This Matlab code replicates the results in all the Tables in Qu and Tkachenko (2015): "Global Identification in DSGE Models Allowing for Indeterminacy". All folders and subfolders must be added to Matlab path for all the scripts to work. They have been tested on Matlab version 2015a.

This readme file is structured as follows.

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

Second, the optimization algorithms (the genetic algorithm, the particle swarm algorithm and the multistart algorithm) are discussed in some detail. There, we highlight what aspects of the specifications are important for convergence and computational time.

Third, we use two examples to illustrate how to implement a constrained and an unconstrained optimization, respectively.

Finally, we use an example to illustrate how to carry out the same analysis but using a different parameterization.

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

Most importantly, the directory "Replication_scripts" contains subfolders with the scripts reproducing and displaying table output labeled by its number and column, e.g., QT_Tables_4_5_part1.m reproduces the first column from the Tables 4 and 5 etc. They can all be run independently.

The directory 'General' contains Chris Sims' gensys and csmiwnel routines as well as the gensys code modified to allow indeterminacy. 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 rest of subroutines are organized by model folder (i.e., AS_Identification, LS_Identification and SW_Identification). Inside each folder, the 'Constraints' folder collects constraint functions used in the global identification analysis and the 'Objectives' folder contains m-files that evaluate the objective function for

different cases. For the Lubik and Schorfheide (2004) model, the 'Local_curve' folder contains files used to obtain the nonidentification curve. The .m files under the root directory deal with model solution, computing KL and empirical distances. The data files contain the benchmark parameter values and the corresponding spectral density.

Finally, the folder "Reparameterization" contains files that illustrate an example of replicating column 1 in Table 2 with standard deviations of shocks re-parameterized as variances.

****
*****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 are 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 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 the medium scale model produce a balance between convergence robustness and computational cost.

- 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 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 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 an explosive growth in swarm diversity and hence 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 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. Similarly to GA, larger swarm size allows the algorithm to better explore the parameter space, while increasing computational burden at the same time. There do not seem to be extensive guidelines to 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.

- 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 is 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 the 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 extreme value of 1 (i.e., each particle immediately learns 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 that 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 practice 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 a 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 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:

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

- 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.

****
***REPLICATION FILE STRUCTURE: THE CONSTRAINED PROBLEM**
****

Here an example of replicating column 1 in Table 2 is provided to demonstrate the structure of replication scripts found in the directory "Replication_scripts". This is a constrained KL minimization problem. All files replicating results for these types of problems have the same structure.

The code is broken up into sections for convenience of modification (lines/sections not preceded by '%' can be copypasted and run directly in Matlab):

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Algorithm selection and Matlab versions
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Here the user has an option of selecting the optimization algorithm:
%GA, PSO or running both in sequence:
runga=0; %1 if using Genetic algorithm + Multistart combination, 0 otherwise
runpso=1; %1 if using Particle Swarm + Multistart combination, 0 otherwise

% Note that PSO is available only in version R2014b and later.

%Specify the constraint handling for the Genetic algorithm.
NonlinCon='penalty';
%this is the preferred option as explained above, available in the
matlab version 2014b and later; for earlier versions, set to 'auglag'.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Set up parallel computation
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Here the user specifies the amount of cores to be used in parallel
computation. It is recommended that at least 4 cores be used for
small scale models and at least 8 cores used for the medium scale
model to keep computation time reasonable. This part can be commented
out if the parallel pool is already enabled or is enabled by an
outside script, e.g., when running on the cluster.

numcore=8;
%specify the number of cores according to available/desired capacity
%
%for versions before R2014a, use the following syntax:
```

```
%matlabpool open local numcore
%
%for versions R2014a and above, use this syntax to create the
parallel pool object:

ppool=parpool('local',numcore); %create parallel pool object

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Load parameter values and bounds
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Here the theta0 parameter value is loaded from a file it is stored
in the AS_Identification directory and upper and lower bounds on
parameters are specified.

load theta0asi %load default parameter vector (theta0)

% Set lower and upper bounds
%Parameter order:
%[tau beta kappa psi1 psi2 rhoR rhog rhoz sigR sigg sigz, m_eta_r,
%m_eta_g, m_eta_z, sig_eta]

lb=[0.01 0.9 0.01 0.01 0.01 0.1 0.1 0.1  0.01  0.01 0.01 -3 -3 -3  0.001];

ub=[10   0.999   5  0.9 5 0.99 0.99 0.99   3   3    3      3   3  3 3];


%%%%%%%%%%%%%%%%%%%%%%
%% Select frequencies
%%%%%%%%%%%%%%%%%%%%%%

%Here the user select whether to minimize KL distance over the full
spectrum or over the business cycle frequencies only.
bc=0; %0 for full spectrum; 1 for business cycle frequencies only

%%%%%%%%%%%%%%%%%%%%%%
%% Specify constraints
%%%%%%%%%%%%%%%%%%%%%%

%This section specifies how constraints are imposed: size of the
neighborhood, the associated norm, weighting of the constraints for
individual parameters, and the subset of parameters on which the
constrained is imposed:

ns=0.1; %size of excluded neighborhood.
```

```
nrm=Inf; %set the norm for the constraint function (1,2, or Inf)

numpar=length(lb); %number of parameters in the objective function

wgt=ones(1,numpar);
%vector of weights for the constraint function. Other examples of
weighting could include (ui-li), where ui and li are upper and lower
bounds of 90% posterior intervals; theta' for using maximum
percentage differences etc.

indp=[1:numpar];
%vector of parameter indices to be constrained. E.g.,if indp=[1:3],
the constraint will be imposed only on the first three parameters.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Detailed specifications; you should not have to modify them
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Create the vector of frequencies to approximate KL over:

if bc==0
    n=100; %number of points to evaluate the integral
    w=2*pi*(-(n/2-1):1:n/2)'/n; %form vector of Fourier frequencies
    resfilename=['tables_23_p1'];
elseif bc==1
    n=500;
%number of points to evaluate the integral (here the number of BC
frequencies will be a bit over 100 - close to the full spectrum case)

    w=2*pi*(-(n/2-1):1:n/2)'/n; %form vector of Fourier frequencies
    resfilename=['tables_23_p1_bc']; %filename for saving of results
end

con1=[0,ns];
%constraint handling: con(1)=0 if algorithm handles constraint,
con(1)=1 if penalty to be added to the objective function for
infeasible pioints. con(2) passes the excluded neighborhood size to
the constraint function.

con2=[1,ns]; %set constraint handling for Particle Swarm optimization.


%Specify objective function and constraint handles:
```

```
ObjectiveFunction = @(theta0)kloptas(theta0,con1,w,wgt,nrm,indp,bc);
%set objective function for GA/multistart. It is located in
AS_Identification\Objectives

ObjectiveFunctionP = @(theta0)kloptas(theta0,con2,w,wgt,nrm,indp,bc);
%set penalized objective function for PSO.

ConstraintFunction=@(xest)constraintas(xest,ns,wgt,nrm,indp); %set
%constraint for the problem. It is located in
AS_Identification\Constraints

%Technical details: set algorithms to show iterations to see progress.
Set dispalg to 'off' to display no output, or to 'final' to display
only the final result.

dispalg='iter'; %set whether algorithm iterations are displayed.

dispint=20; %interval between displayed iterations (for PSO only).


%%%%%%%%%%%%%%%%%%%%%%%%
%% GA algorithm settings
%%%%%%%%%%%%%%%%%%%%%%%%

%Here we check whether the flag to run GA is on, and set the
algorithm options if so.
if runga==1
gen=1000; %max number of generation for GA
stgenlim=50;
%max number of stall generations (i.e., over which no improvement
found)

initpop=[];
%set initial population (if smaller than popsize, MATLAB will
randomly draw the rest. If [], the whole population is randomly drawn.
Can be a row vector of dimension numpar or a matrix. Each row of
thematrix is then a candidate initial value.

popsize=100; %population size

elcnt=3;
%elite count - number of elite individuals retained in population

tolfunga=1e-10;
%tolerance level for improvement in the objective for GA
```

```
tolconga=1e-10; %tolerance level for constraint for GA

usepga=['Always'];
%Set to 'Always' to use parallel computation, otherwise to 'Never' or
[].In later versions of Matlab, 1 and 0 can also be used respectively.
end

%%%%%%%%%%%%%%%%%%%%%%%%
%% PSO algorithm settings
%%%%%%%%%%%%%%%%%%%%%%%%

%Here we check whether the flag to run PSO is on, and set the
algorithm options if so.

if runpso==1

swarmsize=300; %swarm size (similar concept to population size for GA)

maxitpso=1000;
%maximum number of iterations (similar concept to generations for GA)

stiterlim=100;
%max number of stall PSO iterations (i.e., over which no improvement
found)

initswarm=[];
%set initial population (if smaller than swarmsize, MATLAB will
randomly draw the rest. If [], the whole population is randomly drawn.
Can be a row vector of dimension numpar or a matrix. Each row of the
matrix is then a candidate initial value.

minfn=0.1;
%smallest fraction of neighbors for PSO (smallest size of the
adaptive neighborhood)

tolfunpso=1e-06;
%tolerance level for improvement in the objective for PSO

psoname=['psoas_c',num2str(ns*100)];
%set name for a temp output file that stores the swarms (problem-
based, the file is automatically deleted after the algorithm finishes
and its contents are saved in %the main results file)

OutFun=@(optimValues,state)psout(optimValues,state,psoname);
```

```
%output function for extracting swarms from PSO runs for further
local optimization. Located in the General folder

useppso=['Always'];
%Set to 'Always' to use parallel computation, otherwise to 'Never' or
[]. For later matlab versions, 1 and 0 can be used respectively.


end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Multistart algorithm settings
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Here we set the options for Multistart: number of random starting
point, which solver to use and some options for that solver.

numrpoints=50; %number of random starting points for Multistart

usepms=['Always'];
%Set to 'Always' to use parallel computation, otherwise to 'Never' or
[]. For later matlab versions, 1 and 0 can be used respectively.

% settings for fmincon
maxit=1000; % set max number of iterations

maxfev=10000; % set max number of function evaluations

tolfunfmc=1e-10;
%tolerance level for improvement in the objective for fmincon

tolconfmc=1e-10; %tolerance level for constraint for fmincon

tolx=1e-10; %tolerance on solution value

localg='active-set';
%set which local algorithm to be used for Multistart


%%%%%%%%%%%%%%%%%%%
%% Run optimization
%%%%%%%%%%%%%%%%%%%
```

```
%Here both GA and PSO flags are checked and the selected algorithm is
run via GA_optim.m or PSO_optim.m script respectively (located in the
General folder)

if runga==1
    timega=tic;
    GA_optim %run GA+Multistart
    timelga=toc(timega); %time taken by GA/Multistart
    save(resfilename) %save intermediate results
end
save(resfilename)

if runpso==1
    timepso=tic;
    PSO_optim %run PSO+Multistart
    timelpso=toc(timepso); %time taken by PSO/Multistart
    save(resfilename) %save intermediate results
end


%%%%%%%%%%%%%%%%%%
%% Arrange results
%%%%%%%%%%%%%%%%%%

%Here results are grouped together from both GA and PSO runs, if
applicable, and the minimizer and the corresponding KL distance value
are saved.


values=[]; %blank for storing best function values
solvecs=[]; %blank for storing solution vectors

if runga==1
    values=[values;fvalga;fvalga2];
    solvecs=[solvecs;xestga;xestga2];
end
if runpso==1
    values=[values;fvalpso;fvalpso2];
    solvecs=[solvecs;xestpso;xestpso2];
end

err=find(values<0);

values(err)=1e07;
```

```
%penalize negative values that may rarely occur due to algorithm
error

indm=find(values==min(values)); %minimum value(s)

indm=indm(1);
%index of the parameter with the lowest objective function value

temp1=['thetaind=solvecs(',num2str(indm),',:)'';'];
%string to evaluate to define minimum parameter value

temp2=['kl=values(',num2str(indm),')/10000;'];
%string to evaluate the resulting KL distance

eval(temp1); %save the final parameter result

eval(temp2); %save the minimized KL distance


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Empirical distance computation
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%After finding the minimizer, empirical distances between it and
theta0 for T=80,150,200,1000 are computed at the 5% significance
level.


%The empirical distance function take the following inputs:
i)theta0 (benchmark model); ii) theta (the model we are comparing
with); iii) benchmark model index (=1 if current inflation + output
gap rule; =2 if expected inflation + output gap rule; =3 if current
inflation + output growth rule.); iv) alternate model index (see %iii
for interpretation); v) significance level (0.05 for 5%, 0.01 %for 1%
etc.); vi) sample size T at which the distance is evaluated.

if bc==0

ed=[pfhas(theta,thetaind,1,1,0.05,80);pfhas(theta,thetaind,1,1,0.05,1
50);pfhas(theta,thetaind,1,1,0.05,200);pfhas(theta,thetaind,1,1,0.05,
1000)];
elseif bc==1

ed=[pfhasbc(theta,thetaind,1,1,0.05,80);pfhasbc(theta,thetaind,1,1,0.
```

```
05,150);pfhasbc(theta,thetaind,1,1,0.05,200);pfhasbc(theta,thetaind,1
,1,0.05,1000)];
end

%the pfhas.m and pfhasbc.m functions used here are located inside the
AS_identification folder.

%%%%%%%%%%%%%%%%%%%%%%%%%
%% Print and save results
%%%%%%%%%%%%%%%%%%%%%%%%%

%This section of the code prints the first column of Tables 2 and 3
and saves the results again.


%Strings for parameter names and column/row headers:

par=['tau     '; 'beta    '; 'kappa   '; 'psi1    '; 'psi2    ';
'rhoR    '; 'rhog    '; 'rhoz    '; 'sigR    '; 'sigg    '; 'sigz
';'m_eta_r '; 'm_eta_g '; 'm_eta_z '; 'sig_eta '];
t0=num2str(theta,3);
ti=num2str(thetaind,3);
t3=['KL    ';'T=80  ';'T=150 ';'T=200 ';'T=1000'];

for i=1:15
    sp(i,:)='  ';
end

%Display the results table:

disp 'Table 2. Parameter values minimizing the KL criterion, AS (2007)
model'
disp '        (a) All parameters can vary'
disp '           theta0     c=0.1'
disp([par,sp,t0,sp,ti])

disp 'Table 3. KL and empirical distances between theta_c and theta_0,
AS (2007) model'
disp '        (a) All parameters can vary'
disp '           c=0.1'
disp([t3,sp(1:5,:),num2str([kl;ed],3)])

%Save results again:
save(resfilename)
```

```
%%%%%%%%%%%%%%%%%%%%%
%% Close parallel pool
%%%%%%%%%%%%%%%%%%%%%

%Here we clean up by closing the parallel pool. This part can be
commented out if more parallel jobs are run after this file.

% %for versions before R2014a
% matlabpool close


% %for versions R2014a and above
delete(ppool) %delete the parallel pool object
```

****
***REPLICATION FILE STRUCTURE: THE UNCONSTRAINED PROBLEM
****

Here an example of replicating column 1 in Table 4 is provided to demonstrate the structure of replication scripts found in the directory "Replication_scripts". This is an unconstrained KL minimization problem: we search for a model with an expected inflation rule that is closest to the model at theta0 with the current inflation rule. All files replicating results for these types of problems have the same structure.

The code is broken up into sections for convenience of modification (lines/sections not preceded by '%' can be copy-pasted and run directly in Matlab), and the structure is almost the same as above. The main differences are:

1) The "Specify constraints" section is no longer present as there are no inequality constraints. Also, constraint handling method for GA is no longer necessary to specify in the first section.

2) There is no constraint function, and the objective function takes fewer inputs as a result. Also, there is no need to re-specify the objective function with a different "con" input for PSO for the above reason. As a result, the ConstraintFunction is set to [] (empty), and the PSO objective function is exactly the same as that for GA.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Algorithm selection and Matlab versions
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Here the user has an option of selecting the optimization algorithm: GA, %PSO or running both in sequence:
runga=0; %1 if using Genetic algorithm + Multistart combination, 0 otherwise
runpso=1; %1 if using Particle Swarm + Multistart combination, 0 otherwise

% Note that PSO is available only in version R2014b and later.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Set up parallel computation
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Here the user specifies the amount of cores to be used in parallel computation. It is recommended that at least 4 cores be used for
```

```
small scale models and at least 8 cores used for the medium scale
model to keep computation time reasonable. This part can be commented
out if the parallel pool is already enabled or is enabled by an
outside script, e.g., when running on the cluster.

numcore=8; %specify the number of cores according to desired capacity
%
% for versions before R2014a, use the following syntax:
% matlabpool open local numcore
%
% for versions R2014a and above, use this syntax to create the
parallel pool object:

ppool=parpool('local',numcore); %create parallel pool object


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Load parameter values and bounds
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Same structure as the unconstrained case above

load theta0asi %load default parameter vector (theta0)

% Set lower and upper bounds
%Parameter order:
%[tau beta kappa psi1 psi2 rhoR rhog rhoz sigR sigg sigz, m_eta_r,
%m_eta_g, m_eta_z, sig_eta]

lb=[0.01 0.9 0.01 0.01 0.01 0.1 0.1 0.1  0.01  0.01 0.01 -3 -3  -3  0.001];

ub=[10   0.999   5  0.9 5 0.99 0.99 0.99   3   3    3      3   3  3 3];

numpar=length(lb);%number of parameters in the objective function


%%%%%%%%%%%%%%%%%%%%%
%% Select frequencies
%%%%%%%%%%%%%%%%%%%%%

%Here the user select whether to minimize KL distance over the full
spectrum or over the business cycle frequencies only.
bc=0; %0 for full spectrum; 1 for business cycle frequencies only
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Detailed specifications; you should not have to modify them
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Same structure as unconstrained case, except for difference in
specifying objective functions and constraints:

if bc==0
    n=100; %number of points to evaluate the integral
    w=2*pi*(-(n/2-1):1:n/2)'/n; %form vector of Fourier frequencies
    resfilename=['table_4_p1'];
elseif bc==1
    n=500; %number of points to evaluate the integral
    w=2*pi*(-(n/2-1):1:n/2)'/n; %form vector of Fourier frequencies
    resfilename=['table_4_p1_bc'];
end

ObjectiveFunction = @(theta0)kloptas4(theta0,w,bc);
%set objective function for GA/multistart

ObjectiveFunctionP = ObjectiveFunction;
%PSO objective is now the same as that for GA

ConstraintFunction = [];
%note empty constraint function. We still need to specify it in order
for the same optimization script to work for both constrained and
unconstrained cases.

dispalg='iter'; %set whether algorithm iterations are displayed.
dispint=20; %interval between displayed iterations (for PSO)


%%%%%%%%%%%%%%%%%%%%%%%%
%% GA algorithm settings
%%%%%%%%%%%%%%%%%%%%%%%%

%Here we check whether the flag to run GA is on, and set the
%algorithm options if so.
if runga==1
gen=1000; %max number of generation for GA
stgenlim=50;
%max number of stall generations (i.e., over which no improvement
found)
```

```
initpop=[]; %set initial population (if smaller than popsize, MATLAB
will randomly draw the rest. If [], the whole population is randomly
drawn. Can be a row vector of dimension numpar or a matrix. Each row
of thematrix is then a candidate initial value.

popsize=100; %population size

elcnt=3;
%elite count - number of elite individuals retained in population

tolfunga=1e-10;
%tolerance level for improvement in the objective for GA
tolconga=1e-10; %tolerance level for constraint for GA

usepga=['Always'];
%Set to 'Always' to use parallel computation, otherwise to 'Never' or
[]


%%%%%%%%%%%%%%%%%%%%%%%%
%% PSO algorithm settings
%%%%%%%%%%%%%%%%%%%%%%%%

%Here we check whether the flag to run PSO is on, and set the
%algorithm options if so.

if runpso==1

swarmsize=300; %swarm size (similar concept to population size for GA)

maxitpso=1000;
%maximum number of iterations (similar concept to generations for GA)

stiterlim=100;
%max number of stall PSO iterations

initswarm=[]; %set initial population (if smaller than swarmsize,
MATLAB will randomly draw the rest. If [], the whole population is
randomly drawn. Can be a row vector of dimension numpar or a matrix.
Each row of the matrix is then a %candidate initial value.

minfn=0.1; %smallest fraction of neighbors for PSO

tolfunpso=1e-06;
%tolerance level for improvement in the objective for PSO
```

```
psoname=['psoas_c',num2str(ns*100)]; %set name for a temp output file
that stores the swarms (problem-based, the file is automatically
deleted after the algorithm finishes and its contents are saved in
the main results file)

OutFun=@(optimValues,state)psout(optimValues,state,psoname); %output
function for extracting swarms from PSO runs for further local
optimization. Located in the General folder

useppso=['Always']; %Set to 'Always' to use parallel computation,
otherwise to 'Never' or []. For later matlab versions, 1 and 0 can be
used respectively.


end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Multistart algorithm settings
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


numrpoints=50; %number of random starting points for Multistart

usepms=['Always']; %Set to 'Always' to use parallel computation,
otherwise to 'Never' or []. For later matlab versions, 1 and 0 can be
used respectively.

% settings for fmincon
maxit=1000; % set max number of iterations

maxfev=10000; % set max number of function evaluations

tolfunfmc=1e-10;
%tolerance level for improvement in the objective for fmincon

tolconfmc=1e-10; %tolerance level for constraint for fmincon

tolx=1e-10; %tolerance on solution value

localg='active-set';
%set which local algorithm to be used for Multistart
```

```
%%%%%%%%%%%%%%%%%%
%% Run optimization
%%%%%%%%%%%%%%%%%%

%Here both GA and PSO flags are checked and the selected algorithm is
run via GA_optim.m or PSO_optim.m script respectively (located in the
General folder)

if runga==1
    timega=tic;
    GA_optim %run GA+Multistart
    timelga=toc(timega); %time taken by GA/Multistart
    save(resfilename) %save intermediate results
end
save(resfilename)

if runpso==1
    timepso=tic;
    PSO_optim %run PSO+Multistart
    timelpso=toc(timepso); %time taken by PSO/Multistart
    save(resfilename) %save intermediate results
end


%%%%%%%%%%%%%%%%%%
%% Arrange results
%%%%%%%%%%%%%%%%%%

Here results are grouped together from both GA and PSO runs, if
applicable, and the minimizer and the corresponding KL distance value
are saved.


values=[]; %blank for storing best function values
solvecs=[]; %blank for storing solution vectors

if runga==1
    values=[values;fvalga;fvalga2];
    solvecs=[solvecs;xestga;xestga2];
end
if runpso==1
    values=[values;fvalpso;fvalpso2];
    solvecs=[solvecs;xestpso;xestpso2];
end
```

```
err=find(values<0);

values(err)=1e07;
%penalize negative values that may occur due to algorithm error

indm=find(values==min(values)); %minimum value(s)

indm=indm(1);
%index of the parameter with the lowest objective function value

temp1=['thetaind=solvecs(',num2str(indm),',:)'';'];
%string to evaluate to define minimum parameter value

temp2=['kl=values(',num2str(indm),')/10000;'];
%string to evaluate the resulting KL distance

eval(temp1); %save the final parameter result

eval(temp2); %save the minimized KL distance

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Empirical distance computation
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%After finding the minimizer, empirical distances between it and
theta0 for T=80,150,200,1000 are computed at the 5% significance
level.

%The empirical distance function take the following inputs:
 i)theta0 (benchmark model); ii) theta (the model we are comparing
with); iii) benchmark model index (=1 if current inflation + output
gap rule; =2 if expected inflation + output gap rule; =3 if current
inflation + output growth rule.); iv) alternate model index (see iii
for interpretation); v) significance level (0.05 for 5%, 0.01 for 1%
etc.); vi) sample size T at which the distance is evaluated.

%Note here the second model has index 2 (expected inflation rule), so
the 4th input of the empirical distance function reflects that.

if bc==0

ed=[pfhas(theta,thetaind,1,2,0.05,80);pfhas(theta,thetaind,1,2,0.05,1
50);pfhas(theta,thetaind,1,2,0.05,200);pfhas(theta,thetaind,1,2,0.05,
1000)];
```

```
elseif bc==1

ed=[pfhasbc(theta,thetaind,1,2,0.05,80);pfhasbc(theta,thetaind,1,2,0.
05,150);pfhasbc(theta,thetaind,1,2,0.05,200);pfhasbc(theta,thetaind,1
,2,0.05,1000)];
end

%the pfhas.m and pfhasbc.m functions used here are located inside the
AS_identification folder.

%%%%%%%%%%%%%%%%%%%%%%%%
%% Print and save results
%%%%%%%%%%%%%%%%%%%%%%%%

t3=['KL    ';'T=80  ';'T=150 ';'T=200 ';'T=1000'];

for i=1:5
    sp(i,:)='  ';
end
disp 'Table 4. KL and empirical distances between monetary policy
rules, AS (2007) model'
disp '        (a) Expected inflation rule'
disp '         Indeterminacy'
disp([t3,sp,num2str([kl;ed],3)])
save(resfilename)


%%%%%%%%%%%%%%%%%%%%%%
%% Close parallel pool
%%%%%%%%%%%%%%%%%%%%%%

%Here we clean up by closing the parallel pool. This part can be
commented out if more parallel jobs are run after this file.

% %for versions before R2014a
% matlabpool close


% %for versions R2014a and above
delete(ppool) %delete the parallel pool object
```

****
***EXAMPLE OF HANDLING REPARAMETERIZATIONS
****

An example of re-parameterizing standard deviations as variances for checking global identification in the AS(2007) model is provided in the folder 'Reparameterization'. The folder contains an auxiliary file gtheta_as.m that shows an example of writing a script transforming parameters (here from standard deviations to variances and vice versa) from the original vector theta through some vector valued function g(theta) as discussed on p. 18 of the paper. As noted in the paper, there is no need to modify the objective or the empirical distance functions as the KL and empirical distance are invariant to re-parameterizations.

The only modification needed in the constraint function (constraintasg.m) is to apply the above transformation to the candidate solution and the benchmark model parameters before checking the constraint, as follows:

```
%line 12:
xest=gtheta_as(xest,1);
%reparameterize standard deviations as variances
%line 15:
theta=gtheta_as(theta,1);
%reparameterize the benchmark in terms of shock variances
```

The rest of the function is identical to the original case - see constraintas.m

Finally, the script Reparameterized_AS_c01_example.m script, which has the same structure as the replication file for the constrained problem above, showcases the few changes needed to work with the re-parameterization. Specifically, the only differences (apart from renaming result files) from the corresponding script without re-parameterization (see QT_Tables2_3_part1.m) are:

```
%line 94:
ConstraintFunction=@(xest)constraintasg(xest,ns,wgt,nrm,indp);
%call the above constraint function instead of the one without
reparameterization

%line 206:
t0=num2str(gtheta_as(theta,1),3);
```

```
%display the relevant default parameter values as variances

%line 207:
ti=num2str(gtheta_as(thetaind,1),3);
%display the relevant resulting minimizer values as variances
```

In summary, the main work in incorporating re-parameterization is in defining the function that performs the parameter transformation and adjusting the constraint function accordingly.