PROPOSAL SCALING block
   - set `proposal_groups` from the PROPOSAL GROUP block if provided

3. Read data from the `data/` folder using `datablock.read_data()`

   - data file must contain columns for observables
   - columns must be separated by whitespace

4. Create **usermodel class** (of `usermodel_type`) named as `usermodel` with the arguments

   - just imported `data`
   - `model_blk.len_model_param` from the InputParamFile
   - `model_blk.len_model_func` from the InputParamFile
   - `specification.get_model_addlines()` from the InputParamFile

What does this do? (see model/usermodel/usermodel.cpp)

(a) read `initial_particle.dat` to set the value of `saved_particle` member of `usermodel`

(b) initialize `moment_cond=moments()`

- default: `data(0), particle(0), n(0), lag_gmm(0), theta(4, 1)`
- `theta` gets initialized from `default_params.h` default values

(c) initialize `gmm_objfun=gmm(moment_cond, lag_gmm, &data, data.ncol(), lag_hac_gmm)` with regularized W (ridge= $1.0e-3$)

- default: `mf(moment_cond), mfl(lag_gmm), data, sample_size(data.ncol()), Lhac(lag_hac_gmm), correct_for_mean(true), regularize_W(false), ridge(0.0), warning_messages(true)`
- reset `lag_gmm, data, n` for `moment_cond` from these (!!!)
- run `gmm_objfun->set_regularize_W(true, 1.0e-3)`

(d) initialize `model=model(moment_cond, gmm_objfun)`

- default: parameters (`default_params.h`), `moment_cond` and `gmm_objfun`
- run `gmm_objfun->set_moment_function(moment_cond)`, i.e.
  - set_lag_gmm(), set_data() and set_sample_size() for `moment_cond` from provided objects (!!!) (this is insurance, should be redundant)

(e) `lag_gmm, lag_hac_gmm, N, len_simul, particle_update` get read from `specification.get_model_addlines()`

(f) instantiate `draws, smooth` and `filter` of `usermodel` with length `N`

---

5. Create **proposal class** (of `proposal_base` type) named `proposal` using

   - `estblock.proposaltype` (`group_move` or `conditional_move`) and
   - `specification.get_proposal_groups()` defined above

6. Create the **mcmc class** using the `proposal` and `usermodel` classes

   - default: `proposal(proposal), usermodel(usermodel), simulation_size(1), thin(1), draw_from_posterior(true), temperature(1.0), posterior_mode(), posterior_maxval(-REAL_MAX)`
   - reset elements from the ESTIMATION DESCRIPTION block:
     - `simulation_size` becomes `len_mcmc_draws`
     - reset `thin, draw_from_posterior, temperature` from `est_blk`

7. Create **asymptotics class** (of `asymptotics_base` type) with args `data, usermodel, mcmc`

   - default: `data(data), T(data.get_cols()), mcmc(mcmc), len_theta(usermodel.get_len_theta()), theta_sum(len_theta, 1, 0.0), mean_old(len_theta, 1, 0.0), theta_sse(len_theta, len_theta, 0.0), mean(len_theta, 1, 0.0), posterior_mode(len_theta, 1, 0.0), cov(len_theta, len_theta, 0.0), foc(len_theta, 1, 0.0), cum_sample_size(0), posterior_maxval(-REAL_MAX)`

8. Define `realmat theta = specification.get_theta();` and `INT_32BIT seed = est_blk.seed;`

9. Loop `num_mcmc_files` many times over the sampler of size `num_mcmc_draws` starting at `theta`

[**A**] `mcmc.draw(seed, theta_start, theta_sim, stats_sim, pi_sim);`

- this function communicates with PF in `usermodel.likelihood()`
  - upon new proposal mcmc class tells usermodel the new and old theta's
  - it draws particle_update many proposals before a new set of particles
  - it returns the rejection probabilities as a table: rejection/total
- `realmat theta_old = theta_start;`

  ```
  usermodel.set_theta(theta_start);
  usermodel.set_theta_old(theta_old);
  usermodel.get_stats(stats_old);

  den_val likehood_old = usermodel.likelihood();
  den_val prior_old = usermodel.prior(theta_old, stats_old);
  den_val pi_old = prior_old;
  if (draw_from_posterior)  pi_old += likehood_old;
  posterior_mode = theta_start;
  posterior_maxval = pi_old.log_den;
  if (pi_old.positive) pi_old.log_den *= temperature;
  ```
- Define `*_new` variables from `*_old` ones
- `num_mcmc_draws` many times
  - `proposal.draw(jseed, theta_old, theta_new);`
    `usermodel.set_theta(theta_new);`
    `usermodel.set_theta_old(theta_old);`
  - `if (usermodel.support(theta_new) && usermodel.get_stats(stats_new))`
    - call `usermodel.likelihood()`
    - update `likelihood_new, prior_new, pi_new`
    - if `pi_new > high`, reset `posterior_mode` and `posterior_maxval`
    - calculate alpha from `pi_old` and `pi_new` and rejection/acceptance step
- last simulated value becomes `theta_start`, so we can use that again in the `num_mcmc_files` many iterations

[**B**] `asymptotics.set_asymptotics(theta_sim);`

- taking `theta_sum`, this updates asymptotics' `mean, posterior_mode, cov, cum_sample_size` and `posterior_maxval`

[**C**] `asymptotics.get_asymptotics(theta_hat, V_hat, T);`

- assigns `mean, cov/T, T` to arguments

[**C**] `asymptotics.get_asymptotics(theta_mean, theta_mode, posterior_high, I, invJ, foc_hat, reps);`

- assigns `mean, posterior_mode, posterior_maxval, I = null; invJ = cov; foc_hat = null; reps = 0;` to arguments

[**D**] `usermodel.set_theta(theta_mode)`: reset usermodel's `theta` with the new mode

[**E**] `specification.write_params(paramfile, prefix, seed, theta, theta_mode, invJ/T)`

```
usermodel.set_theta(theta_mode);
filename = "../result_files/" + prefix + ".usrvar";
usermodel.write_usrvar(filename.c_str());
```

[**F**] call `output` function

- `theta_mean -> prefix + ".theta_mean.dat";`
- `theta_mode -> prefix + ".theta_mode.dat";`
- `V_hat_hess -> prefix + ".V_hat_hess.dat";`
- `theta_mean, theta_mode, V_hat_hess -> prefix + ".summary.dat";`
- `theta_sim -> prefix + ".theta." + number + ".dat";`
- `stats_sim -> prefix + ".stats." + number + ".dat";`
- `pi_sim -> prefix + ".pi." + number + ".dat";`
- `reject -> prefix + ".reject." + number + ".dat";`

TODO

Notes

- Both specification class and usermodel class have theta member
  - During the mcmc sampling: only the usermodel's theta private member gets updated (both theta and theta_old) and while it calls usermodel.likelihood(), usermodel automatically updates *_model and *_moments classes as well
  - The specification class's theta on the other hand does not get updated, so it is always stays at the start values of theta (could rename actually)
- Both mcmc class and asymptotics class have posterior_mode and posterior_maxval members
  - mcmc versions get updated just before the acceptance/rejection step
  - temperature for posterior affects the acceptance/rejection step
  - after `num_mcmc_draws` many draws the mcmc versions are passed to asymptotics => `theta_mode` and `posterior_high` get updated in asymptotics along with the cumulative mean theta and cov => everything gets written into summary.dat files

References

A. Ronald Gallant, Raffaella Giacomini, and Giuseppe Ragusa. Bayesian estimation of state space models using moment conditions. *Journal of Econometrics*, forthcoming, 2017.