SolveDSGE.jl v0.5.x — A User Guide

Richard Dennis[*]
University of Glasgow and CAMA

July 2022

Abstract

SolveDSGE is a Julia package for solving nonlinear Dynamic Stochastic General Equilibrium models. A variety of solution methods are available, and they are interchangable so that one solution can be used subsequently as an initialization for another allowing accurate solutions to be quickly obtained. The package can compute first- through fourth-order perturbation solutions and Chebyshev-based, Smolyak-based, hyperbolic-cross-based, and piecewise linear-based projection solutions. Once a model has been solved, the package can used used to simulate data and compute impulse response functions.

JEL Classification: E3, E4, E5.

---

[*]Address for Correspondence: Adam Smith Business School, University of Glasgow, Main Building, University Avenue, Glasgow G12 8QQ; email: richard.dennis@glasgow.ac.uk.

1 Introduction

SolveDSGE.jl is a framework to help you solve and analyze Dynamic Stochastic General Equilibrium (DSGE) models. SolveDSGE.jl can solve nonlinear DSGE models using perturbation methods, producing solutions that are accurate to first, second, third, and fourth order, but this is not its focus. The package's focus is on applying projection methods to obtain solutions that are "globally" accurate, or at least accurate over some specified domain.

Obtaining globally accurate solutions to nonlinear DSGE models is notoriously difficult. Solutions are invariably slow to obtain and model-specific characteristics are often exploited to speed up the solution process. SolveDSGE.jl does not exploit model-specific characteristics in order to solve a model. Instead, SolveDSGE.jl applies the same general solution strategy to all models. Nonetheless, making use of Julia's speed, SolveDSGE.jl allows models to be solved "relatively quickly", and it provides users with an easy, natural, and unified way of organizing and expressing their model. Solutions can be obtained using Chebyshev polynomials, Smolyak polynomials, hyperbolic-cross approximation, or piecewise linear approximation, with the solution obtained from one approximation scheme able to be used as an initialization for the others, allowing greater speed and accuracy to be obtained via a form of homotopy. Model parameters can be varied to facilitate robustness analysis and so that the solution routines can be employed to estimate models.

To use SolveDSGE.jl to solve a model, two files must be supplied. The first file (the model file) summarizes the model to be solved. The second file (the solution file) reads the model file, solves the model, and performs any post-solution analysis.

It is worth repeating that SolveDSGE.jl is a set of functions developed to help you solve and analyze nonlinear models, but it is not automatic that it will solve any model you feed it at the first attempt. Solving nonlinear models almost always involves a process of homotopy—the intent of the package is to help you with this homotopy process.

Quite a lot of time and effort has gone into writing SolveDSGE.jl, together with the underlying modules: ChebyshevApprox.jl, SmolyakApprox.jl, HyperbolicCrossApprox.jl, PiecewiseLinearApprox.jl, and NLboxolve.jl. but it is far from perfect. SolveDSGE.jl may not be able to solve your model, or it may not obtain a solution quickly enough to be useful to you. You are welcome to suggest improvements to fix bugs or add functionality. At the same time, I am hopeful that you will find the package useful for your research. If it is, then please cite this User Guide and add an acknowledgement to SolveDSGE.jl to your paper/report.

2 What types of models can be solved?

SolveDSGE.jl is designed to solve discrete-time models, $t = 1, 2, 3, ...$, that can be written in the following standard form:

$$E_t \left[\mathbf{f} \left(\mathbf{x}_t, \mathbf{y}_t, \mathbf{x}_{t+1}, \mathbf{y}_{t+1}, \boldsymbol{\varepsilon}_{t+1} \right) \right] = \mathbf{0}, \tag{1}$$

where $\mathbf{x}_t$ is a vector of state variables, $\mathbf{y}_t$ is a vector of jump variables, $\boldsymbol{\varepsilon}_{t+1}$ is a vector of shocks, $\mathbf{f}$ is a vector-function, and E_t is the mathematical expectation operator conditional upon information up to the end of period t. The state and jump variables can be constrained, as per $\mathbf{x}^l \leq \mathbf{x} \leq \mathbf{x}^u$ and $\mathbf{y}^l \leq \mathbf{y} \leq \mathbf{y}^u$. The model's first-order conditions and constraints are specified equation-by-equation in the model file. The shocks, state variables, and jump variables are then defined and then SolveDSGE.jl takes the model and expresses it in the form of equation (1) in preparation for solution. The solution takes the form:

$$\mathbf{x}_{t+1} = \mathbf{h} \left(\mathbf{x}_{t+1} \right) + \mathbf{k} \boldsymbol{\varepsilon}_{t+1}, \tag{2}$$

$$\mathbf{y}_t = \mathbf{g} \left(\mathbf{x}_t \right), \tag{3}$$

and the goal of the package is to provide approximations to the unknown, stationary, vector-functions: $\mathbf{h}$ and $\mathbf{g}$.

Equation (1) covers a wide set of models, but obviously not all models. In principle SolveDSGE.jl can be applied to standard business cycle models of the real and new Keynesian varieties, and to (some) models with heterogeneous agents, but it cannot (yet) be applied to models with generalized Euler equations, like those that emerge from discretionary policy problems or quasi-geometric discounting, or to learning models, and it hasn't been tested on models with "news" shocks.

2.1 Shock processes

It would be nice for the shocks processes to be very general, but this is a little tricky because the shock processes are interwound with the quadrature methods used to compute expectations. Rather than restrict the shock processes to a form suitable for all of the package's solution methods, I have instead, allowed differences across the solution methods regarding the forms that the shock processes can take. The baseline, if you like, is for the model to have shocks that obey simple AR(1) processes:

$$z_{t+1} = \rho z_t + \sigma_\varepsilon \varepsilon_{t+1}, \tag{4}$$

where $\rho \in (-1, 1)$, σ_ε is positive, finite, and "small", and $\varepsilon_t \sim i.i.d.N(0, 1)$. The assumptions underpinning equation (4) are restrictive, but less-so than may first appear.

Consider the more general process:

$$s_{t+1} = a + \rho s_t + \sigma_\epsilon \varepsilon_{t+1}. \tag{5}$$

Let $(1 - \rho)\,\overline{s} = a$, then equation (5) can be rescaled and written as:

$$\frac{\sigma_\varepsilon}{\sigma_\epsilon}\left(s_{t+1} - \overline{s}\right) = \frac{\sigma_\varepsilon}{\sigma_\epsilon}\left(\rho\left(s_t - \overline{s}\right) + \sigma_\epsilon \varepsilon_{t+1}\right). \tag{6}$$

Now define $z_t = \frac{\sigma_\varepsilon}{\sigma_\epsilon}\rho\left(s_t - \overline{s}\right)$, allowing equation (6) to be expressed in the desired form:

$$z_{t+1} = \rho z_t + \sigma_\varepsilon \varepsilon_{t+1}. \tag{7}$$

In other words, through a suitable transformation of the shocks, models with high-volatility innovations can be recast as ones with low-volatility innovations and models with non-zero mean shocks can be recast in terms of zero-mean shocks—one simply needs to employ to inverse transformation elsewhere in the model so that the model's original properties are preserved.

2.1.1 Vector-shocks

More generally still, all of the solution methods can be applied to models where the shocks are of the form:

$$\mathbf{z}_{t+1} = \mathbf{\Phi}\mathbf{z}_t + \mathbf{\Gamma}\boldsymbol{\varepsilon}_{t+1}, \tag{8}$$

provided $\mathbf{\Phi}$ has spectral radius less than one and $\mathbf{\Gamma}$ is a diagonal matrix (so the innovations are independent of each other).

In the case that $\mathbf{\Gamma}$ is not a diagonal matrix, the model can still be solved using the perturbation methods, and using projection methods with Chebyshev polynomials or piecewise linear approximation (but not using Smolyak polynomials or hyperbolic-cross approximation).

2.1.2 MA shocks

When the shock process has an MA(1) component, i.e.:

$$z_{t+1} = \rho z_t + \sigma_1 \varepsilon_t + \sigma_2 \varepsilon_{t+1},$$

one can rewrite the process as:

$$
\begin{aligned}
z_{t+1}^1 &= \sigma_1 \varepsilon_{t+1}, \\
z_{t+1}^2 &= \rho z_t^2 + z_t^1 + \sigma_2 \varepsilon_{t+1},
\end{aligned}
$$

which is in the form of equation (8) with correlated innovations.

2.1.3 Volatility shocks

Volatility shocks can be modeled as follows:

$$
\begin{aligned}
z_{t+1} &= \rho z_t + \sigma z \exp(\sigma_t)\varepsilon_{t+1}, \\
\sigma_{t+1} &= \gamma\sigma_t + \epsilon_{t+1},
\end{aligned}
$$

and all of the solution methods in the package can be applied to such processes.

3 The model file

SolveDSGE.jl requires the model to be solved to be summarized and stored in a model file. The model file is simply a text file so there is nothing particularly special about it. Every model file must contain the following five information categories: "states:", "jumps:", "shocks:", "parameters:", and "equations:"; each category name must end with a colon. Each category will begin with its name, such as "states:" and conclude with an "end". The model file can present these five categories in any order.

It is perhaps easiest to illustrate a model file with an example: here is the contents of a model file for the stochastic growth model.

3.1 Example

Model file for the Brock and Mirman (1972) stochastic growth model.

states:

k, z

end

jumps:

c, ce

end

shocks:

ε

end

parameters:

$\beta = 0.99$

$\sigma = 1.1$

$\delta = 0.025$

$\alpha = 0.30$

$\rho = 0.8$

$\sigma z = 0.01$

end

solvers: Any

equations:

k(+1) = (1.0 - δ)*k + exp(z)*k^α - c

c^(-σ) = β*ce(+1)

ce = c^(-σ)*(1.0 - δ + α*exp(z)*k^(α - 1.0))

z(+1) = ρ*z + σz*ε

end

3.2 Elaboration

The model file can contain comments, preceded by a #, which are ignored by the parser; generally there should be only one # per line. If there are multiple #'s in a line, then the odd numbered #'s are interpreted as "comment on" while the even numbered #'s are interpreted as "comment off". Unless you are careful, having multiple #'s in a line will produce a parser error.

Following the comment, which here simply documents the model being solved, there are the five categories: "states:", "jumps:", "shocks:", "parameters:", and "equations:". The information in each category can be presented with one element per line, or with multiple elements on each line with each element separated by either a comma or a semi-colon. So if the jump variables in a model happen to be labor, consumption, and output, then this could be presented in a variety of ways, such as:

jumps:

labor

consumption

output

end

or:

jumps:

labor, consumption, output

end

or:

jumps:

labor; consumption, output

end

The model file also contains the optional category "solvers:", which in the example file is designated as "Any". Other possibilities are "Linear", "Perturbation", and "Projection"; the default is "Any".

The first lag of a variable is denoted with a -1, so the lag of c is $c(-1)$. Similarly the first lead of a variable is denoted with a $+1$, so the lead of c is denoted $c(+1)$. The first lag of any model variable is automatically included as a state variable, second and higher lags should be given a name, defined by an equation, and included as a state variable explicitly.

In the package, shocks refers to the innovations to the shock processes, so if the shock process is given by:

$$z(+1) = \rho * z + \sigma z * \varepsilon,$$

then "z" will be a state variable, "ε" will be a shock, and "ρ" and "σz" will be parameters. If the model is deterministic, then it will contain no shocks.

Every element in the equations category must contain an "=" sign, such as: "$y = exp(z) * k^\alpha * l^{\,}(1.0 - \alpha)$". If the model is deterministic, or if you are only interested in a perturbation solution, then there is more flexibility regarding how the model is written. But if your model is stochastic and you are interested in a projection solution then the model must be written so that it is linear in expectations (to avoid errors associated with Jensen's inequality). What does this mean? Well, taking the consumption-Euler equation as an example, if you were only going to apply the perturbation solvers, then you could write it as (form one):

c^(-σ) = β*c(+1)^(-σ)*(1.0 - δ + α*exp(z(+1))*k(+1)^(α - 1.0))

while if you were going to apply a projection solver, then you would need to write it as two equations (form two):

c^(-σ) = β*ce(+1)

ce = c^(-σ)*(1.0 - δ + α*exp(z)*k^(α - 1.0))

6

My suggestion is that you should generally write your model in the second form so that all solvers can be applied. However, if your model has lots of state variables and/or you are only interested in using perturbation to solve the model, then the first form is simpler—just make sure you put "solvers: Perturbation" in your model file. In many instances the errors associated with Jensen's inequality are small—certainly smaller than the errors between the model and reality—, but for some models/parameterizations or if you are interested in risk premia and such like, then the errors can matter. Similarly, if there are restrictions on the values for some variables, such as investment or the nominal interest rate cannot be negative, then you would need to use the second form and you would want to put "solvers: Projection" in your model file.

In the case of the paramerters category, parameter values can (and will usually) be assigned in the model file, such as: "$\alpha = 0.30$". However, parameters can also be assigned values at a later stage—after the model has been processed. It can be useful to assign values to parameters after the model has been processed as this facilitates estimation and allows a model to be solved for a range of parameterizations. To assign a value to a parameter after the model has been processed, only the parameter name gets listed in the parameters category: i.e., "α".

Finally, variables and parameters can generally be given any name you want, but it is nice and convenient to be able to give variables and parameters the same names in the model file as you are using in your paper, and the parser allows you to do that. There are some reserved names, however: "exp", "log", "x", "p", "deriv", ":", and ";".

4 Solving a model

Solving a model is straightforward; it consists of the following steps:

1. Read and process the model file. During the processing the order of variables and equations in the system may be changed, typically the changes are to place the shocks at the top of the system. After processing is complete you will be told what the variable-order is. Any parameters that do not have values assigned are also reported.

2. Assign values to any parameters that were not given a value in the model file.

3. Solve for the model's steady state. This is actually an optional step if the model is to be solved using a projection method, but knowing the steady state is often useful.

4. Specify a SolutionScheme. A SolutionScheme specifies the solution method along with any parameters needed to implement that solution method.

7

5. Solve the model according to the chosen SolutionScheme.

4.1 Reading the model and solving for its steady state

To read and process a model file we simply supply the path/filename to the process_model() function, for example:

process_model("c:/desktop/model.txt")

The processed model is saved in the same folder as the model file, which in then retrieved and stored in a structure:

dsge = retrieve_processed_model("c:/desktop/model_processed.txt")

When the model is processed the package may report that one of more parameters do not have values assigned (the parameter names are listed). If this is the case, then values must be assigned to these parameters before the model can be solved. This is simple to do through the assign_parameters() function:

dsge = assign_parameters(dsge,params)

where *params* is either a vector containing the needed values in the order that the parameters were earlier listed or a dictionary. (The name of the model generated by the assign_parameters() function does not need to be the same as the model fed into the function, and will generally be different.) We can then solve for the model's steady state as follows:

ss_obj = compute_steady_state(dsge,tol,maxiters)
ss = ss_obj.zero

where *dsge* is the model whose steady state is to be computed, *tol* is a convergence tolerance, and *maxiters* is an integer specifying the maximum number of iterations before the function exits. The *compute_steady_state()* function employs the solvers from NLsolve.jl, which allows different solution methods. These can be accessed via the optional keyword argument: method, where the available methods are *:newton* (the default) and *:trust_region*. So, one could solve for the steady state using:

ss_obj = compute_steady_state(dsge,tol,maxiters,method=:trust_region)
ss = ss_obj.zero

Once the solver has finished it returns the results in a structure that has been called *ss_obj* in the example above. *ss_obj* (or whatever you have called it), contains the steady state and information regarding convergence. The steady state vector itself is extracted from this structure in the second line (*ss = ss_obj.zero*).

4.2   Specifying a SolutionScheme

To solve a model a SolutionScheme must be supplied. A SolutionScheme specifies the solution method and the parameters upon which that solution method relies. The solution methods in SolveDSGE.jl are either perturbation methods or projection methods. Accordingly, the Solution-Schemes can be divided into PerturbationSchemes and ProjectionSchemes. We present each in turn.

4.2.1   PerturbationSchemes

To solve a model using a perturbation method requires a PerturbationScheme. Regardless of the model or the order of the perturbation, a PerturbationScheme is a structure with three fields: the point about which to perturb the model (generally the steady state), a cutoff parameter that separates unstable from stable eigenvalues (eigenvalues whose modulus is greater than cutoff are placed in the model's unstable block), and the order of the perturbation. For a first-order perturbation, a typical PerturbationScheme might be the following

cutoff = 1.0
P = PerturbationScheme(ss,cutoff,"first")

while those for second and third order perturbations might be

PP = PerturbationScheme(ss,cutoff,"second")

and

PPP = PerturbationScheme(ss,cutoff,"third")

The method used to compute a first-order perturbation follows Klein (2000), that for a second-order perturbation follows Gomme and Klein (2011), that for a third-order perturbation follows Binning (2013), while that for a fourth-order perturbation builds on Binning (2013) and Levintal (2017). At this point, perturbation solutions higher than fourth order are not supported.

4.2.2 ProjectionSchemes

ProjectionSchemes are either ChebyshevSchemes, SmolyakSchemes, HyperbolicCrossSchemes or PiecewiseLinearSchemes, and for each of these there are stochastic (for stochastic models) and deterministic (for deterministic models) versions and versions that allow for occasionally binding constraints and those that do not. The SolutionScheme for a deterministic model is a special case of the stochastic one, so we focus on the stochastic case in what follows.

ChebyshevSchemes Solutions based on Chebyshev polynomials rely on and make use of all of the functionality of the module ChebyshevApprox.jl. This means that an arbitrary number of state variables can be accommodated (if you have enough time!) and both tenser-product and complete polynomials can be used. A stochastic ChebyshevScheme requires the following arguments:

- initial_guess — This will usually be a vector containing the model's steady state. It is used as the initial guess at the solution in the case where an initializing solution is not provided (see the section on model solution below).

- node_generator — This is the name of the function used to generate the nodes for the Chebyshev polynomial. Possible options include: chebyshev_nodes and chebyshev_extrema.

- node_number — This gives the number of nodes to be used for each state variable. If there is only one state variables then node_number will be an integer. When there are two or more state variables it will be a vector of integers.

- num_quad_nodes — This is an integer specifying the number of quadrature points used to compute expectations.

- order — This defines the order of the Chebyshev polynomial to be used in the approximating functions. For a complete polynomial order will be an integer; for a tenser-product polynomial order will be a vector of integers.

- domain — This contains the domain for the state variables over which the solution is obtained. Domain will be a $2-$element vector in the one-state-variable case and a $2 \times n$ array in the n-state-variable case, with the first row of the array containing the upper values of the domain and the second row containing the lower values of the domain. If an initializing solution is provided, then the domain associated with that initializing solution can be used by setting domain to an empty array, Float64[].

- tol_fix_point_solver — This specifies the tolerance to be used in the inner loop to determine convergence at each solution node.

- tol_variables — This specifies the tolerance to be used in the outer loop to determine convergence of the overall solution.

- maxiters — This is an integer specifying the maximum number of outer-loop iterations before the solution exits.

- solver — Specifies the solver in NLsolve.jl, either :newton or :trust_region.

An example of a stochastic ChebyshevScheme is:

$C = ChebyshevSchemeStoch(ss,chebyshev\_nodes,[21,21], 9, 4,[0.1\ 30.0;\ -0.1\ 20.0],1e-8,1e-6,1000,:newton)$

In the deterministic case the number of quadrature nodes is not needed, i.e.,

$Cdet = ChebyshevSchemeDet(ss,chebyshev\_nodes,[21,21],4,[0.1\ 30.0;\ -0.1\ 20.0],1e-8,1e-6,1000,:newton)$

ChebyshevSchemes with occasionally binding constraints   A stochastic ChebyshevScheme with occasionally binding constraints requires the following arguments:

- initial_guess — This will usually be a vector containing the model's steady state. It is used as the initial guess at the solution in the case where an initializing solution is not provided (see the section on model solution below).

- node_generator — This is the name of the function used to generate the nodes for the Chebyshev polynomial. Possible options include: chebyshev_nodes and chebyshev_extrema.

- node_number — This gives the number of nodes to be used for each state variable. If there is only one state variables then node_number will be an integer. When there are two or more state variables it will be a vector of integers.

- num_quad_nodes — This is an integer specifying the number of quadrature points used to compute expectations.

- order — This defines the order of the Chebyshev polynomial to be used in the approximating functions. For a complete polynomial order will be an integer; for a tenser-product polynomial order will be a vector of integers.

11

- domain — This contains the domain for the state variables over which the solution is obtained. Domain will be a $2-$element vector in the one-state-variable case and a $2 \times n$ array in the n-state-variable case, with the first row of the array containing the upper values of the domain and the second row containing the lower values of the domain. If an initializing solution is provided, then the domain associated with that initializing solution can be used by setting domain to an empty array, Float64[].

- lb — This a vector containing the lower bound on the model's variables.

- ub — This is a vector containing the upper bound on the model's variables.

- tol_fix_point_solver — This specifies the tolerance to be used in the inner loop to determine convergence at each solution node.

- tol_variables — This specifies the tolerance to be used in the outer loop to determine convergence of the overall solution.

- maxiters — This is an integer specifying the maximum number of outer-loop iterations before the solution exits.

- solver — Specifies the solver in NLboxsolve.jl, either :nr,:lm, :lm_kyf, or :lm_ar.

An example of a stochastic ChebyshevScheme with occasionally binding constraints is:

$C = ChebyshevSchemeOBCStoch(ss,chebyshev\_nodes,[21,21],\ 9,\ 4,[0.1\ 30.0;\ -0.1\ 20.0],[-Inf, -Inf,\ -Inf],[Inf,\ Inf,\ Inf],1e-8,1e-6,1000,:lm\_ar)$

In the deterministic case the number of quadrature nodes is not needed, i.e.,

$Cdet = ChebyshevSchemeOBCDet(ss,chebyshev\_nodes,[21,21],4,[0.1\ 30.0;\ -0.1\ 20.0],[-Inf,\ -Inf, -Inf],[Inf,\ Inf,\ Inf],1e-8,1e-6,1000,:lm\_ar)$

SmolyakSchemes   Underlying the Smolyak polynomial based solution is the module SmolyakApprox.jl. This module allows for both isotropic polynomials and ansiotropic polynomials and several different methods for producing nodes. SolveDSGE.jl exploits all of this functionality. A stochastic SmolyakScheme requires the following arguments:

- initial_guess — This will usually be a vector containing the model's steady state. It is used as the initial guess at the solution in the case where an initializing solution is not provided (see the section on model solution below).

- node_generator —This is the name of the function used to generate the nodes for the Smolyak polynomial. Possible options include: chebyshev_gauss_lobatto and clenshaw_curtis_equidistant

- num_quad_nodes — This is an integer specifying the number of quadrature points used to compute expectations.

- layer — This is an integer (isotropic case) or a vector of integers (ansiotropic case) specifying the number of layers to be used in the approximation.

- domain — This contains the domain for the state variables over which the solution is obtained. Domain will be a 2−element vector in the one-state-variable case and a $2 \times n$ array in the n-state-variable case, with the first row of the array containing the upper values of the domain and the second row containing the lower values of the domain. If an initializing solution is provided, then the domain associated with that initializing solution can be used by setting domain to an empty array, Float64[].

- tol_fix_point_solver — This specifies the tolerance to be used in the inner loop to determine convergence at each solution node.

- tol_variables — This specifies the tolerance to be used in the outer loop to determine convergence of the overall solution.

- maxiters — This is an integer specifying the maximum number of outer-loop iterations before the solution exits.

- solver — Specifies the solver in NLsolve.jl, either :newton or :trust_region.

An example of a stochastic SmolyakScheme is:

$S = SmolyakSchemeStoch(ss,chebyshev\_gauss\_lobatto,9,3,[0.1\ 30.0;\ -0.1\ 20.0],1e\text{-}8,1e\text{-}6,1000,:newton)$

In the deterministic case the number of quadrature nodes is not needed, i.e.,

$Sdet = SmolyakSchemeDet(ss,chebyshev\_gauss\_lobatto,3,[0.1\ 30.0;\ -0.1\ 20.0],1e\text{-}8,1e\text{-}6,1000,:newton)$

SmolyakSchemes with occasionally binding constraints   A stochastic SmolyakScheme with occasionally binding constraints requires the following arguments:

- initial_guess — This will usually be a vector containing the model's steady state. It is used as the initial guess at the solution in the case where an initializing solution is not provided (see the section on model solution below).

- node_generator —This is the name of the function used to generate the nodes for the Smolyak polynomial. Possible options include: chebyshev_gauss_lobatto and clenshaw_curtis_equidistant

- num_quad_nodes — This is an integer specifying the number of quadrature points used to compute expectations.

- layer — This is an integer (isotropic case) or a vector of integers (ansiotropic case) specifying the number of layers to be used in the approximation.

- domain — This contains the domain for the state variables over which the solution is obtained. Domain will be a $2-$element vector in the one-state-variable case and a $2 \times n$ array in the n-state-variable case, with the first row of the array containing the upper values of the domain and the second row containing the lower values of the domain. If an initializing solution is provided, then the domain associated with that initializing solution can be used by setting domain to an empty array, Float64[].

- lb — This a vector containing the lower bound on the model's variables.

- ub — This is a vector containing the upper bound on the model's variables.

- tol_fix_point_solver — This specifies the tolerance to be used in the inner loop to determine convergence at each solution node.

- tol_variables — This specifies the tolerance to be used in the outer loop to determine convergence of the overall solution.

- maxiters — This is an integer specifying the maximum number of outer-loop iterations before the solution exits.

- solver — Specifies the solver in NLboxsolve.jl, either :nr,:lm, :lm_kyf, or :lm_ar.

An example of a stochastic SmolyakScheme with occasionally binding constraints is:

$S = SmolyakSchemeOBCStoch(ss,chebyshev\_gauss\_lobatto,9,3,[0.1\ 30.0;\ -0.1\ 20.0],[-Inf,\ -Inf,$
$-Inf],[Inf,\ Inf,\ Inf],1e-8,1e-6,1000,:lm\_ar)$

In the deterministic case the number of quadrature nodes is not needed, i.e.,

$Sdet = SmolyakSchemeOBCDet(ss,chebyshev\_gauss\_lobatto,3,[0.1\ 30.0;\ -0.1\ 20.0],[-Inf,\ -Inf,$
$-Inf],[Inf,\ Inf,\ Inf],1e-8,1e-6,1000,:lm\_ar)$

HyperbolicCrossSchemes   Underlying the hyperbolic cross approximation scheme is the module HyperbolicCrossApprox.jl.   HyperbolicCrossApprox.jl implements a sparse-grid alternative to Smolyak's method and the package allows for both isotropic and ansiotropic grids.   A stochastic HyperbolicCrossScheme requires the following arguments:

- initial_guess — This will usually be a vector containing the model's steady state. It is used as the initial guess at the solution in the case where an initializing solution is not provided (see the section on model solution below).

- node_generator —This is the name of the function used to generate the nodes for the Smolyak polynomial. Possible options include: chebyshev_gauss_lobatto and clenshaw_curtis_equidistant

- num_quad_nodes — This is an integer specifying the number of quadrature points used to compute expectations.

- layer — This is an integer specifying the number of layers to be used in the approximation.

- n — This is an integer or a vector of integers specifying the number of nodes along each spacial dimension.

- domain — This contains the domain for the state variables over which the solution is obtained. Domain will be a $2-$element vector in the one-state-variable case and a $2 \times n$ array in the n-state-variable case, with the first row of the array containing the upper values of the domain and the second row containing the lower values of the domain.   If an initializing solution is provided, then the domain associated with that initializing solution can be used by setting domain to an empty array, Float64[].

- tol_fix_point_solver — This specifies the tolerance to be used in the inner loop to determine convergence at each solution node.

- tol_variables — This specifies the tolerance to be used in the outer loop to determine convergence of the overall solution.

- maxiters — This is an integer specifying the maximum number of outer-loop iterations before the solution exits.

- solver — Specifies the solver in NLsolve.jl, either :newton or :trust_region.

An example of a stochastic HyperbolicCrossScheme is:

$H = HyperbolicCrossSchemeStoch(ss,chebyshev\_gauss\_lobatto,9,5,11,[0.1\ 30.0;\ -0.1\ 20.0],1e-8,1e-6,1000,:newton)$

In the deterministic case the number of quadrature nodes is not needed, i.e.,

$Hdet = HyperbolicCrossSchemeDet(ss,chebyshev\_gauss\_lobatto,5,11,[0.1\ 30.0;\ -0.1\ 20.0],1e-8,1e-6,1000,:newton)$

HyperbolicCrossSchemes with occasionally binding constraints   A stochastic HyperbolicCrossScheme with occasionally binding constraints requires the following arguments:

- initial_guess — This will usually be a vector containing the model's steady state. It is used as the initial guess at the solution in the case where an initializing solution is not provided (see the section on model solution below).

- node_generator —This is the name of the function used to generate the nodes for the Smolyak polynomial. Possible options include: chebyshev_gauss_lobatto and clenshaw_curtis_equidistant

- num_quad_nodes — This is an integer specifying the number of quadrature points used to compute expectations.

- layer — This is an integer specifying the number of layers to be used in the approximation.

- n — This is an integer or a vector of integers specifying the number of nodes along each spacial dimension.

- domain — This contains the domain for the state variables over which the solution is obtained. Domain will be a $2-$element vector in the one-state-variable case and a $2 \times n$ array in the n-state-variable case, with the first row of the array containing the upper values of the domain

and the second row containing the lower values of the domain. If an initializing solution is provided, then the domain associated with that initializing solution can be used by setting domain to an empty array, Float64[].

- lb — This a vector containing the lower bound on the model's variables.

- ub — This is a vector containing the upper bound on the model's variables.

- tol_fix_point_solver — This specifies the tolerance to be used in the inner loop to determine convergence at each solution node.

- tol_variables — This specifies the tolerance to be used in the outer loop to determine convergence of the overall solution.

- maxiters — This is an integer specifying the maximum number of outer-loop iterations before the solution exits.

- solver — Specifies the solver in NLboxsolve.jl, either :nr,:lm, :lm_kyf, or :lm_ar.

An example of a stochastic SmolyakScheme with occasionally binding constraints is:

$S = SmolyakSchemeOBCStoch(ss,chebyshev\_gauss\_lobatto,9,5,11,[0.1\ 30.0;\ -0.1\ 20.0],[-Inf,\ -Inf,\ -Inf],[Inf,\ Inf,\ Inf],1e-8,1e-6,1000,:lm\_ar)$

In the deterministic case the number of quadrature nodes is not needed, i.e.,

$Sdet = SmolyakSchemeOBCDet(ss,chebyshev\_gauss\_lobatto,5,11,[0.1\ 30.0;\ -0.1\ 20.0],[-Inf,\ -Inf,\ -Inf],[Inf,\ Inf,\ Inf],1e-8,1e-6,1000,:lm\_ar)$

PiecewiseLinearSchemes   To obtain piecewise linear solutions, SolveDSGE.jl employs the module PiecewiseLinearApprox, which allows approximations over an arbitrary number of state variables. A stochastic PiecewiseLinearScheme requires the following arguments:

- initial_guess — This will usually be a vector containing the model's steady state. It is used as the initial guess at the solution in the case where an initializing solution is not provided (see the section on model solution below).

- node_number — This gives the number of nodes to be used for each state variable. If there is only one state variables then node_number will be an integer. When there are two or more state variables it will be a vector of integers.

- num_quad_nodes — This is an integer specifying the number of quadrature points used to compute expectations.

- domain — This contains the domain for the state variables over which the solution is obtained. Domain will be a $2-$element vector in the one-state-variable case and a $2 \times n$ array in the n-state-variable case, with the first row of the array containing the upper values of the domain and the second row containing the lower values of the domain. If an initializing solution is provided, then the domain associated with that initializing solution can be used by setting domain to an empty array, Float64[].

- tol_fix_point_solver — This specifies the tolerance to be used in the inner loop to determine convergence at each solution node.

- tol_variables — This specifies the tolerance to be used in the outer loop to determine convergence of the overall solution.

- maxiters — This is an integer specifying the maximum number of outer-loop iterations before the solution exits.

- solver — Specifies the solver in NLsolve.jl, either :newton or :trust_region.

An example of a stochastic PiecewiseLinearScheme is:

$PL = PiecewiseLinearStoch(ss,[21,21],9,[0.1\ 30.0;\ -0.1\ 20.0],1e\text{-}8,1e\text{-}6,1000,:newton)$

In the deterministic case the number of quadrature nodes is not needed, i,e,,

$PLdet = PiecewiseLinearDet(ss,[21,21],[0.1\ 30.0;\ -0.1\ 20.0],1e\text{-}8,1e\text{-}6,1000,:newton)$

PiecewiseLinearSchemes with occasionally binding constraints   A stochastic Piecewise-LinearScheme with occasionally binding constraints requires the following arguments:

- initial_guess — This will usually be a vector containing the model's steady state. It is used as the initial guess at the solution in the case where an initializing solution is not provided (see the section on model solution below).

- node_number — This gives the number of nodes to be used for each state variable. If there is only one state variables then node_number will be an integer. When there are two or more state variables it will be a vector of integers.

- num_quad_nodes — This is an integer specifying the number of quadrature points used to compute expectations.

- domain — This contains the domain for the state variables over which the solution is obtained. Domain will be a 2−element vector in the one-state-variable case and a $2 \times n$ array in the n-state-variable case, with the first row of the array containing the upper values of the domain and the second row containing the lower values of the domain. If an initializing solution is provided, then the domain associated with that initializing solution can be used by setting domain to an empty array, Float64[].

- lb — This a vector containing the lower bound on the model's variables.

- ub — This is a vector containing the upper bound on the model's variables.

- tol_fix_point_solver — This specifies the tolerance to be used in the inner loop to determine convergence at each solution node.

- tol_variables — This specifies the tolerance to be used in the outer loop to determine convergence of the overall solution.

- maxiters — This is an integer specifying the maximum number of outer-loop iterations before the solution exits.

- solver — Specifies the solver in NLboxsolve.jl, either :nr,:lm, :lm_kyf, or :lm_ar.

An example of a stochastic PiecewiseLinearScheme with occasionally binding constraints is:

P = PiecewiseLinearOBCStoch(ss,[21,21],9,[0.1 30.0; -0.1 20.0],[-Inf, -Inf, -Inf],[Inf, Inf, Inf],1e-8,1e-6,1000,:lm_ar)

In the deterministic case the number of quadrature nodes is not needed, i,e,,

Pdet = PiecewiseLinearOBCDet(ss,[21,21],[0.1 30.0; -0.1 20.0],[-Inf, -Inf, -Inf],[Inf, Inf, Inf],1e-8,1e-6,1000,:lm_ar)

4.3   Model solution

Once a SolutionScheme is specified we are in a position to solve the model. In order to do so we use the solve_model() function, which takes either two or three arguments. For a perturbation

solution solve_model() requires two arguments: the model to be solved and the SolutionScheme, as follows:

soln_first_order = solve_model(dsge,P)

soln_second_order = solve_model(dsge,PP)

soln_third_order = solve_model(dsge,PPP)

Alternatively, for a projection solution solve_model() takes either two, three, or four arguments. To provide a concrete example, suppose we wish to solve our model using Chebyshev polynomials. If we want the projection solution to be initialized using the steady state, then solve_model() requires only two arguments: the model to be solved and the SolutionScheme:

soln_chebyshev = solve_model(dsge,C)

If we want the projection solution to be initialized using the third-order perturbation solution, then solve_model() requires three arguments: the model to be solved, the initializing solution, and the SolutionScheme:

soln_chebyshev = solve_model(dsge,soln_third_order,C)

Although this example uses a third-order perturbation as the initializing solution, any solution (first-order, second-order, third-order, fourth-order, Chebyshev, Smolyak, hyperbolic cross, or piecewise linear) can be used.

Finally, the routines for obtaining projection solutions have multi-threaded variants where the final argument in the function is an integer specifying the number of threads to use. For example:

soln_chebyshev = solve_model(dsge,C,4)

soln_chebyshev = solve_model(dsge,soln_third_order,C,4)

would solve the model using 4 threads. Before using these multi-threaded functions you will need to know how many threads are available on your computer (Threads.nthreads()). Note, that there is an overhead to using multi-threading so these multi-threaded functions may not always solve your model more quickly, and it is often the case that better performance can be achieved by not using all available threads.

4.3.1 A comment on third-order perturbation

Sometimes it can be useful to add skewness to the shocks, but this is not easy to do through the model file. If you want your shocks to be skewed, then you can access the third order perturbation solution by calling:

$soln\_third\_order = solve\_third\_order(dsge,PPP,skewness)$

where skewness is a 2D array containing the skewness coefficients. If there is only one shock, then the skewness array is:

$$skewness = E\left[\epsilon_1\epsilon_1\epsilon_1\right].$$

If there are two shocks, then the skewness array is:

$$skewness = E\left[\begin{array}{cccc} \epsilon_1\epsilon_1\epsilon_1 & \epsilon_1\epsilon_1\epsilon_2 & \epsilon_1\epsilon_2\epsilon_1 & \epsilon_1\epsilon_2\epsilon_2 \\ \epsilon_2\epsilon_1\epsilon_1 & \epsilon_2\epsilon_1\epsilon_2 & \epsilon_2\epsilon_2\epsilon_1 & \epsilon_2\epsilon_2\epsilon_2 \end{array}\right].$$

Etc.

4.3.2 Solution structures

When a model is solved the solution is returned in the form of a structure. The exact structure returned depends on the solution method.

First-order perturbation The first-order perturbation solution takes the following form:

$$\begin{aligned} \mathbf{x}_{t+1} &= \mathbf{h_x}\mathbf{x}_t + \mathbf{k}\epsilon_{t+1}, \\ \mathbf{y}_t &= \mathbf{g_x}\mathbf{x}_t. \end{aligned}$$

The solution structure for a stochastic first-order perturbation has the following fields:

- hbar — The steady state of the state variables

- hx — The first-order coefficients in the state-transition equation

- k — The loading matrix on the shocks in the state-transition equation.

- gbar — The steady state of the jump variables

- gx — The first-order coefficients in the jump's equation

- sigma — An identy matrix

- grc — The number of eigenvalues with modulus greater than cutoff.

- Soln_type — Either "determinate", "indeterminate", or "unstable".

The solution to a deterministic model has the same fields as the stochastic solution with the exceptions of $\mathbf{k}$ and sigma.

Second-order perturbation The second-order perturbation solution takes the following form:

$$
\begin{aligned}
\mathbf{x}_{t+1} &= \mathbf{h_x}\mathbf{x}_t + \frac{1}{2}\mathbf{h_{ss}} + \frac{1}{2}\left(\mathbf{I} \otimes \mathbf{x}_t\right)\mathbf{h_{xx}}\left(\mathbf{I} \otimes \mathbf{x}_t\right) + \mathbf{k}\epsilon_{t+1}, \\
\mathbf{y}_t &= \mathbf{g_x}\mathbf{x}_t + \frac{1}{2}\mathbf{g_{ss}} + \frac{1}{2}\left(\mathbf{I} \otimes \mathbf{x}_t\right)\mathbf{g_{xx}}\left(\mathbf{I} \otimes \mathbf{x}_t\right).
\end{aligned}
$$

The solution structure for a stochastic second-order perturbation has the following fields:

- hbar — The steady state of the state variables

- hx — The first-order coefficients in the state-transition equation

- hss — The second-order stochastic adjustment to the mean in the state-transion equation

- hxx — The second-order coefficients in the state-transition equation

- k — The loading matrix on the shocks in the state-transition equation.

- gbar — The steady state of the jump variables

- gx — The first-order coefficients in the jump's equation

- gss — The second-order stochastic adjustment to the mean in the jump's equation

- gxx — The second-order coefficients in the jump's equation

- sigma — An identy matrix

- grc — The number of eigenvalues with modulus greater than cutoff.

- Soln_type — Either "determinate", "indeterminate", or "unstable".

The solution to a deterministic model has the same fields as the stochastic solution with the exceptions of $\mathbf{h_{ss}}$, $\mathbf{k}$, $\mathbf{g_{ss}}$, and sigma.

Third-order perturbation   The third-order perturbation solution takes the following form:

$$
\begin{aligned}
\mathbf{x}_{t+1} &= \mathbf{h_x}\mathbf{x}_t + \frac{1}{2}\mathbf{h_{ss}} + \frac{1}{2}\mathbf{h_{xx}}\left(\mathbf{x}_t \otimes \mathbf{x}_t\right) + \frac{1}{6}\mathbf{h_{sss}} + \frac{3}{6}\mathbf{h_{ssx}}\mathbf{x}_t + \frac{1}{6}\mathbf{h_{xxx}}\left(\mathbf{x}_t \otimes \mathbf{x}_t \otimes \mathbf{x}_t\right) + \mathbf{k}\epsilon_{t+1}, \\
\mathbf{y}_t &= \mathbf{g_x}\mathbf{x}_t + \frac{1}{2}\mathbf{g_{ss}} + \frac{1}{2}\mathbf{g_{xx}}\left(\mathbf{x}_t \otimes \mathbf{x}_t\right) + \frac{1}{6}\mathbf{g_{sss}} + \frac{3}{6}\mathbf{g_{ssx}}\mathbf{x}_t + \frac{1}{6}\mathbf{g_{xxx}}\left(\mathbf{x}_t \otimes \mathbf{x}_t \otimes \mathbf{x}_t\right).
\end{aligned}
$$

The solution structure for a stochastic third-order perturbation has the following fields:

- hbar — The steady state of the state variables

- hx — The first-order coefficients in the state-transition equation

- hss — The second-order stochastic adjustment to the mean in the state-transion equation

- hxx — The second-order coefficients in the state-transition equation

- hsss — The third-order stochastic adjustment for skewness to the mean in the state-transition equation

- hssx — The volatility adjustment to the state-transition equation

- hxxx — The third-order coefficents in the state-transition equation

- k — The loading matrix on the shocks in the state-transition equation.

- gbar — The steady state of the jump variables

- gx — The first-order coefficients in the jump's equation

- gss — The second-order stochastic adjustment to the mean in the jump's equation

- gxx — The second-order coefficients in the jump's equation

- gsss — The third-order stochastic adjustment for skewness to the mean in the jump's equation

- gssx — The volatility adjustment to the jump's equation

- gxxx — The third-order coefficients in the jump's equation

- sigma — An identy matrix

- grc — The number of eigenvalues with modulus greater than cutoff.

- Soln_type — Either "determinate", "indeterminate", or "unstable".

The solution to a deterministic model has the same fields as the stochastic solution with the exceptions of $\mathbf{h_{ss}}$, $\mathbf{h_{sss}}$, $\mathbf{h_{ssx}}$, $\mathbf{k}$, $\mathbf{g_{ss}}$, $\mathbf{g_{sss}}$, $\mathbf{g_{ssx}}$, and sigma.

Fourth-order perturbation   The fourth-order perturbation solution takes the following form:

$$
\begin{aligned}
\mathbf{x}_{t+1} \;=\; & \mathbf{h_x}\mathbf{x}_t + \frac{1}{2}\mathbf{h_{ss}} + \frac{1}{2}\mathbf{h_{xx}}\left(\mathbf{x}_t \otimes \mathbf{x}_t\right) + \frac{1}{6}\mathbf{h_{sss}} + \frac{3}{6}\mathbf{h_{ssx}}\mathbf{x}_t + \frac{1}{6}\mathbf{h_{xxx}}\left(\mathbf{x}_t \otimes \mathbf{x}_t \otimes \mathbf{x}_t\right) \\
& + \frac{1}{24}\mathbf{h_{ssss}} + \frac{6}{24}\mathbf{h_{ssxx}}\left(\mathbf{x}_t \otimes \mathbf{x}_t\right) + \frac{1}{24}\mathbf{h_{xxxx}}\left(\mathbf{x}_t \otimes \mathbf{x}_t \otimes \mathbf{x}_t \otimes \mathbf{x}_t\right) + \mathbf{k}\epsilon_{t+1}, \\
\mathbf{y}_t \;=\; & \mathbf{g_x}\mathbf{x}_t + \frac{1}{2}\mathbf{g_{ss}} + \frac{1}{2}\mathbf{g_{xx}}\left(\mathbf{x}_t \otimes \mathbf{x}_t\right) + \frac{1}{6}\mathbf{g_{sss}} + \frac{3}{6}\mathbf{g_{ssx}}\mathbf{x}_t + \frac{1}{6}\mathbf{g_{xxx}}\left(\mathbf{x}_t \otimes \mathbf{x}_t \otimes \mathbf{x}_t\right) \\
& + \frac{1}{24}\mathbf{g_{ssss}} + \frac{6}{24}\mathbf{g_{ssxx}}\left(\mathbf{x}_t \otimes \mathbf{x}_t\right) + \frac{1}{24}\mathbf{g_{xxxx}}\left(\mathbf{x}_t \otimes \mathbf{x}_t \otimes \mathbf{x}_t \otimes \mathbf{x}_t\right).
\end{aligned}
$$

The solution structure for a stochastic third-order perturbation has the following fields:

- hbar — The steady state of the state variables

- hx — The first-order coefficients in the state-transition equation

- hss — The second-order stochastic adjustment to the mean in the state-transion equation

- hxx — The second-order coefficients in the state-transition equation

- hsss — The third-order stochastic adjustment for skewness to the mean in the state-transition equation

- hssx — The volatility adjustment to the state-transition equation

- hxxx — The third-order coefficents in the state-transition equation

- hssss — The fourth-order stochastic adjustment for skewness to the mean in the state-transition equation

- hssxx — The volatility adjustment to the state-transition equation

- hxxxx — The fourth-order coefficents in the state-transition equation

- k — The loading matrix on the shocks in the state-transition equation.

- gbar — The steady state of the jump variables

- gx — The first-order coefficients in the jump's equation

- gss — The second-order stochastic adjustment to the mean in the jump's equation

- gxx — The second-order coefficients in the jump's equation

- gsss — The third-order stochastic adjustment for skewness to the mean in the jump's equation

- gssx — The volatility adjustment to the jump's equation

- gxxx — The third-order coefficients in the jump's equation

- gssss — The fourth-order stochastic adjustment for skewness to the mean in the jump's equation

- gssxx — The volatility adjustment to the jump's equation

- gxxxx — The fourth-order coefficients in the jump's equation

- sigma — An identy matrix

- grc — The number of eigenvalues with modulus greater than cutoff.

- Soln_type — Either "determinate", "indeterminate", or "unstable".

The solution to a deterministic model has the same fields as the stochastic solution with the exceptions of $\mathbf{h_{ss}}$, $\mathbf{h_{sss}}$, $\mathbf{h_{ssx}}$, $\mathbf{h_{ssss}}$, $\mathbf{h_{ssxx}}$, $\mathbf{k}$, $\mathbf{g_{ss}}$, $\mathbf{g_{sss}}$, $\mathbf{g_{ssx}}$, $\mathbf{g_{ssss}}$, $\mathbf{g_{ssxx}}$ and sigma.

Chebyshev solution   The solution structure for the Chebyshev solution has the following fields:

- variables — A vector of arrays containing the solution for each variable

- weights — A vector of arrays containing the weights for the Chebyshev polynomials

- integrals — An array of arrays containing the scale factors needed for scaled weights

- nodes — A vector of vectors containing the Chebyshev nodes

- order — The order of the Chebyshev polynomials

- domain — The domain for the state variables

- k — Innovation loading matrix

- iteration_count — The number of iterations needed to achieve convergence

- node_generator — The function used to generate the nodes

The solution to a deterministic model has the same fields with the exception of k.

Smolyak solution   The solution structure for the Smolyak solution has the following fields:

- variables — A vector of arrays containing the solution for each variable

- weights: — A vector of vectors containing the weights for the polynomials

- scale_factor — A vector for computing scaled weights

- grid — A matrix containing the Smolyak grid

- multi_index — A matrix containing the multi-index underlying the polynominals

- layer — The number of layers in the approximation

- domain — The domain for the state variables

- k — Innovation loading matrix

- iteration_count — The number of iterations needed to achieve convergence

- node_generator — The function used to generate the nodes

The solution to a deterministic model has the same fields with the exception of k.

Hyperbolic cross solution   The solution structure for the Smolyak solution has the following fields:

- variables — A vector of arrays containing the solution for each variable

- weights: — A vector of vectors containing the weights for the polynomials

- scale_factor — A vector for computing scaled weights

- grid — A matrix containing the Smolyak grid

- multi_index — A matrix containing the multi-index underlying the polynominals

- layer — The number of layers in the approximation

- domain — The domain for the state variables

- k — Innovation loading matrix

- iteration_count — The number of iterations needed to achieve convergence

- node_generator — The function used to generate the nodes

The solution to a deterministic model has the same fields with the exception of k.

Piecewise linear solution   The solution structure for the piecewise linear solution has the following fields:

- variables — A vector of arrays containing the solution for each variable

- nodes — A vector of vectors containing the nodes

- domain — The domain for the state variables

- k — Innovation loading matrix

- iteration_count — The number of iterations needed to achieve convergence

The solution to a deterministic model has the same fields with the exception of k.

5   Post-solution analysis

Once you have solved your model there are many things that you might want to use the solution for. Some of the more obvious things, such as simulating data from the solution and computing impulse response functions have been built into SolveDSGE.jl to make things easier for you.

5.1   Simulation

To simulate data from a model's solution there are two functions: *simulate()* and *ensemble_simulate()*, intended for representative agent models and heterogenous agent models respectively. For the function *simulate()*, the arguments are a model solution, an initial state, and the number of time periods to simulate. An optional final argument is the seed for the random number generator. There is also a method for *simulate()* that imposes the lower bound and the upper bound associated with any occasionally binding constraint. Examples of *simulate()* in action are:

data = simulate(soln,initial_ state,n_ periods)
data = simulate(soln,initial_ state,lb,ub,n_ periods)

where *lb* and *ub* are 1D vectors representing the lower and upper values one each model variable.

As this example makes clear, the simulate function returns a single 2D array, containing simulated outcomes for the each of the model's variables. The *simulate()* function can be applied to both stochastic and deterministic models.

The *ensemble_ simulate()* function is intended to simulate data on a cross-section of different individuals over time. The function provides several methods depending on whether the model has

aggregate shocks as well as idiosyncratic shocks. For the *ensemble_simulate()* function the arguments are: a model solution, an initial state, the number of agents, and the number of time periods to simulate. As previously, there are also methods that impose any lower bound and/or upper bound on each variable, and methods that specify which shocks are aggregate shocks. Examples of *ensemble_simulate()* are:

data = ensemble_simulate(soln,initial_state,n_agents,n_periods)

data = ensemble_simulate(soln,initial_state,lb,ub,n_agents,n_periods)

data = ensemble_simulate(soln,initial_state,agg_shocks,n_agents,n_periods)

data = ensemble_simulate(soln,initial_state,agg_shocks,lb,ub,n_agents,n_periods)

The *ensemble_simulate()* function returns a vector of 2D arrays where the vector has length equal to *n_agents*.

5.2 Impulse response functions

Impulse responses are obtained using the *impulses()* function, which takes three arguments: the model solution, the length of the impulse response function (number of periods), the size of the impulse to apply to each shock, and the number of repetitions to use for the Monte Carlo integration. The method used to compute the impulses draws on Potter (2000). Responses to both a positive and a negative innovation are generated. An optimal final argument is the seed for the random number generator. For a model with two shocks, an example of *impulses()* in use would be:

pos_responses, neg_responses = impulses(soln,n_periods,[1,0],mc_reps)

which applies a one standard deviation impulse to the first shock and no impulse to the second shock. For the nonlinear solutions (second-order perturbation, third-order perturbation, fourth-order perturbation, and the projection-based solutions) the initial state is "integrated-out" via a Monte Carlo that averages over draws taken from the unconditional distribution of the state variables. At this stage in the package's development, the impulses need to be computed one perturbation at a time; this will probably change at some point.

An alternative method allows an initial state to be specified (in which case the initial state is not integrated out):

pos_responses, neg_responses = impulses(soln,n_periods,initial_state,[1,0],mc_reps)

5.3 PDFs and CDFs

SolveDSGE.jl contains functions for approximating the probability density function and the cumulative distribution function of a variable, where the approximation is based on Fourier series (Kronmal and Tarter, 1968). Most of the functionality relates to the univariate case, but some functionality is included to approximate and evaluate the PDF in the multivariate case.

5.3.1 Univariate

To approximate the probability density function and evaluate the approximated function at a point the function is:

$f = approximate\_density(sample,point,order,lower\_bound,upper\_bound)$

where *sample* is a vector of data, *point* is the value at which the PDF is evaluated, *order* is the order of the Fourier series approximation, and *lower_bound* and *upper_bound* specify the support over which the PDF is constructed. If an approximation of the entire PDF is sought, then the function is:

$nodesf, f = approximate\_density(sample,order,lower\_bound,upper\_bound)$

Similarly, the cumulative distribution function is approximated and evaluated at a point using the function:

$F = approximate\_distribution(sample,point,order,lower\_bound,upper\_bound)$

where *sample* is a vector of data, *point* is the value at which the CDF is evaluated, *order* is the order of the Fourier series approximation, and *lower_bound* and *upper_bound* specify the support over which the CDF is constructed. If an approximation of the entire CDF is sought, then the function is:

$nodesF, F = approximate\_distribution(sample,order,lower\_bound,upper\_bound)$

5.4 Decision rules, laws-of-motion

The model's solution can be used to construct functions of the state that govern the behavior of the jump variables (decision rules) and the future outcomes for the state variables (laws-of-motion or state-transition functions). These functions are produced by calling the *state_space_eqm()* function with the model as the only input, i.e.,

$eqm\_dyn = state\_space\_eqm(soln\_third\_order)$

or

$eqm\_dyn = state\_space\_eqm(solna)$

Then the decision rule function and the state-transition function are accessed via

$dec\_rule = eqm.dyn.g$

$state\_trans = eqm\_dyn.h$

The former of these (dec_rule) is a function of the state (a vector) while the latter ($state\_trans$) is a function of the state (a vector) for deterministic models and a function of the state (a vector) and the shocks (a vector) for stochastic models.

5.5 Evaluating accuracy

SolveDSGE.jl offers three ways to think about solution accuracy. The first way is to compare two solutions and assess the magnitudes of any differences. This facilitates an adaptive approach to approximation and it allows robustness of the solution to be assessed across approximation schemes. When comparing two models, SolveDSGE.jl looks at the predicted values for the jump variables, returning the maximum absolute difference for each jump variable found for a random sample of $100,000$ realizations of the state variables. We compare two solution according to:

$errors = compare\_solutions(solna,solnb,domain,seed)$

where *solna* and *solnb* are the two solutions to be compared, *domain* is the domain for the state variables over which the comparison takes place, and *seed* is an optional argument that sets the seed for the random number generator.

The second way of assessing solution accuracy is through traditional Euler-equation errors. In the case of a perturbation solution, we compute the Euler-equation errors through:

$e\_errors,\ states = euler\_errors(dsge,soln\_first\_order,domain,ndraws,seed)$

and in the case of a projection solution, through:

$e\_errors,\ states = euler\_errors(dsge,solna,ndraws,seed)$

In each case, e_errors is a 2D matrix containing the Euler-errors, *dsge* is the model, *ndraws* is an integer for the number of random points in the domain to be analyzed, and *seed* is an (optional)

seed for the random number generator. For a perturbation solution, the domain to be used needs to be supplied, while for a projection solution the domain is taken from that used to solve the model. In the case of the stochastic growth model presented earlier, approximation takes place in only one equation (the consumption Euler-equation) so e_errors is a $1 \times ndraws$ matrix. $states$ is a matrix containing the points in the domain that were randomly chosen to be analyzed.

The final way of assessing solution accuracy is through a basic Den-Haan and Marcet statistic. The arguments for the $den\_haan\_marcet()$ function are: a model, the models solution, and the steady state; an initial seed is a final optional argument.

$$dhm = den\_haan\_marcet(dsge,soln,steady\_state)$$

The $den\_haan\_marcet()$ function performs 1000 simulations of 3000 observations and returns the 1%, the 5%, and the 10% statistics, and the number of degrees of freedom (you need to look up the Chi-square critical values yourself).

6   Examples

The Github directory includes an examples subfolder containing four examples. Depending on your model and application, these examples might provide you with a helpful starting point.

1. A deterministic growth model.

2. A stochastic growth model.

3. A New Keynesian DSGE model with the zero lower bound.

4. The Aiyagari (1994) model.

References

[1] Aiyagari, R., (1994), "Uninsured Idiosyncratic Risk and Aggregate Savings", *Quarterly Journal of Economics*, 109, 3, pp.659–684.

[2] Andreasen, M., Fernández-Villaverde, J., and J. Rubio-Ramirez, (2017), "The Pruned State-Space System for Non-Linear DSGE Models: Theory and Empirical Applications", *Review of Economic Studies*, 0, pp. 1—49.

[3] Binning, A., (2013), "Third-Order Approximation of Dynamic Models Without the Use of Tensors", *Norges Bank Working Paper* 2013–13.

[4] Gomme, P., and P. Klein, (2011), "Second-Order Approximation of Dynamic Models Without the Use of Tensors", *Journal of Economic Dynamics and Control*, 35, pp. 604—615.

[5] Judd, K. (1992), "Projection Methods for Solving Aggregate Growth Models", *Journal of Economic Theory*, 58, pp.410—452.

[6] Judd, K., Maliar, L., Maliar, S., and R. Valero, (2014), "Smolyak Method for Solving Dynamic Economic Models: Lagrange Interpolation, Anisotropic Grid and Adaptive Domain", *Journal of Economic Dynamics and Control*, 44, pp. 92—123.

[7] Judd, K., Maliar, L., Maliar, S., and I. Tsener, (2017), "How to Solve Dynamic Stochastic Models Computing Expectations just Once", *Quantitative Economics*, 8, pp.851—893.

[8] Klein, P., (2000), "Using the Generalized Schur Form to Solve a Multivariate Linear Rational Expectations Model", *Journal of Economic Dynamics and Control*, 24, pp. 1405—1423.

[9] Kronmal, R., and M. Tarter, (1968), "The Estimation of Probability Densities and Cumulatives by Fourier Series Methods", *Journal of the American Statistical Association*, 63, 323, pp.925–952.

[10] Levintal, O., (2017), "Fifth-Order Perturbation Solution to DSGE models", *Journal of Economic Dynamics and Control*, 80, pp. 1–16.

[11] Potter, S., (2000), "Nonlinear Impulse Response Functions", *Journal of Economic Dynamics and Control*, 24, pp. 1425—1446.