User's Manual for GAUSS and Matlab/Octave Software Presented in "Solution of Macromodels with Hansen-Sargent Robust Policies: Some Extensions"

Paolo Giordani* Paul Söderlind†

April 2003

1 Introduction

The purpose of this short manual is to minimize the reader's effort in working with the GAUSS and Matlab (or Octave) procedures used in the paper "Solution of Macromodels with Hansen-Sargent Robust Policies: Some Extensions," by Giordani and Söderlind. We assume that the reader is familiar with the concepts and the notation introduced in the paper.

The procedures can be download from http://home.tiscalinet.ch/paulsoderlind/, together with some test programs. You may want to use part of these programs as templates for your own programs once you understand the procedures.

2 Installing the Procedures

For GAUSS: put all the files in a folder where you will (eventually) run your programs. Also download the procs and dll files for the Schur decompositions: you find them (and installation instructions) at http://home.tiscalinet.ch/paulsoderlind/. The dll files should be in c:\gauss\dlib (assuming the Gauss is installed in c:\gauss), and PStgSen.prc and PsGees.prc should be in the same folder as the file where you have the other programs. When you run the program, make sure you are in the right directory (the one in which you downloaded the files). You can do this with the command: chdir directory_path.

For Matlab or Octave[1]: put all the m files in a folder and run your own programs from that same folder. Make sure to start Octave in MatLab compatibility mode (that is, start as octave –traditional).

3 Test Programs

The next section explains the structure of these programs in more detail. This section is simply intended to verify that the procedures you installed are working correctly.

Open the file *test1.prg/m*. This program solves for a simple backward-looking model, due to Svensson (1997).

*University of New South Wales. p.giordani@unsw.edu.au.
†University of St. Gallen and CEPR. Paul.Soderlind@unisg.ch.
[1]We thank Paul Klein for providing the files qzswitch.m and reorder.m.

If you are using GAUSS, the program #INCLUDEs files with the procedures used later on.

Run the program. You should obtain the following output: a figure, plotting the response of output, inflation, and the interest rate to an aggregate demand shock, for the worst case solution and for the approximating solution (as you increase θ the two will converge to the standard RE solution). Set θ to one or two and run the program again. Now the planner is absurdly pessimist, and reacts so forcefully that the impulse responses diverge under the approximating model (you are also warned that "Some abs(eigenvalue) of Ma > 1."

Open the file *test2.prg/m*. This program solves for the commitment equilibrium in the forward-looking model used in our first example in the paper. The output consists of impulse responses following a cost-push shock (standard RE solution, approximating solution, worst case solution), of error detection probabilities and values of the loss function in the worst case model. Now set θ to 20. You should receive an error message "Too few stable roots. No stable solution". The evil agent's budget is large enough to make the economy unstable.

Open the file *test3.prg/m*. This program solves the forward-looking New Keynesian model used in the paper. The output consists of two pictures: the first is the figure in the paper, the second is the detection error probability for this model. Now change the value of the variable "discretion" to any number other than one. The output will be the same as before, but this time the algorithm has solved for the simple rule (a standard Taylor rule) rather than for the discretionary case. Notice that a higher θ value is needed to get a stable solution with the simple rule.

4 General Structure of a Program

The core structure of a program to solve for a robust policy is exemplified in the test programs test1.prg/m, test2.prg/m, and test3.prg/m. This structure is made up of the following steps:

1. If you use GAUSS: activate the dll libraries and import the procedures used in the program. If you are solving for the commitment solution you need to import PsDtgSen (through the command: dlibrary PsDtgSen;), for the simple rule you need PsdGees (command dlibrary PsdGees;). Then, import the procedures you need. In fact, while not all the procedures are needed all the time, the easiest thing to do is to import them all at the beginning of every program (you can do this by copy and paste from either one of the test programs). You don't need to do this if you use Matlab, provided you run the program from the folder that contains the functions.

2. Prepare the inputs required by all procedures. These are the matrices and scalars defining the approximating model (A, B, C, $n1$, $n2$), the loss function (β, Q, R, U), and θ. In our test programs we do this by constructing another procedure (called "Svensson" in test1.prg/m and "MonPol3_1" in test2.prg/m and test3.prg/m).

3. Solve the program by using the appropriate procedure (backward-looking model, commitment, discretion, simple rule).

4. Use the solution to compute any object of interest, such as impulse response functions, asset prices, or whatever.

5. There is a special function for calculating detection error probabilities.

4.1 Backward-Looking Model

Refer to test1.prg/m and to the comments in the proc ComAlgSR (in MonRob.prc for GAUSS users).

To solve for a backward-looking model, use the proc ComAlgSR, which solves the case when there are no forward-looking variables. This proc, like all the others, can also deal with eigenvalues on the unit circle: just set $cutoff$ to a number slightly higher than one.

Usage:

```
{M,N,Ma,Na,Fv,J0,J0a} = ComAlgSR(A,B,Q,R,U,beta,cutoff,C,theta)
```

(In Matlab/Octave, use [] instead of {})

Input: refer to the paper and to the proc. If the system is stable, set $cutoff = 1$. If the system has one or more unit roots, set cutoff to a number slightly higher (only slightly, or the procedure may mistake some shadow prices for predetermined variables).

Output: refer to the appendix of the paper and to the comments in the proc.

For a backward-looking model, the evolution of the predetermined variables under the worst case model is given by

$$x_{t+1} = Mx_t + C\epsilon_{t+1},$$

and by

$$x_{t+1} = M_a x_t + C\epsilon_{t+1},$$

under the approximating model. Thus $M = A - BF_u - CF_v$, and $M_a = A - BF_u$. The vector u_t is equal to the first k rows of Nx_t (which are the same as the first k rows of $N_a x_t$) where k is the number of instruments of the policy maker. F_v is the evil agent's policy function. J0 is the loss function under the worst case model, and J0a is the loss function under the approximating model.

The proc ComAlgS solves for a backward-looking system in the standard (no robustness case). The same solution can be calculated with ComAlgSR by setting θ to a large number.

Pure forecasting is a special case, corresponding to $B = 0$. For the proc to work, you may need to set R to a positive number (this will not change anything of course, since there is no actual instrument).

4.2 Commitment Case

Refer to test2.prg/m and to the comments in the proc ComAlgR (in MonRob.prc for GAUSS users).

To solve for the commitment case in forward-looking models, use the proc ComAlgR.

Usage:

```
{M,N,Ma,Na,Fv,J0,J0a} = ComAlgR(A,B,Q,R,U,beta,n1,n2,cutoff,C,theta)
```

The inputs and outputs are the same as for the ComAlgSR, except that $n2$ is the number of forward-looking variables—which are assumed to show up in the last n_2 rows of the system (in A, B, etc). Notice that C is $(n_1 + n_2)\times\ n_2$ and that the last n_2 rows of C are all zero (the program will return an error message if you set an element of the last n_2 rows of C to a non-zero value).

The dynamics of the worst case solution is of the form

$$[x_{1t+1}, p_{2t+1}] = M[x_{1t}, p_{2t}] + C\epsilon_{t+1},$$

where p_{2t} is an $n_1 \times 1$ vector of Lagrange multiplier. In the approximating solution, the dynamics is

$$[x_{1t+1}, p_{2t+1}] = M_a[x_{1t}, p_{2t}] + C\epsilon_{t+1}.$$

The first $n2$ rows of Nx_{1t} define x_{2t}, and the $n2+1$ to $n2+k$ rows define u_t (refer to the appendix of the paper), so in the worst case solution:

$$[x_{2t}, u_t, v_{t+1}, p_{1t}] = N[x_{1t}, p_{2t}].$$

The same equation holds for the approximating solution, but with N_a instead of N.

The proc ComItAlg solves the standard (no robustness case). The same solution can be calculated with ComAlgR by setting θ to a large number.

4.3 Discretionary Case

Refer to test3.prg/m and to the comments in the proc DiscAlgR (in MonRob.prc for GAUSS users).

To solve for the discretionary case in forward-looking models, call on the proc DiscAlgR.

Usage:

```
{M,N,Ma,Na,Fu,Fv,J0,J0a} = DiscAlgR(A,B,Q,R,U,bet,n1,n2,
Vt1,Ct1,ConvCrit,Vweight,Fweight,CritLags,step,PrintIt,MaxIter,C,theta)
```

Inputs: A, B, Q, R, U, beta, n1, n2, C, and theta are as above. Vt1 and Ct1 are initial guesses of the value function and the mapping from the state vector to expectations, respectively. ConvCrit, Vweight, Fweight, CritLags, step, PrintIt, and MaxIter are used to control the iterations. See the proc for more detailed information.

In the approximating solution, the dynamics is

$$\begin{aligned} x_{1t+1} &= M_a x_{1t} + C_1 \epsilon_{t+1} \\ x_{2t} &= N x_{1t} = N_a x_{1t} \\ u_t &= -F_u x_{1t} \\ v_{t+1} &= -F_v x_{1t}, \end{aligned}$$

where C_1 is the matrix of the first n_1 rows of C. Notice that F_u relates the instruments to the predetermined variables only (this will be different in the simple rule case). The same equations hold for the worst case solution, but with M instead of M_a.

If you set θ too low (say 100) the algorithm will probably not converge. Notice that θ is not directly comparable across the equilibria (commitment, discretion, and simple rules)—only the detection error probabilities are.

4.4 Simple Rule

Refer to test3.prg/m and to the comments in the proc SimAlgR (in MonRob.prc for GAUSS users).

The simple rule case is slightly more complicated because it is, in fact, an optimal simple rule from the evil agent's part. We recommend to proceed in three steps:

1. `{M,N,Ma,Na,J0,J0a} = SimpAlgR(A,B,Q,R,U,beta,n1,n2,Fu,x10,cutoff,C,theta,Fv,J0aIt)` where x10 is an n_1 vector of initial values of the predetermined variables. A natural choice is their unconditional mean (typically zero). Fu is the planner's rule: $u_t = -F_u x_t$. Notice that F_u has $n1 + n2$ columns. Fv is an intial guess of the evil agent policy function, which you can set to an $n_1 \times (n_1 + n_2)$ matrix or zeros. The last n_2 columns of Fv must be zeros (you will get an error

message otherwise). If J0alt is set to 1, J0a is computed, otherwise it is not (to save some time). This step is strictly speaking not necessary, but it provides a check (does is work or not?) on the coding and parameter choices (theta, in particular).

2. Set Fv0 (another initial guess: can be an $n_1 \times (n_1 + n_2)$ matrix or zeros. Then, let the evil agent maximize the loss functionby using
   in Gauss:
   `{M,N,Ma,Na,J0,J0a,Fv} = SimpAlgROpt(Fv0)`
   Notice that A, B, Q, R, U, bet, n1, n2, Fu, cutoff, C, and theta must be defined (with these names) in the main program (here test3.prg) because they are used inside the proc;
   in MatLab/Octave:
   `[M,N,Ma,Na,J0,J0a,FvX] = SimpAlgROpt(Fv0,A,B,Q,R,U,bet,n1,n2,Fu,cutoff,C,theta)`
   Notice that MatLab/Octave allows us to pass extra arguments to optimization routines, which we use to avoid having any global variables.

3. Finally, `{M,N,Ma,Na,J0,J0a} = SimpAlgR(A,B,Q,R,U,beta,n1,n2,Fu,x10,cutoff,C,theta,FvX,J0aIt)`. This step takes the optimal policy rule of the evil agent (FvX) and calculates the equilibrium.

In the approximating solution, the dynamics is

$$x_{1t+1} = M_a x_{1t} + C_1 \epsilon_{t+1}$$
$$x_{2t} = N x_{1t}. = N_a x_{1t}.$$

The same equations hold for the worst case solution, but with M instead of M_a.

If you set θ too low (say 100) you will get an error message warning you that there are too few stable roots. This means that the evil agent is making the system unstable.

5 Detection Error Probabilities

Refer to test2.prg/m, test3.prg/m, and to the comments in the proc ErrDetProc (in MonRob.prc for GAUSS users).

The purpose of this procedure is to find the detection error probabilities corresponding to a sequence of different values of θ. This, in turn, provides guidance on choosing θ. Following Hansen and Sargent, we use $\sigma = -1/\theta$ rather than θ. First, set up a vector of values of σ for which the proc will compute corresponding detection error probabilities. If your range of values for σ includes at least a value for which the system is unstable, you will get an error message and you will need to increase the minimum θ in your grid. Some trial and error will be necessary in most cases, but if you start with high values of θ and then move down you should get to the relevant range rather quickly. Values of σ are on the x axis. Detection error probabilities are on the y axis. $\theta = \infty$ will produce a probability of 0.5.

Usage:

```
[rho,JcM] = ErrDetProb(A,B,Q,R,U,bet,Fu,n1,n2,cutoff,C,
T,NSim,sigma,randnstate,ModelType,Fv0,MaxIter,constant)
```

Inputs: most of the inputs are the same as in the previous procedures. The new inputs are as follows:

- T is the number of observations in the sample (those that agents can use to try to distinguish the two models).

- Nsim is the number of simulations. This should be set to a large number, like 10000. The procedure is fast, so this is perfectly feasible on a PC. However, much smaller numbers give sufficiently good approximations for some purposes. For example, you may set NSim at, 100 if you are just searching for the relevant range of values of σ, and then move to higher number as you refine the grid.
- sigma is a column vector of values of $\sigma = -1/\theta$
- randstate is a scalar. Sets the seed for the random number generator, allowing results to be reproduced exactly.
- ModelType model is a scalar, which takes value 1 for a backward-looking model, 2 for the commitment solution, 3 for the discretionary solution, 4 for the simple rule solution.
- Fu, and Fv0 are needed only in the simple rule case (see section on simple rules). In the other cases they can be set to misising values (use, e.g., NaN=miss(1,1); Fu=NaN; Fv=NaN).

Output: rho is the vector of probabilities associated with sigma

JcM is a matrix with as many rows as sigma and two columns. The first (second) column gives values of the average loss function under the worst case (approximating) model for the corresponding σ.