DSGE Code Guide

Chris Cotton

December 18, 2019

1 Introduction

I have written some DSGE codes in Python. They should do the following:

- Generate a basic DSGE setup in discrete (and hopefully soon continuous) time.
- Apply basic DSGE solution methods.
- Apply a solution method similar to Occbin which allows for inequality constraints.
- Apply a basic DSGE solution method but allow for a variable to be picked by a shock i.e. allow for nominal interest rates to be fixed for a number of periods before adopting a normal rule.
- Some interplay with Dynare - it can input Dynare form equations and output a basic Dynare file.

The codes are located at `nopubliclocation/dsge-perturbation/`. Examples of how to use these are located at `nopubliclocation/dsge-perturbation-examples`.

2 How I Run Models with a First-Order Approximation

I tend to break the code into two parts:

- Basic Setup
- Actual Run

In the basic setup step, I tend to do the following:

- Function to solve for the steady state given parameters. Input: paramssdict. Output: paramssdict with steady states included.
- Function to create an inputdict without doing anything else. I like to allow for both log-linearized equations and the full model (but with variables in log form) which should produce identical results if done correctly. This function then has an optional argument loglineareqs = True which determines whether to use the log-linearized equations or the full model with variables in log form.
- Check model function: This verifies the steady state holds for the full model and then verifies the coefficients are the same for the full and log-linearized models.

I give basic examples of how I typically set up respectively a simple RBC and simple NK model in `nopubliclocation/dsge-perturbation-examples/dsgesetup/rbc_simple.py` and `nopubliclocation/dsge-perturbation-examples/dsgesetup/nk_simple.py`. I typically add the parameters in a separate function which allows me to adjust them. The code verifies that the models satisfy the steady state of the model. I also typically specify both the log-linearized and normal model (which I convert to logs). These models should produce the same results so this serves as an additional check that I have not misspecified the equations.

ACTUAL RUN COMPLETE

3 Dynare 2 Python COMPLETE

These functions can be found in `nopubliclocation/dsge-perturbation/python2dynare_func.py`. Dynare 2 Python conversion before doing the main Python functions. But this is a bit incomplete.

3.1 python2dynare_inputdict(inputdict)

This function converts parts of dynare inputted into the equation into the Schmitt-Grohe Uribe format that I use. However, I'm not currently using it so it may not work. I would apply it before applying 'getmodel' and doing any of the other conversion.

4 Parse Discrete DSGE Functions

These functions can be found in `nopubliclocation/dsge-perturbation/python2dynare_func.py`. They adjust a set of inputs into a form that can be used as a DSGE model. I could just input the model in the correct form to begin with but using this function can save time and also includes important checks i.e. verifying there are not misused variables/parameters and that the steady state is correct.

4.1 getmodel_inputdict(inputdict)

The main function that converts a list of string equations into a DSGE setup on which different methods can be performed is 'getmodel_inputdict' in `nopubliclocation/dsge-perturbation/dsgesetup_func.py`. This also does various checks to ensure the inputs make sense. An example of the usage of this function is found at `nopubliclocation/dsge-perturbation-examples/dsgesetup/rbc_simple.py`.

The necessary input arguments for 'getmodel' are:

- 'equations': A list of equations i.e. ['x + y = 1', 'x - y = 0']. By default, controls and states should only be specified at time t and time $t+1$. x_t is represented by x while x_{t+1} is represented by x_p by default.

I also nearly always need to specify:

- 'states': A list of states.
- 'controls': A list of controls.
- 'paramssdict': A dictionary of any parameters specified in the equations. Parameters are fixed and do not vary over time. Optionally, I can also include the steady state of variables in this dictionary as well (there is no reason not to do this beyond keeping things separate for neatness).

The commonly used optional input arguments for 'getmodel' are:

- 'varssdict': A dictionary with a steady state for each control and state. I need to specify this unless I don't need to include a steady state unless I specify 'skipsscheck': True and there are no steady state variables.
- 'loglineareqs': Specify that the inputted equations are in loglinear form.
- 'logvars': Input: a list of variables to be converted into log form in the outputted equations. I shouldn't use this with loglineareqs = True. Default: No variables.
- 'shocks': A list of shocks that are included in the equations. I do not need to specify the fact that $\mathbb{E}_t[\epsilon_{t+1}] = 0$ in the list of equations where ϵ is a shock.

Commonly used optional arguments which are used in later functions but not actually in getmodel:

- mainvars: Specify which variables are the most important i.e. which show graphically.
- mainvarnames: Specify names to use in graphs etc. for most important variables.

Optional arguments that work fine with their defaults are:

- 'ssending1': This specifies the ending I use in varssdict for the steady state. So if 'ssending' is '_boo' then I would specify varssdict['x_boo'] = 1 if the steady state of x is 1. The default is '' i.e. varssdict['x'] = 1.
- 'ssending2': If I want to refer to the steady state of variable 'x' in the equations then I add ssending2 to the end. Default '_ss'.
- 'evalequals': Convert equations to remove any equals terms i.e. replace $x = y$ with $x - y = 0$. Default: True.
- 'skipsscheck': Skip the steady state check. This can be useful with a large model. Default: False.

The output arguments are:

- Equations without parameters
- Equations and variables adjusted for shocks and logs
- Position dicts

Optional Variables Optional controls, states and shocks: I have options to add optional variables into the DSGE model. This allows me to include variables only when I want to examine their outcomes and otherwise exclude them. To do this, I should not include the definitions of these variables in the DSGE directly but instead include them in optional dictionaries.

For a control `x = y + z`, I would then define the dictionary `optionalcontroldict['x'] = y + x`. If I then do not include `x` in the specified controls, this will replace all occurrences of `x` in equations, optionalcontroldict and optionalstatedict with `y + z`. Similarly, it will replace any future occurrences `x_p` with `y_p+z_p`. This future substitution wouldn't work if x already contained a future variable i.e. `optionalcontroldict['x']=y_p+z`. In this case, I'd need to define specifically what I should replace `y_p` with using a second dictionary futureoptionaldict i.e. `futureoptionaldict['x']=yp+z`. If the optional control is included in controls then the definition of it is just added to the list of equations as normal. An example of this is given in addoptionalcontrols_test() in `nopubliclocation/dsge-perturbation/dsgesetup_func.py`.

For states, I can define an optionalstatedict i.e. if I would specify `a_p=RHO*a+epsilon` then I would write `optionalstatedict['a_p']=RHO*a+epsilon`. Then if I include `a` in the list of states, this line is added to the equations while if I do not `a` and `a_p` are replaced with the steady state of `a`. An example of this is given in addoptionalstates_test() in `nopubliclocation/dsge-perturbation/dsgesetup_func.py`. Similarly, for controls, I can define an optionalshocklist. If a variable `e` is in optionalshocklist and shocks then it is added to shocks in the usual way. If it is in optionalshocklist but not in shocks then `e` and `e_p` are replaced with the steady state of `e` (which is zero).

4.2 getshocks_inputdict(inputdict)

This defines the standard deviation of shocks and specifies an actual path for those shocks if desired.

Optional argument for inputdict:

- 'shockdict': A dictionary containing information about shocks.

Potential arguments for shockdict:

- 'sd': shockdict['sd'] is a dictionary of different standard deviations for shocks i.e. shockdict['sd'] = 'epsilon_a': 0.5. If I don't specify 'sd' then all standard deviations of 'shocks' are set to be 1.
- 'path': An actual path for shocks. This should take the form numberofperiods x numberofshocks.
- 'pathsimperiods': If path is not specified, I can instead specify the number of periods that I want to simulate over and then generate a simulation of shocks in the same format as 'path' taking into account the standard deviations.
- 'pathsimperiods_noshockend': This needs to be specified with 'pathsimperiods'. It means that there are 'pathsimperiods' of shocks followed by 'pathsimperiods_noshockend' of no shocks in 'path'.

5 Get Shocks

5.1 getshocksddict

Generate dictionary of shock standard deviations.

Necessary inputs:

- 'shocks': List of shocks

Optional inputs:

- 'shocksddict': dictionary of shock standard deviations. Allows me to input a partly completed dictionary. Default: No dictionary (in which case every shock is specified to have a standard deviation of 1).

I specify the standard deviations are 1 for each variable in 'shocks' unless the standard deviation already exists in 'shockssddict'.

5.2 getshocksddict_inputdict

Just applies getshocksddict to an inputdict.

5.3 getshockpath

Simulate paths for shocks. Shockpath is returned as a 2-dimensional array of form numperiods x numshocks.

Necessary inputs:

- 'pathsimperiods': Number of periods to simulate for
- 'shocks': List of shocks
- 'shocksddict': Shock standard deviation dict

Optional inputs:

- 'pathsimperiods_noshockend': Number of extra periods without any shocks where shocks are zero. Default: 0

I specify the standard deviations are 1 for each variable in 'shocks' unless the standard deviation already exists in 'shockssddict'.

5.4 getshockpath_inputdict

Just applies getshocksddict to an inputdict.

6 Python 2 Dynare

These functions can be found in `nopubliclocation/dsge-perturbation/python2dynare_func.py`. They convert a model written in Python into one that can be run in Dynare.

6.1 python2dynare

Function to convert from Python to Dynare file. Start from a template file loadfilename and then adjust it based upon the arguments inputted into the function.

Necessary arguments:

- equations_string: A list of equations in string form.
- paramssdict: A list of parameters

Main optional arguments:

- dynarevariables: Only include this if the equations are in dynare format.

- states, controls: Lists of states and controls. Only include this if the equations are not in Dynare format. Note that in this case, the variables used are states plus controls while we also define a predetermined_variables variable which is just states.
- varssdict: Steady state for variables
- shocks: A list of the shocks
- shocksddict: Include standard deviation information in mod file
- shockpath: Include paths for shocks in mod file. Need to be in form T x numvars.
- loadfilename: Where load Dynare mod file from. Note that the default is `nopubliclocation/dsge-perturbation/dynare/template.mod`.
- savefilename: Where save Dynare mod file
- simulation: The command used to run the simulation in Dynare. Typically, this is stoch_simul or simul.

Other rare optional arguments:

- ssending1
- ssending2
- futureending

Note that at least one of shocksddict and shockpath must be defined if there are a nonzero number of shocks.

6.2 python2dynare_inputdict

Function to run python2dynare using a standard dictionary input. An example of this function to generate a Dynare code for stochastic simulation is given in `nopubliclocation/dsge-perturbation-examples/python2dynare_input/stochsimul.py`. Examples of how to do a non-stochastic simulation with and without specifying a path for shocks are given respectively in `nopubliclocation/dsge-perturbation-examples/python2dynare_input/simul_shockpath.py` and `nopubliclocation/dsge-perturbation-examples/python2dynare_input/simul_shockdict.py`. An example of this function to generate a Dynare code for an extended path simulation is given in `nopubliclocation/dsge-perturbation-examples/python2dynare_input/extendedpath.py`.

Necessary arguments:

- inputdict

Options within inputdict:

- 'python2dynare_loadfilename': Specify filename to load dynare mod file from
- 'python2dynare_savefilename': Specify where to save dynare mod file
- 'python2dynare_nofilename': Specify to not return an error if there is no filename specified
- 'python2dynare_simulation': Command for Dynare simulation
- 'python2dynare_noparams': Remove all parameters from Dynare file (to run quickly)

- 'python2dynare_usedparams': Only include parameters that are used in the Dynare file

Note that if 'shockpath' is specified in inputdict then I include it as an option in python2dynare (so I include the shock paths in the mod file). If not and 'shockdict' is specified then I include it as an option (so I include shock sds in the mod file). Otherwise, I do not either in the mod file.

6.3 rundynare_inputdict

This function runs a Dynare file from inputdict. This should be run after generating a Dynare from Python using python2dynare_inputdict (since this function uses the python2dynare_savefilename argument in inputdict. An example of the use of this function is given in `nopubliclocation/dsge-perturbation-examples/python2dynare_input/simul_shockpath.py`.

Options:

- 'runwithoctave': By default, Dynare runs with Matlab. To run it with Octave, either specify `inputdict['runwithoctave'` save a file in this project at `dsge-perturbation/paths/rundynarewithoctave.txt`. The file does not need to contain any text.
- 'dynarepath': To specify the path where Matlab/Octave can find Dynare, specify `inputdict['dynarepath']=path`or save a file containing the path at `dsge-perturbation/paths/dynarepath.txt`.

6.4 getirfs_dynare

This function generates IRFs from the output created by running a Dynare file. The only input argument is 'dynarefilename' which is the path the Dynare file that was run.

6.5 getirfs_dynare_inputdict

This function runs getirfs_dynare but with an inputdict argument. An example is given at `nopubliclocation/dsge-perturbation-examples/python2dynare/simul_shockpath.py`.

7 DSGE Differentiation

These functions can be found in `nopubliclocation/dsge-perturbation/dsgediff_func.py`. They convert a model written in Python into one that can be run in Dynare.

7.1 dsgeanalysisdiff/dsgeanalysisdiff_split

Differentiate a sets of equations with respect to states, shocks and controls. In dsgeanalysisdiff, I differentiate the states and shocks together. In dsgeanalysisdiff_split, I differentiate the states and shocks separately. I can merge the state and shock derivatives using convertsplitxe and I can split them using convertjoinxe. An example is given at `nopubliclocation/dsge-perturbation-examples/dsgediff/simple_analysisdiff.py`.

7.2 getnfxenfy_inputdict

Input an inputdict which takes elements generated by getmodel. The function then computes the numerical values for the derivatives with respect to states/controls now and in the future evaluated at the steady state.

Options specifically for this function:

- 'fxefy_cancelparams': Uses equations without parameters to make the numerical evaluation easier. By default, True.
- 'fxefy_subquick': Replace the sympy variables using a non-lambdify function based upon just replacing sympy variables one at a time. By default: False.
- 'fxefy_substringsympy': Replace the sympy variables using a function that converts the matrix to a string and then does a string replace. Quicker but there is a possibility for error. By default: False.

Note that if 'loglineareqs' is defined as True then the function does not do any numerical evaluation since it should not be necessary (as the equations were linear before differentiation). By default, the numerical evaluation is done by converting the matrices to a function using lambdify and then inputting the values into the function (the lambdify may fail for earlier values of Python in which case either fxefy_subquick or fxefy_substringsympy should be specified to be True.

7.3 Partial Parameters Functions

These functions allow me firstly to specify some parameters and partially solve for the derivatives of a linear system and then secondly to input the rest of the parameters and quickly solve for the full system. This is useful when I need to solve a system of equations many times for different parameters and I want to save time. An example is given at `nopubliclocation/dsge-perturbation-examples/dsgediff/partialparams.py`.

The typical way I do this is to specify no parameters initially and then derive functions of the parameters for each of the fxe, fxep, fy, fyp matrices. However, I may be able to save time by specifying some of the parameters initially and replacing those parameters in the equations (parameters that I'm not changing across runs). Or I can even replace some of the steady states of variables. How to do this can be seen in the example script.

I allow for the possibility of returning the full inputdict or just the solved derivatives.

7.4 checksame_inputdict

This function verifies that the inputdict for two functions produce identical derivatives with respect to states and controls in the present and future. I use this to verify that non-linearized and linearized versions of equations are the same. Examples of this function are given in `nopubliclocation/dsge-perturbation-examples/dsgesetup/rbc_simple.py` and `nopubliclocation/dsge-perturbation-examples/dsgesetup/nk_simple.py`.

A key requirement of this function is that the linearized and non-linearized models should be of the same equations. No substitutions of equations should be made and equations should be in the same order.

8 Blanchard-Kahn Policy Function

These functions can be found in `nopubliclocation/dsge-perturbation/dsge_bkdiscrete_func.py`. They convert a model written in Python into one that can be run in Dynare.

8.1 polfunc_inputdict

This function generates a policy function from elements created by getmodel. An example of this function is given in `nopubliclocation/dsge-perturbation-examples/dsge_bkdiscrete/simple_polfunc.py`. It first applies getnfxenfy_inputdict and then converts these into a policy function using a Cholesky decomposition in a similar method to codes by Schmitt-Grohe and Uribe.

Options specifically for this function:

- 'gxhxraise': Raise an error if the conversion to policy functions fails. By default: True.

8.2 interpretpolfunc_inputdict

This function prints the policy function and outputs IRFs. I would also like it to add method of moments and matrices of correlations but I've not finished this yet.

Inputs:

- irfshocks: Shocks or states to investigate irfs for. Default: shocks.
- irfshockdict: Size of shock to use in irf. Default: 1 for each shock.
- showirfs: Which IRFs to show. Default: irfshocks.
- irf_T: Length of IRF to use. Default: 40 periods.
- savefolderlatexname: If savefolder is defined, I produce a tex file during this function. The function calls graphs in savefolder. To refer to savefolder in the texfile, I use savefolderlatexname. By default, this is the absolute path to savefolder but I can change this so the texfile still runs even if I run the texfile from a different absolute path.

Inputdict inputs i.e. inputs that should already be in inputdict:

- gx and hx: Policy functions
- stateshockcontroldict: Where variables are positioned

Outputs:

8.3 discretelineardsgefull

This implements the full DSGE conversion. An example of this function is given in `nopubliclocation/dsge-perturbation-examples/dsge_bkdiscrete/discretelineardsgefull.py`.

It gets the inputdict model, adds a Dynare version of the model, computes policy functions and adds IRFs etc.

8.4 irfmultiplemodels

This runs a model for different parameter values and plots the impulse responses to a given shock for given variables for each of the parameters. An example of this function is given in `nopubliclocation/dsge-perturbation-examples/dsge_bkdiscrete/irfmultiplemodels.py`.

Necessary arguments:

- Parameter list: This is necessary to include a legend in the IRF
- Inputdicts i.e. with the parameter values already included
- Variables to shown in the IRFs
- Shock variable

Optional arguments:

- T = 40: Number of periods to plot IRF for
- shocksize = 1: Size of shock
- plotnames: Names of variables to use as a variable title for each graph in the IRF plots
- graphswithlegend = [0]: The graphs to include the legend for. By default, use the first graph i.e. the first variable in the list of variables.
- pltsavename: Path for where to save graph. If not specified, the graph is displayed instead.

9 Simulate Linear Approximation

9.1 simpathlinear

This function simulates out the path of an economy given a set of shocks. I need to have already computed the linear policy function gx, hx to use this simluation method. An example is given at `nopubliclocation/dsge-perturbation-examples/simlineardsge/simple_sim.py`.

10 Regime Change

The code used to run these functions is given at `nopubliclocation/dsge-perturbation/regime_func.py`.

10.1 regimechange

Solve out for a DSGE model with multiple regimes. I need to specify both the DSGE models for each of the regimes and a path for when the regimes apply. An example can be found at `nopubliclocation/dsge-perturbation-examples/regimes/zlbsolve_func.py`.

The solution method used to solve for regime changes is given at `www.cdcotton.com/review/macro-technique/technique-dsge/regime.html`.

Necessary arguments:

- List of inputdicts for each regime. I should have run getmodel_inputdict on each of these regimes.
- List of regimes: Regime 0 corresponds to the first inputdict; regime 1 corresponds to the second inputdict etc. So I specify something like [0, 1, 0, 2, 0] - this would imply the first regime applies at times 1, 3, 5, the second regime applies at time 2 and the third regime applies at time 4.

Optional arguments:

- irf: If True, generate IRFs.
- Xm1: Initial value for the states.
- epsilon0:

11 Occbin

The code used to run these functions is given at `nopubliclocation/dsge-perturbation/calloccbin_func.py`.

11.1 calloccbin_oneconstraint

Calls a simple version of the Matlab function Occbin (developed by Guerrieri and Iacoviello) through my Python DSGE codes. Occbin works by running Dynare on two mod files with and without an occasionally binding function. My function converts a model generated through my Python DSGE codes to Dynare so that it can be used with Occbin. An example can be found at `nopubliclocation/dsge-perturbation-examples/regimes/zlbsolve_func.py`.

Main inputs:

- inputdict_nozlb: The inputdict without any constraints. getmodel_inputdict does not need to be applied to the inputdict before running the function (it is applied during the function run).
- inputdict_zlb: The inputdict with constraints. getmodel_inputdict does not need to be applied to the inputdict before running the function (it is applied during the function run).
- slackconstraint: A string which returns True when the constraint is slack and False otherwise. I input this into Occbin and Occbin replaces the variables with the values each period to determine whether the constraint binds or not.
- shocks: A numpy array of shocks in the form T x numshocks.
- folder: Folder where I save the results.

Other options:

- occbinpath: Where Occbin is saved. Default: None.
- deletedir: Whether I should delete the directory from any previous runs. Default: True.
- simperiods: The number of periods of the simulation. This can be longer than the periods of shocks specified.
- run: Whether I run the function or not. Default: False.
- irf: Whether I produce IRFs. For this to work, I also require run = True. Default: False.

Note that I also need to specify the Dynare path in occbin/setpathdynare.m for Occbin to work.

12 My Occbin

The code used to run these functions is given at `nopubliclocation/dsge-perturbation/myoccbin_func.py`.

12.1 myoccbin

Calls my version of Occbin which allows for occasionally binding constraints. An example can be found at `nopubliclocation/dsge-perturbation-examples/regimes/zlbsolve_func.py`.

The main differences with the original Occbin are:

- The DSGE models are run through my Python codes rather than Dynare.
- I use my own code to run the Occbin steps.
- I allow for alternative regime updating methods.

Main inputs:

- inputdict_nobind: The inputdict without any constraints. getmodel_inputdict does not need to be applied to the inputdict before running the function (it is applied during the function run).
- inputdict_bind: The inputdict with constraints. getmodel_inputdict does not need to be applied to the inputdict before running the function (it is applied during the function run).
- slackconstraint: A string which returns True when the constraint is slack and False otherwise. I input this into Occbin and Occbin replaces the variables with the values each period to determine whether the constraint binds or not.
- shockpath: A numpy array of shocks in the form T x numshocks.

Other options:

- savefolder: Folder where I save the results. Default: None.
- replacedir: Replace the folder when I run it. Default: True.
- solverperiods: How many periods I go forward when I solve for each period (equivalent to solverperiods in extended_path in DYnare). Default: 100.
- printdetails: Print a set of details as I run the iteration. Default: False.
- irf: Whether I produce IRFs. For this to work, I also require run = True. Default: False.
- printvars: Print the values of the values during the run. Default: False.
- printprobbind: Print the probability that the constraint bound during the run. Deafult: False.
- regimeupdatefunc: See below.

regimeupdatefunc: I allow for alternative regime updating methods. Here are the methods:

- Occbin: Iteration 0 guess that never bind. If incorrect, set that the periods when the constraint bound in iteration 0 are when it binds in iteration 1. Keep doing this until convergence (or it becomes clear that convergence is not possible).

- Bind until not: Iteration 0 guess that never bind. If incorrect, set that in iteration 1 the constraint binds in the first period but not afterwards. If incorrect, set that in iteration 2 the constraint binds in the first two periods but not afterwards. This might make sense with monetary policy since if the central bank expects to hit the ZLB they may lower the nominal interest rate immediately.
- Bind until not and save policy function: One part of the Occbin solution method that can take a while to implement is computing the policy functions for a given regime (since it can involve computing a lot of potentially large inverse matrices). To get around this, we can see that in the bind until not case the number of potential regime combinations that we consider equals only solverperiods. So we can save the policy function as we go and use it. It is also worth noting that we can speed up the computation of the policy functions by taking into account that if the constraint binds for n periods this will be the same as when it binds for $n-1$ periods except with one additional constraint at the end. Using these methods, we can potentially save a lot of time relative to Occbin or the basic bind until not method.

13 Continuous Time DSGE Modeling

The code used to run these functions is given at `nopubliclocation/dsge-perturbation/dsge_continuous_func.py`.

I try to set up this code in a similar way to my discrete time modeling code. There is a basic function to parse a system of equations (getmodel_continuous_inputdict) which is like the discrete time function (getmodel_inputdict). There is a code to verify two systems are the same (checksame_inputdict_cont) which is like the discrete time function (checksame_inputdict). There is a function to do a standard computation of policy functions and get IRFs (continuouslineardsgefull) which is like the discrete time function (discretelineardsgefull).

Unlike in the discrete time case, I don't allow for shocks. Instead, I only allow for shocks to be analyzed through states e.g. to see the impulse response as the economy returns to normal following a loss of capital.

14 To Do

- Fix Dynare 2 Python
- Add:
  - Method of Moments and correlations
  - Matrix of correlations
  - Coefficients of autocorrelation
- Fix continuous time