This version: January 2023.

This Matlab code replicates the results in all the Tables in Qu and Tkachenko (2023): "**Using Arbitrary Precision Arithmetic to Sharpen Identification Analysis for DSGE Models**". All folders and subfolders must be added to Matlab path for all the scripts to work. They have been tested on Matlab version 2016b.
Performing computations in arbitrary precision and viewing files containing such results requires an installed Advanpix Multiprecision Computing Toolbox (free 1-week trial available at www.advanpix.com).

All queries regarding the code can be directed to Denis Tkachenko (denis.tkachenko@nus.edu.sg).

Note that using different Matlab and Advanpix toolbox versions, as well as different PC hardware and OS, may produce slightly different values for the tiny numbers that are supposed to be zero from those reported in the paper (i.e., numbers that are shrinking to zero as precision improves) – this is normal and a known fact in the numerical analysis literature.

This readme file is structured as follows.

First, we give some general information on how the replication files are organized.

Finally, we append a discussion on optimization algorithms (the genetic algorithm, the particle swarm algorithm, and the multistart algorithm). There, we highlight what aspects of the specifications are important for convergence and computational time.

****
***GENERAL INFORMATION
****

Most importantly, the directories ‘Section X_Y_Z’ contain subfolders with the scripts and necessary section-specific subroutines reproducing results discussed in the paper in the particular section. E.g., the folder Section_5_2 reproduces all results discussed in Section 5.2 of the paper in the order they appear, with the main script being “Section_5_2.m”. For ease of following results, the specific analyses are called via respective scripts. The scripts may have path dependency (e.g., previous results need to be obtained and loaded), however, all can be run independently if the “Output_files” folder is on the Matlab path. The comments within the main section of the replication files and the scripts there as well as short readme.txt files within each section clarify their contents and purpose. Finally, the “Output_Files” directory contains all the results as generated by the replications scripts, which enable examining all the relevant results and running any separate script independently.

The directory ‘Model files’ contains files that set up and solve the linearized version of the respective model, both in double and quadruple precision.

The directory 'General' contains Chris Sims (2002) gensys and csmiwnel routines as well as the gensys code modified to allow indeterminacy following the development in Lubik and Schorfheide (2004). It also contains the general scripts that perform Genetic algorithm and Particle Swarm optimization, as well as additional local optimization using MultiStart when minimizing the Kullback-Leibler (KL) distance after taking the problem inputs. See GA_optim.m and PSO_otpim.m for more details.

The directory “Leeper91_identification” contains the files pertaining to computing local identification conditions, objective functions, and constraints for global identification for the small scale monetary-fiscal interactions model.

The directory “Leeperetal17_identification” contains the files pertaining to computing local identification conditions, objective functions, and constraints for global identification for the medium scale monetary-fiscal interactions model. The model solution code is based on the Matlab code accompanying Leeper et al. (2017).

The directory “SGU2012_identification” contains the files pertaining to computing local identification conditions, objective functions, and constraints for global identification for the medium scale model with news shocks from Schmitt-Grohé and Uribe (2012). The model solution code is based on the Fortran code of Herbst and Schorfheide (2014). We also rely on the Chahrour and Jurado (2016) Matlab code for the example of observational equivalence between news and noise model structures.

The rest of the auxiliary scripts and subroutines are placed within the section folders. Inside each subfolder, where possible, the files are split into “Double” and “MP” subfolders to separate the code that computes in double precision only or arbitrary precision only. Some scripts have both types of computation (e.g., the local identification analyses in Sections 5.2 and 6.1).

References:

Chahrour, R., and K. Jurado (2018): “News or Noise? The Missing Link”, American Economic Review, 108(7), 1702-36.

Herbst, E. P., and F. Schorfheide (2014): “Sequential Monte Carlo Sampling For DSGE Models”, Journal of Applied Econometrics, 29, 1073-1098.

Leeper, E. M., N. Traum, and T. B. Walker (2017): “Clearing Up the Fiscal Multiplier Morass”, American Economic Review, 107, 2409-2454.

Lubik, T., and F. Schorfheide (2004): “Testing for Indeterminacy: An Application to U.S. Monetary Policy”, American Economic Review, 94, 190-217.

Schmitt-Grohé, S., and M. Uribe (2012): “What’s News in Business Cycles”, Econometrica, 80, 2733-2764.

Sims, C. A. (2002): “Solving Linear Rational Expectations Models”, Computational Economics, 20, 1-20.

****
*****GENETIC ALGORITHM: BRIEF DESCRIPTION AND NOTES ON IMPLEMENTATION**
****

The Genetic algorithm belongs to the class of evolutionary optimization algorithms. The optimization begins with usually a randomly drawn "population" of individuals (candidate solutions). The objective function value is evaluated for each individual and a group of "parents" is selected based on their fitness. Then, a fraction of parents is randomly either combined (crossover operation) or mutated via some random perturbation (mutation operation) to produce the next generation to be used in the next iteration. A small fraction of individuals with high fitness values are carried over unchanged (elite children). This process of recombining and mutating the population of candidate solutions continues until specified stop criteria are met. The algorithm implementation in Matlab is parallelized so computation time can be reduced with multiple cores.

The main options for the algorithm in the code are:

- Population size. Larger populations allow GA to better explore the parameter space while increasing the computational burden at the same time. There are no formal rules derived as to how the population size should be set. One popular rule of thumb in the evolutionary computation literature is to set the population size roughly to 10 times the dimension of the problem. Experimenting with our problems, we found that population sizes of 100 for small scale models and 300 for medium scale model produce a balance between convergence robustness and computational cost.

- The maximum number of generations. This is one of the stopping criteria and determines the maximum number of times a population is evolved. We set this for 1000 for all cases. The rationale is that GA has good global exploration ability but low local exploitation ability. We found that 1000 generations is typically enough to pinpoint promising regions of parameter space where the minimum can be located. While in principle GA can be allowed to run until the global minimum is reached, at a certain point it becomes inefficient to use GA for what effectively becomes a local search. This is especially apparent in unconstrained problems such as those considered in Tables 4 and 13 - GA can go through several thousand generations making very small improvements and the computation time can be substantial. By limiting the GA run to 1000 generations and using the MultiStart algorithm on points from its final population produces one seems to achieve a good balance between GA's global exploration and the efficiency of the derivative-based local solver in the second stage (note that the KL distance is typically infinitely differentiable).

- Elite Count. This is the number of so-called "elite" individuals that continue on without crossover or mutation. Setting it to a positive number guarantees that the algorithm will improve on the next iteration. It is advised to keep this parameter low to prevent a few solution points dominating the population and making the search less efficient. We find that assigning a small number, such as 3, works well in cases of both the small and medium scale models. Note that originally the default Matlab option for this parameter was 2, and was changed to 5% of the population in more recent releases. We find that using fewer elite individuals than 5% of the population in our examples improves both the speed and the efficiency of the search.

- Stall Generation Limit. This is another stopping criterion, which halts the algorithm if no improvement has been made over a certain number of generations. We set this parameter to 50. We also experimented with setting it to 100. The results are not sensitive. The tradeoff is a standard one: setting it too low may result in the algorithm stopping in a locally flat region that is not a global optimum, while setting it too high may cause the algorithm to go for more generations than necessary which increases computation time.

- Initial population. We let the initial population be randomly initialized via uniform draws within parameter bounds. This way all regions of the parameter space are treated equally. If the researcher possesses knowledge about possible regions where the global optimum is expected, particular points from those regions can be added to potentially increase the speed of the search. However, this may bias the search in a particular direction and miss the global optimum.

- Objective function tolerance level. We set it to 1E-10 throughout. We also experimented with setting it to 1E-12. Setting it to a low value may increase the computational time, but is often worthwhile to avoid premature termination of the code.

- Constraint handling method. From version 2014b onwards, Matlab provides two options for handling inequality constraints: "auglag" and "penalty". The "penalty" method is preferred as it is often faster and also allows us to easily control the number of generations. The "auglag" option uses the Augmented Lagrangian algorithm that creates a sequence of subproblems using the inequality constraints. After a subproblem is minimized to desired accuracy, the outer problem result is updated. Thus, the effective number of generations (i.e., iterations of the outer problem) is not easily predictable.

There are other tuning parameters of the algorithm, such as mutation and crossover fractions. We experimented with changing their values and did not find a consistent improvement over the Matlab default values. Therefore, we do not provide options for changing them within the code. For more details, one can consult Matlab documentation for the Global Optimization Toolbox.

****
**PARTICLE SWARM ALGORITHM: BRIEF DESCRIPTION AND NOTES ON IMPLEMENTATION
****

Particle Swarm algorithm is similar to GA in that it is an iterative stochastic search procedure that involves a population (in PSO language - swarm) of candidate solutions (particles). The key difference of PSO from GA is that the particles have memory and can communicate with each other and thus can change the direction of the search. There are many variations of the PSO algorithm in the literature. The one implemented by Matlab is as follows. Similarly to GA, the algorithm initializes the swarm of a specified size randomly via uniform draws from within the parameter bounds. However, the updating scheme is different. Particles move through the parameter space according to the following equations:

```
v_j(t + 1) = w*v_j(t) + c1*R1(pbest_j - theta_j(t)) + c2*R2(nbest_j - theta_j(t)),
theta_j(t + 1) = theta_j(t) + v_j(t + 1),
```

where theta_j(t) stand for particle j at iteration t, v_j(t) is particle's velocity at iteration t, pbest_j is the parameter vector that achieved the best function value so far for particle j, nbest_j is the best parameter so far in the current neighborhood of particle j. The parameter w is called the inertia weight, c1 and c2 are called cognitive and social weights respectively, and R1 and R2 are randomly drawn vectors, each element being a draw from a uniform [0,1] distribution. It can be seen that the updating of candidate solution consists of three components: 1) inertia (maintaining the same step size in updating); 2) cognitive attraction (moving towards personal best achieved so far); 3) social attraction (moving towards the best solution obtained over all particles in the neighborhood). The parameters w, c1, and c2 control the relative importance of the three components. Matlab default values for c1 and c2 are set the same at 1.49.The inertia parameter w is allowed to vary with iterations in MATLAB. The literature in the field suggests that starting out with higher inertia and progressively lowering it leads to better performance. Namely, starting out with a value in the range [0.9,1.2] and going down progressively to something like 0.1 gives good results. The intuition is that the high inertia weight initially throws the particles around a lot creating explosive growth in swarm diversity and thus good exploration of the parameter space. Then, as promising parts of the parameter space are found, lowering the inertia weight allows the particles to concentrate on a more local search instead of "flying over" the minimum if the inertia weight were still high. MATLAB has an adaptive way to change the inertia weight: it blows up the weight if there are fewer than two stall iterations to promote further exploration, and cuts down the weight when the algorithm is stalling in order to conduct a more local search. The inertia weight is kept within the bounds specified by the user (option InertiaRange).

The default is [0.1,1.1]. At the start, MATLAB initializes the inertia weight to the upper bound of the specified range in order to have maximum exploration ability. The neighborhood of a particle is determined randomly and its size, specified as a fraction of the total swarm size, is adaptive: it shrinks when a better point is found, otherwise grows all the way up to the whole swarm. Using the neighborhood's best rather than the swarm's best for the social aspect of updating solutions proved to provide better performance and prevent premature convergence. Intuitively, by slowing down the exchange of information between the particles via neighborhood, they do not rush toward the swarm's best, but rather have a chance to explore other promising areas of the parameter space. The algorithm stops when some specified stopping criterion is reached. The algorithm implementation in Matlab is parallelized so computation time can be reduced with multiple cores.

The main options for the algorithm in the code are:

- Swarm size. Similar to GA, a larger swarm size allows the algorithm to better explore the parameter space, while increasing the computational burden at the same time. There do not seem to be extensive guidelines for setting swarm size in the relevant literature, rather, it is usually problem specific. Balancing the performance/computation time tradeoff, we set the swarm size of 300 for all applications of small scale models, 600 for unconstrained optimization in the medium scale model, and 1000 for constrained cases of the medium scale model.

- The maximum number of iterations. This is one of the stopping criteria and determines the maximum number of times a swarm is updated. We found that 1000 generations are typically enough to pinpoint promising regions of parameter space where the minimum can be located. In principle, the number of iterations can be increased to let the algorithm search for a global minimum by itself, however, in some especially unconstrained problems such as those considered in Tables 4 and 13, it can go through as many as 5000 iterations making very small improvements thus making computation time several times longer than the PSO + Multistart procedure implemented in the code. By limiting the PSO run to 1000 iterations and using the MultiStart algorithm on points from its final swarm seems to achieve a good balance between PSO global exploration and the efficiency of derivative-based local solver in the second stage (note that the KL distance is typically infinitely differentiable).

- Stall iteration limit. This is another stopping criterion, which halts the algorithm if no improvement has been made over a certain number of iterations. We set this parameter to 100. This setting is higher than an analogous one for GA since we find that PSO can more effectively escape a flat region even after many stall iterations due to the adaptive nature of updating candidate solutions.

- Initial swarm. We let the initial swarm be randomly initialized via uniform draws within parameter bounds. This way all regions of the parameter space are treated equally. If the researcher possesses knowledge about possible regions where the global optimum is expected, particular points from those regions can be added to potentially increase the speed of the search. However, this may bias the search in a particular direction and miss the global optimum.

- Objective function tolerance level. We set the tolerance level at 1E-6 throughout for small scale models and 1e-10 for medium scale model.

- Constraint handling method. Matlab implementation of PSO only provides boundary constraint handling. Since we have additional inequality constraints in some cases, we modify the objective function to apply a flat penalty in case a candidate solution point violates the constraints. We found this method works as well or better than setting penalty level proportional to violations of feasibility.

- Minimum Neighborhood Fraction. This is the parameter that determines the minimum neighborhood size that a particle communicates with as a fraction of the total swarm. We found this to be a very important tuning parameter. Setting it to an extreme value of 1 (i.e., each particle immediately learns the swarm's best solution) seems to produce premature convergence even when swarm size is relatively large. We found that lowering this parameter to 0.1 produces good results, as, intuitively, restricting communications between particles prolongs exploration of the parameter space. Recall that this parameter only controls the lower bound of the neighborhood size. Matlab adaptively enlarges or shrinks the neighborhood depending on the search progress.

- Retrieving the swarm. Unlike GA, Matlab does not automatically report the final swarm of particles. In order to retrieve it, we utilize the output function psout.m located in the General folder. It saves the swarms after each 200 iterations. Although not necessary here (we use the final swarm only), this can be helpful in cases where the final swarm could be very homogeneous, so starting the local optimizer from points in an earlier swarm could produce better results.

As is evident from the brief description above, there are other potential tuning parameters, such as inertia weight range and coefficients on social and cognitive attraction parts. We found that results are not as sensitive to modifying these as they are to changing swarm or minimum neighborhood size and that Matlab defaults perform well and seem to correspond to best practices in the relevant literature. Therefore, these parameters are left at default values.

For more details, consult Matlab documentation for the Global Optimization Toolbox. For an overview and the recent standard practice of PSO optimization, see Clerc (2012): "Standard Particle Swarm Optimization" (http://clerc.maurice.free.fr/pso/SPSO_descriptions.pdf).

****

*****MULTISTART ALGORITHM: BRIEF DESCRIPTION AND NOTES ON IMPLEMENTATION**

****

Multistart is not a separate algorithm, but rather a conveniently packaged suite of local optimization routines that allows us to conduct local searches using a fairly large number of initial values. We choose the Active-Set local search algorithm to use with Multistart based on performance.

Multistart is invoked at the second stage of optimization, after GA or PSO population or swarm complete 1000 iterations or another stopping criterion is triggered. We then select the first 50 points from the respective final population/swarm, and additionally 10 equally spaced points from the rest of the population/swarm. For added robustness, we generate 50 random starting points within parameter bounds and run the local search from the 110 specified points. Due to the parallel evaluation of multiple local solvers, such a procedure is computationally feasible and takes a few minutes for small scale models and a few hours for a medium scale model on a modern desktop computer with 8 cores.

The options here are standard options for the Active-Set local optimization algorithm:

- The maximum number of iterations. This option is set to 1000 in the code.

- The maximum number of function evaluations. This is set to 10000 in the code. For more challenging problems, i.e., constrained minimization of KL distance between medium scale models, we set this value to 20000.

- Tolerance level. This is set 1E-10.