MCE SOLVE Examples

March 2014

This note describes the example programs that are distributed with the *mce_solve* code for solving linear and nonlinear rational expectations models in EViews. The examples illustrate how to set up a model for use by the solution code and how to run various types of simulations. The examples all use a very simple three-equation model. Inflation depends on expected inflation next quarter, actual inflation last quarter, and the output gap. The output gap also depends on its first lead and lag, and on the real short-term rate of interest. The nominal short-term rate of interest is determined by an inertial form of a Taylor-type rule. The model contains two unique future-dated variables: the inflation lead, which appears twice; and the output gap lead, which appears once. The model is structured so that in equilibrium inflation, the output gap, and the rate of interest are all zero. The example programs initialize all data to zero.

Much of the material presented below is described in more detail in the *MCE Solve Users Guide.*

The *mce_solve* algorithms are extended path methods and thus require that the simulation interval be long enough that extending the terminal date by an additional quarter has no effect on the simulated outcomes over the period of interest. The algorithms also require that all variables that appear as leads have defined observations (terminal values) in the quarters after the end of the simulation interval. For example, a variable with a second lead requires terminal values in the first two quarters after the end of the simulation period. Finally, because the algorithms impose perfect foresight of future outcomes, all future-dated shocks are fully anticipated at the start of a simulation. The analysis of unanticipated shocks requires a sequence of simulations.

The *mce_solve* algorithms are coded as a set of subroutines, and thus they can be executed only as part of a program. Each example program loads the subroutines with the command

```
include mce_solve_library
```

A path to *mce_solve_library* may also be needed. Among the more than 20 subroutines in *mce_solve_library*, the one named *mce_run* provides the interface for designing and executing a simulation. The *mce_run* subroutine takes three arguments, each of which is a string.

```
%mopts = " …. "
%aopts = " …. "
%sopts = " …. "
call mce_run(%mopts,%aopts,%sopts)
```

The first string *(%mopts*) provides the model setup instructions needed by the *mce_solve* algorithms; the second string *(%aopts*) chooses an algorithm; and the third string (%sopts) defines the type of simulation.

%mopts

In the following discussion, let Model O contain *n* distinct future values of endogenous variables (ie, expectations variables). In order to use the *mce_solve* algorithms to simulate this model, two separate operational models must first be created. One, Model B, is formed from Model O by replacing all leads of endogenous variables with expectations proxies, which can be either new exogenous variables or new endogenous variables; the proxies must have current (or past) dates. The other model, Model F, must

contain for each of the *n* expectations variables an equation that defines its MC expectations error, that is, the gap between the value of the expectations proxy and the future-dated outcome. Depending on the contents of the *%mopts* string, Model B and Model F can be constructed automatically from Model O or the user can supply the names of two operational models that have already been created.

Program *example1* illustrates the syntax for the automated case, which constructs Model B by replacing all leads of endogenous variables in Model O with new exogenous variables and assigns to Model F an MC expectations error equation for each lead in Model O. The *%mopts* string in this program,

```
%mopts = "create,mod=%mod,adds,track"
```

uses the *create* keyword to initiate code that parses the model whose name the program has previously placed in the string *%mod* and then constructs the two required operational models. The *adds* keyword causes add factors to be assigned to all equations in the two models, and the *track* keyword causes the add factors to be given tracking values for the sample period in effect when the call to *mce_run* is made. (The *adds* and *track* keywords are in fact unnecessary in this particular example, as all equations hold exactly for the baseline data in which all variables are zero, but this will not be true in general.)

Before turning to an example in which the two operational models are set up manually, it's worth noting the main reason why the automated approach is not always preferable: The user does not have access to the operational models in the automated case between the time they are created and the time the simulation is run, because the setup steps and the simulation are all executed with a single call to *mce_run*. For this reason, if tracking add factors are desired in the automated approach, they have to be introduced via keywords in the *%mopts* string, as noted above. In addition, the automated approach also makes it difficult to define before the call to *mce_run* any shocks that are to be included in the simulation, because the shocked variables may not yet exist (if they are add factors), or if they do exist, modifications to the variables will be fully offset if tracking adds are computed automatically. In *example1*, the commands that define the shock to be simulated are placed in a text file. The *txt* keyword in the *%sopts* string causes the commands to be executed during the call to *mce_run* after tracking adds have been computed and before the simulation is run.

Program *example2* runs the same simulation with the same model used in *example1* but constructs the two operational models manually. In the *%mopts* string in *example2*,

```
%mopts = "mod_b=%modb,mod_f=%modf,mce_instrus=%instrus,mce_errs=%errs,adds,track"
```

the keyword *mod_b* points to the name of the operational Model B; *mod_f* to the name of the operational Model F; *mce_instrus* to the names of the exogenous expectations proxies in Model B; and *mce_errs* to the names of the MC expectations errors in Model F. The number of expectations proxies and expectations errors must be the same. See the program for information on how *%modb*, *%modf*, *%instrus*, and *%errs* are defined.

In program *example3*, which also sets up the two operational models manually, the expectations proxies in Model B are not exogenous variables, as is the case in *example1* and *example2*, but rather they are endogenous variables with their own equations. The equation for the expected inflation proxy is the four-quarter average of lagged inflation; the equation for the proxy for expected output has a similar structure. Given this design, Model B can be thought of as a complete system in which expectations have a particular autoregressive form. A consequence of shifting from exogenous to endogenous expectations proxies is that the instruments whose values are adjusted to impose the MC conditions must be the add factors on the equations for the endogenous expectations proxies.

%aopts

The choice of an MC solution algorithm and its various options is made using the second or *%aopts* argument of the *mce_run* subroutine. The *meth* keyword specifies the algorithm: *meth=newton* for the Newton algorithm; *meth=qnewton* for a limited-memory implementation of Broyden's quasi-Newton method. The default is *newton*, which is usually the faster of the two for single simulations of linear models of small-to-medium size. For nonlinear models, *newton* tends to be penalized relative to *qnewton* to the extent that the nonlinearity eliminates patterns that can be exploited in computing the Newton expectations Jacobian. The *newton* algorithm has a substantial advantage over *qnewton* on experiments that involve a large number of MCE solutions, as long as the same expectations Jacobian can be used for each *newton* solution.

Associated with the *newton* algorithm are various options for the computation of the expectations Jacobian. Both *example1* and *example2* are able to use the *jinit=linear* option, whose operation is very efficient at computing the exact Jacobian of a linear model in which the maximum endogenous lead and lag is one period, as is the case in these examples. The model in *example3* does not satisfy these conditions, because the expectations proxies depend on four lagged values. An efficient choice in this example is the approximate Jacobian calculated with the *jinit=interp(4)* setting. This example also illustrates the greater flexibility that the manual creation of the two operational models permits in the assignment and initialization of add factors, the declaration of scenarios, and the specification of shocks.

Another feature of the third example is that it runs three simulations and demonstrates in the second and third simulations how the assignment of null strings to the first two arguments of the *mce_run* causes the simulations to be run with the same operational models and algorithm (including the Newton MCE Jacobian) that were created or declared in the first simulation.

The first three examples run simulations of a linear model. *Example4* introduces nonlinearity by modifying the model in *example3* so that a lower bound on the short-term rate of interest is imposed. Given that the baseline data is set to zero, the lower bound on the interest rate is set at -1.0. The simulation is a negative output shock that is sufficiently large to make the lower bound bind for several years. The example sets *meth=qnewton* because, as noted above, nonlinearity tends to favor the *qnewton* algorithm over the *newton* algorithm, although the speed advantage of *qnewton* turns out to be slight in this case, because the model is relatively small. This example uses a kinked function to impose the interest-rate lower bound. The discontinuous first derivative of this type of function can be the source of convergence problems, though this is not the case in the example.

%sopts

The third or *%sopts* argument of the *mce_run* subroutine declares the type of simulation to be run and sets related options. In addition to the *single* type that the first four examples illustrate, two optimization simulation types are available: *opt* and *opttc*.

When the *opt* simulation type is chosen, *mce_run* invokes an iterative procedure that attempts to find the time paths of one or more exogenous instrument variables that minimize a loss function. When used to find the optimal path of a policy instrument, the solution is one of *commitment*. The algorithm usually requires many individual MCE solutions to find the optimal solution. For models that are not too nonlinear, the *newton* algorithm is likely to be execute much more quickly than the *qnewton* algorithm in this case.

The *opttc* simulation type is used when the optimizing agent cannot commit to a fixed instrument trajectory. In this case, the algorithm attempts to find the Nash time-consistent solution using a backward-induction approach. Note that the ``opttc'' algorithm provides only an approximate solution.[1]

Program *example5* illustrates how to set up and execute both types of optimization simulations when the path of the short-term rate of interest is chosen to minimize a loss function. The policy instrument is an adjustment factor to the policy rule equation. For the commitment (*opt*) case, the program makes use of the default settings in which the interest rate is chosen optimally over the first 40 simulation periods to minimize the loss function when it is evaluated over the first 60 periods. The evaluation period is chosen to be longer than the instrument period to minimize any discontinuities in the interest rate that may arise when the end of the instrument and evaluation periods coincide. For the time-consistent (*opttc*) case, the procedure assumes that each of 40 policymakers (one for each of the first 40 simulation periods) adjusts his single-period instrument to minimize a loss function that extends 60 periods from the date of their instrument.

[1] A first source of divergence between the true time-consistent solution and the computed solution concerns the assumption that, at each iteration, one policymaker optimizes and the other policymakers hold their policy instruments fixed. This assumption is correct only when the non-optimizing policymakers' instruments are adjustment factors to the model's true time-consistent policy rule. The *opttc* simulation procedure is designed for use with models for which it is difficult or impossible to compute the optimal time-consistent rule. (Testing the sensitivity of the *opttc* solution to variations in the assumed policy rule may be prudent.) In addition, for nonlinear models the solution is based on the linearized relationship between the instruments and targets faced by the first policymaker.