Matlab Toolkit for Solving Macroeconomic Models using Value Function Iteration

Robert Kirkby
Victoria University of Wellington

February 12, 2015

Abstract

This document explains how to use this Toolkit for solving Value function problems in Matlab. The toolkit is written to be applicable to as wide a range of problems as possible. It is based on value function iteration on a discrete state space and utilizes parallelization on the GPU.

The toolkit is available for download from my website robertdkirkby.com.

If you want to get straight into using the Toolkit, look briefly at Sections 2 and 3, and then go to Examples F.1 and F.2 on how to solve the Stochastic Neo-Classical Growth Model and a Basic Real Business Cycle Model respectively. You can also check out the Julia notebooks on my website. (As a long term goal I want to shift this toolkit over to Julia. The code to use the toolkit looks almost the same in Julia and Matlab.)

As a rough guide to see some more complicated models that you can solve with this toolkit are see my paper, joint with Javier Díaz-Giménez and Josep Pijoan-Mas, on Flat-Tax Reforms (draft is available on my website). The method of moments estimation, and calculating the transition paths of this general equilibrium heterogeneous agent model, are some of the more advanced capabilities of this toolkit.
Note: This document undergoes periodic revision.

Contents

The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was secretary of Defense, and he actually had a pathological fear and hatred of the word research. Im not using the term lightly; Im using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term research in his presence. You can imagine how he felt, then, about the term mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose?
— Richard Bellman, on the origin of his term dynamic programming (another name for value function iteration) (1984)

1 Introduction

Economists often have reason to solve value function problems. This Matlab Toolkit is intended to make doing so easy. It is likely to be faster than value function iteration codes written by a beginner. It will obviously never be as fast as codes written by someone with substantial experience — there are always aspects of a specific problem that can be exploited to produce faster codes. The Toolkit does take advantage of parallelization on the GPU (without requiring any understanding by the user of how and why this this works). This Toolkit, and value function iteration generally, is most useful the first-order conditions of the problem are not both necessary and sufficient, and so methods like perturbation are not usable; ie. it is most useful when your value function problem is not the kind that can be solved by Dynare.

The value function iteration commands in this toolkit are designed to be easy to use. In the case of inifinite horizon value functions the toolkit require you to define as inputs the return function, the grids, the transition matrix for the exogenous shocks, and the discount factor. The toolkit then does all of the work of solving the value function iteration problem for you and as outputs gives the value function and the optimal policy function. In particular the toolkit can handle any number of variables (speed and memory permitting). Behind the scenes the Toolkit takes advantage of Howards improvement and parallelization on the GPU.

There are many commands in the toolkit that have not yet been documented; if you are feeling adventurous feel free to browse around the contents of the Toolkit. [1]

In the near future the Toolkit will also be available from my github:
https://github.com/robertdkirkby/VFIToolkit-matlab

1.1 Why use value function iteration?

While slower than many other numerical methods it has a number of advantages. The main one is their widespread applicability/robustness. Value function iteration can be used to solve problems in which the first-order conditions are not necessary and sufficient. Economic examples include: when borrowing constraints bind, buying durable goods (large indivisible goods), and means-tested

[1] Among other things there are finite-horizon value function codes, and codes for solving general eqm heterogeneous agent models; but all need improving/cleaning/documenting and many have not yet been adapted to take advantage of the GPU.

benefits.[2].

Their other main strength is their accuracy (Aruoba, Fernandez-Villaverde, and Rubio-Ramirez, 2006). The cost of using value function iteration is that it is slower than other numerical methods, although parallelization on the GPU has substantially reduced this penalty Aldrich, Fernandez-Villaverde, Gallant, and Rubio-Ramirez (2011) If there are many variables or large grids then the code can be very slow, or even that the matrices (in particular the return fn matrix) may simply become to large to be held in memory.

From the theoretical perspective they also have the advantage that their convergence properties, including dealing with numerical errors, are well understood.

A further possible disadvantage is that they are not always easy to implement with state space veriables that are non-stationary.

1.2 Examples and Classic Papers

To provide examples of how to use the toolkit in practice a number of examples (model and code) are provided. Going through these provides probably the easiest way to learn to use the toolkit (while providing a bonus lesson in macroeconomics ;). Some of these are chosen as they represent standard models, others are chosen as they are classic papers. This list of classic papers, as well as demonstrating the abilities and use of the toolkit, can also provide a handy way to figure out which 'Case' of the toolkit you need to use; just look for a classic paper in the list of examples which contains a model with a framework similar to that you wish to implement.

1.3 What Can The Toolkit Do?

The following provides a rough listing of the main command types, although for anyone familiar with the literature the best guide as to what the toolkit can do is just look at the list of the examples, it gives an indication of the kinds of models which can be easily solved using the toolkit;

- ValueFnIter: Solves the discrete state space value function iteration problem giving the Value Function and optimal policies.
- SteadyState: Gives the steady-state object associated with this case. It may be a distribution function for the steady-state (in a model with no aggregate uncertainty), a probability distribution for the asymptotic steady-state (in a model with aggregate uncertainty), or one of a number of other things. See each Case for details.
- SimulateTimeSeries: IMPLEMENTED BUT NOT YET DOCUMENTED
- FiniteHorzValueFnIter: Solves the discrete state space finite-horizon value function iteration problem giving the Value Function and optimal policies. IMPLEMENTED BUT YET DOCUMENTED. GPU NOT YET SUPPORTED.
- A variety of useful miscellaneous functions are also provided,

[2]Other popular methods, such as the perturbation methods used by Dynare require the first-order conditions to be both necessary and sufficient

  - TauschenMethod: Gives grid and transition function, inputs are parameters of an AR(1) process.
  - StandardBusCycStats: Generates standard business cycle statistics from time series. IMPLEMENTED BUT NOT YET DOCUMENTED. GPU NOT YET SUPPORTED.

1.4 Parallelization

A number of the commands have an input called 'Parallel'. If Parallel is set to zero the code will run without any parallelization. If Parallel is set to 1 then codes will run parallelized across CPUs — both the value function iteration and some simulation commands allow this, and where it is available provide substantial speed improvements for medium-to-large size models. If Parallel is set to 2 the code will run in parallelized on the GPU, where available this options provides the best performance.

To be able to use the option to parallelize on the GPU you need to install the CUDA toolkit (GPU parallelization only works with NVIDIA graphics cards that support CUDA). My website has a document explaining how I installed it on my laptop which runs Kubuntu (linux). If you use Windows or Mac you will have to google for instructions on how to set up Matlab to use the GPU.

1.5 Details on the Algorithms

Roughly speaking it performs value function iteration on a discrete state space[3]. To handle multiple dimensions it vectorizes them.

I am in the progress of producing some detailed documentation on how the toolkit solves value function problems, meantime you can always just look at the codes or feel free to email me.

1.6 Theory Behind These Methods

For a discussion of the theory behind Value function iteration see SLP. For a discussion of solving Value function iteration by discrete state space approximation see Burnside (2001), in particular part 5.4.2; this also contains a discussion of how to choose grids.

2 Getting Started

To use this toolkit just download it and put the folder 'VFIToolkit' on your computer. To be able to call the commands/functions that make up the toolkit you have to add the 'VFIToolkit' folder to the 'path's known to Matlab so that it can find them. There are two ways to do this: (i) in Matlab use the 'Current Folder' to navigate to where you have the 'VFIToolkit' folder, right-click on the 'VFIToolkit' folder and select 'Add to path>Selected Folder and Subfolders'; (ii) add the line $addpath(genpath('path/Toolkit'))$ at the top of your codes, replacing *path* with whereever you saved the 'VFIToolkit' folder (if it is in the same folder as your code you can replace *path* with a dot, ie. $./VFIToolkit$).

[3]The code used by Aldrich, Fernandez-Villaverde, Gallant, and Rubio-Ramirez (2011) and described at parallelecon.com/vfi/ is similar, but not exactly the same, as the VFI algorithm used by the Toolkit.

Now that Matlab knows where to find the toolkit you are ready to go.

3 Infinite Horizon Value Function Iteration: Case 1

The relevant command is
[V,Policy] = ValueFnIter_Case1(V0, n_d, n_a, n_z, d_grid, a_grid, z_grid, pi_z,
beta, ReturnFn, [vfoptions], [ReturnFnParams]);
This section describes the problem it solves, all the inputs and outputs, and provides some further info on using this command.

The Case 1[4] infinite-horizon value function iteration code can be used to solve any problem that can be written in the form

$$V(a,z) = \max_{d,a'}\{F(d,a',a,z) + \beta E[V(a',z')|a,z]\}$$

subject to

$$z' = \pi(z)$$

where
z ≡ vector of exogenous state variables
a ≡ vector of endogenous state variables
d ≡ vector of decision variables
notice that any constraints on d, a, & a' can easily be incorporated into this framework by building them into the return function. Note that this case is only applicable to models in which it is possible to choose a' directly; when this is not so Case 2 will be required.

The main inputs the value function iteration command requires are the grids for d, a, and z; the discount rate; the transition matrix for z; and the return function F.

It also requires to info on how many variables make up d, a and z (and the grids onto which they should be discretized). And it accepts an initial guess for the value function $V0$ (if you have a good guess this can make things faster).

$vfoptions$ allows you to set some internal options (including parallization), if $vfoptions$ is not used all options will revert to their default values.

The forms that each of these inputs and outputs takes are now described in detail. The best way to understand how to use the command may however be to just go straight to the examples; in particular those for the Stochastic NeoClassical Growth model (Appendix F.1) and the Basic Real Business Cycle model (Appendix F.2).

3.1 Inputs and Outputs

To use the toolkit to solve problems of this sort the following steps must first be made.

[4]The description of this as case 1 is chosen as it coincides exactly with the definition of case 1 for stochastic value function problems used in Chapter 8 & 9 of Stokey, Lucas & Prescott - Recursive Dynamic Economics (eg. pg. 260). In their notation this is any problem that can be written as $v(x,z) = \sup_{y\in\Gamma(x,z)}\{F(x,y,z) + \beta\int_Z v(y,z')Q(z,dz')\}$

- Set the discount rate
  beta = 0.96;

- Define *n_a*, *n_z*, and *n_d* as follows. *n_a* should be a row vector containing the number of grid points for each of the state variables in a; so if there are two endogenous state variables the first of which can take two values, and the second of which can take ten values then $n_a = [2, 10]$;. *n_d* & *n_z* should be defined analagously.

- Create the (discrete state space) grids for each of the d, a & z variables,
  a_grid=linspace(0,2,100)'; d_grid=linspace(0,1,100)'; z_grid=[1;2;3];
  (They should be column vectors. If there are multiple variables they should be stacked column vectors)

- Create the transition matrices for the exogenous z variables[5]
  pi_z=[0.3,0.2,0.1;0.3,0.2,0.2; 0.4,0.6,0.7];
  (Often you will want to use the Tauchen Method to create *z_grid* and *pi_z*)

- Define the return function. This is the most complicated part of the setup. See the example codes applying the toolkit to some well known problems later in this section for some illustrations of how to do this. It should be a Matlab function that takes as inputs various values for $(d, aprime, a, z)$ and outputs the corresponding value for the return function.
  ReturnFn=@(d,aprime,a,z) ReturnFunction_AMatlabFunction

- Define the initial value function, the following one will always work as a default, but by making smart choices for this inital value function you can cut the run time for the value function iteration.
  V0=ones(n_a,n_z);

- If you wish to use parallelization on the GPU you must also create *ReturnFnParams* as part of defining the return function. See codes for examples.

That covers all of the objects that must be created, the only thing left to do is simply call the value function iteration code and let it do it's thing.

```
[V,Policy] = ValueFnIter_Case1(Tolerance, V0, n_d, n_a, n_z, d_grid, a_grid, z_grid, pi_z,
        beta, ReturnFn, [vfoptions], [ReturnFnParams]);
```

The outputs are

- V: The value function evaluated on the grid (ie. on $a \times z$). It will be a matrix of size $[n_a, n_z]$ and at each point it will be the value of the value function evaluated at the corresponding point (a, z).

- *Policy*: This will be a matrix of size $[prod(n_d) * prod(n_a), n_a, n_z]$. For each point $(a.z)$ the corresponding entries in *Policy*, namely $Policy(:, a, z)$ will be a vector containing the optimal policy choices for (d, a).[6]

[5]These must be so that the element in row i and column j gives the probability of going from state i this period to state j next period.

[6]By default, $vfoptions.polindorval = 1$, they will be the indexes, if you set $vfoptions.polindorval = 2$ they will be the values.

3.2 Some further remarks

- Models where d is unnecessary (only a' need be chosen): set $n\_d = 0$ and $d_grid = 0$ and don't put it into the return fn, the code will take care of the rest (see eg. Example F.1).
- Often one may wish to define the grid for z and it's transition matrix by the Tauschen method or something similar. The toolkit provides codes to implement the Tauchen method, see Appendix C
- There is no problem with making the transitions of certain exogenous state variables dependent of the values taken by other exogenous state variables. This can be done in the obvious way; see Appendix A. (For example: if there are two exogeneous variables z^a & z^b one can have $Pr(z^b_{t+1} = z^b_j) = Pr(z^b_{t+1} = z^b_j | z^b_t)$ and $Pr(z^a_{t+1} = z^a_j) = Pr(z^a_{t+1} = z^a_j | z^a_t, z^b_{t+1}, z^b_t)$.)
- Likewise, dependence of choices and expectations on more than just this period (ie. also last period and the one before, etc.) can also be done in the usual way for Markov chains (see Appendix A).
- Models with no uncertainty: these are easy to do simply by setting $n\_z = 1$ and $pi\_z = 1$.

3.3 Options

Optionally you can also input a further argument, a structure called *vfoptions*, which allows you to set various internal options. Perhaps the most important of these is *vfoptions.parallel* which can be used get the codes to run parallely across multiple CPUs (see the examples). Following is a list of the *vfoptions*, the values to which they are being set in this list are their default values.

- Define the tolerance level to which you wish the value function convergence to reach
  vfoptions.tolerance=10^(-9)
- Decide whether you want the optimal policy function to be in the form of the grid indexes that correspond to the optimal policy, or to their grid values.
  vfoptions.polindorval=1
  (Set vfoptions.polindorval=1 to get indexes, vfoptions.polindorval=2 to get values.)
- Decide whether or not to use Howards improvement algorithm (recommend yes)
  vfoptions.howards=80
  (Set vfoptions.howards=0 to not use it. Otherwise variable is number of time to use Howards improvement algorithms, about 80 to 100 seems to give best speed improvements.)
- If you want to parallelize the code on the GPU set to two, parallelize on CPU set to one, single core on CPU set as zero
  vfoptions.parallel=0
- If you want feedback set to one, else set to zero
  vfoptions.verbose=0
  (Feedback includes some info on how convergence is going and on the run times of various parts of the code)

- When running codes on CPU it is often faster to input the Return Function as a matrix, rather than as a function. vfoptions.returnmatrix=0
  (By default it assumes you have input a function. Setting $vfoptions.returnmatrix = 1$ tells the codes you have inputed it as a matrix. Setting $vfoptions.returnmatrix = 2$ tells codes that you have input a function, but that it should use the GPU and so will also need the $ReturnFnParams$ input when evaluating the function.)

3.4 Some Examples

Example: Stochastic Neoclassical Growth Model

Example: Basic Real Business Cycle Model

Example: Hansen (1985) - Indivisible Labor and Business Cycles

4 Infinite Horizon Value Function Iteration: Case 2

This command has not yet been updated to work on the GPU.

The Case 2[7] code can be used to solve any problem that can be written in the form[8]

$$V(a,z) = \max_{d}\{F(d,a,z) + \beta E[V(a',z')|a,z]\}$$

subject to

$$z' = \pi(z)$$
$$a' = \phi(d,a,z,z')$$

where
z ≡ vector of exogenous state variables
a ≡ vector of endogenous state variables
d ≡ vector of decision variables
notice that any constraints on d, a, & a' can easily be incorporated into this framework by building them into the return function. While $a' = \phi(d,a,z,z')$ is the most general case it is often not very useful. Thus the code also specifically allows for $\phi(d,z,z')$ and $\phi(d)$.

4.1 Preparing the model

To use the toolkit to solve problems of this sort the following steps must first be made.

1. Define the tolerance level to which you wish the value function code to reach
   Tolerance=10ˆ(-9)

[7]The description of this as case 2 is chosen as it coincides exactly with the definition of case 2 for stochastic value function problems used in Chapter 8 & 9 of Stokey, Lucas & Prescott - Recursive Dynamic Economics (eg. pg. 260). In their notation this is any problem that can be written as $v(x,z) = \sup_{y\in\Gamma(x,z)}\{F(x,y,z) + \beta\int_Z v(\phi(x,y,z'),z')Q(z,dz')\}$

[8]The proofs of SLP do not allow ϕ to be a function of z.

2. Set the discount rate
   beta = 0.96;

3. Define n_a, n_z, and n_d as follows. n_a should be a row vector containing the number of grid points for each of the state variables in a; so if there are two endogenous state variables the first of which can take two values, and the second of which can take ten values then $n_a = [2, 10]$;. n_d & n_z should be defined analagously.

4. Create the (discrete state space) grids for each of the d, a & z variables,
   a_grid=linspace(0,2,100)'; d_grid=linspace(0,1,100)'; z_grid=[1;2;3];
   (They should be column vectors. If there are multiple variables they should be stacked column vectors)

5. Create the transition matrices for the exogenous z variables[9]
   pi_z=[0.3,0.2,0.1;0.3,0.2,0.2; 0.4,0.6,0.7];
   (Often you will want to use the Tauchen Method to create z_grid and pi_z)

6. Define the return function matrix. This is the most complicated part of the setup. See the example codes applying the toolkit to some well known problems later in this section for some illustrations of how to do this. It should be a Matlab function that takes as inputs various values for (d, a, z) and outputs the corresponding value for the return function.
   ReturnFn=@(d,a,z) ReturnFunction_AMatlabFunction

7. Define, as a matrix, the function ϕ which determines next periods state. The following one is clearly trivial and silly, see the example codes applying the toolkit to some well known problems later in this section for some illustrations of how to do this.
   Phi_aprime=ones(n_d,n_z,n_z);
   In practice the codes always use $Phi\_aprimeKron$ as input. See Appendix AAA (unwritten) on how $Kron$ variables relate to the standard ones.
   Define $Case2\_Type$. This is what tells the code whether you are using
   $Case2\_Type = 1$: $\phi(d, a, z, z')$
   $Case2\_Type = 2$: $\phi(d, z, z')$
   $Case2\_Type = 3$: $\phi(d)$
   $Case2\_Type = 4$: $\phi(d, a)$ (this has not yet been implemented in all codes)
   You should use the one which makes $\phi(\cdot)$ the smallest dimension possible for your problem as this will be fastest.

8. Define the initial value function, the following one will always work as a default, but by making smart choices for this inital value function you can cut the run time for the value function iteration.
   V0=ones(n_a,n_z);

That covers all of the objects that must be created, the only thing left to do is simply call the value function iteration code and let it do it's thing.

```
[V,PolicyIndexes] =
    ValueFnIter_Case2(Tolerance, V0, n_d, n_a, n_z, pi_z,
        Phi_aprimeKron, beta, ReturnFn);
```

[9]These must be so that the element in row i and column j gives the probability of going from state i this period to state j next period.

4.2 Some further remarks

- Notice that Case 1 models are in fact simply a subset of Case 2 models in which one of the decision variables (d) is simply next periods endogenous state (a'). This means that the Case 2 code could in principle be used for solving Case 1 models, however this would just make everything run slower. (Using the Case 2 code for case one models is done by setting all the elements of Phi_aprime to zero, except those for which (a certain element of) d equals $aprime$ which should be set equal to one.)
- The remaining remarks are simply repeats of remarks from Case 1:
  - Often one may wish to define the grid for z and it's transition matrix by the Tauschen method or something similar. The toolkit provides codes to implement the Tauchen method, see Appendix C.
  - There is no problem with making the transitions of certain exogenous state variables dependent of the values taken by other exogenous state variables. This can be done in the obvious way.
  - Likewise, dependence of choices and expectations on more than just this period (ie. also last period and the one before, etc.) can also be done in the usual way for Markov chains.
  - Models with no uncertainty: these are easy to do simply by setting $n\_z = 1$ and $pi\_z = 1$.

4.3 Options

Optionally you can also input a further argument, a structure called *vfoptions*, which allows you to set various internal options. Perhaps the most important of these is *vfoptions.parallel* which can be used get the codes to run parallely across multiple CPUs (see the examples). The full list of the *vfoptions* is just the same as for the Case 1 code — see there for details.

5 Calculating Steady State Distributions

Once you solve the Value Function Iteration problem and get the optimal policy function, $Policy$, there are two main functions in the Toolkit that can be used to calculate the steady state distribution. The first $SteadyState\_Case1\_Simulation$ (and $Case2$) is based on simulating an individual for multiple time periods and then aggregating this across time (like a simulated estimation of the empirical cdf). The second $SteadyState\_Case1$ (and $Case2$) is based on iterating directly on the agents distribution until it converges. In practice a good combination of speed and accuracy often comes from using $SteadyState\_Case1\_Simulation$ to generate a starting guess, which can then be used as an input to $SteadyState\_Case1$ to get an accurate answer. This is the method followed in many of the examples. The main weakness of this approach is that $SteadyState\_Case1$ is very memory intensive and so is sometimes not usable with larger grids.

5.1 SteadyState_Case1_Simulation

All of the inputs required will already have been created either by running the VFI command, or because they were themselves required as an input in the VFI command.

5.1.1 Inputs and Outputs

To use the toolkit to solve problems of this sort the following steps must first be completed.

- You will need the optimal policy function, *Policy*, as outputed by the VFI commands.
  Policy
- Define *n_a*, *n_z*, and *n_d*. You will already have done this to be able to run the VFI command.
- Create the transition matrices, *pi_z* for the exogenous z variables. Again you will already have done this to be able to run the VFI command.

That covers all of the objects that must be created, the only thing left to do is simply call the value function iteration code and let it do it's thing.
SteadyStateDist=SteadyState_Case1_Simulation(Policy,n_d,n_a,n_z,pi_z, [simoptions]);

The outputs are

- *SteadyStateDist*: The steady state distribution evaluated on the grid (ie. on $a \times z$). It will be a matrix of size $[n_a, n_z]$ and at each point it will be the value of the probability distribution function evaluated at the corresponding point (a, z).

5.1.2 Options

Optionally you can also input a further argument, a structure called *simoptions*, which allows you to set various internal options. Perhaps the most important of these is *simoptions.parallel* which can be used get the codes to run on the GPU (see the examples). Following is a list of the *simoptions*, the values to which they are being set in this list are their default values.

- Define the starting (seed) point for each simulation.
  simoptions.seedpoint=[ceil(N_a/2),ceil(N_z/2)];
- Decide how many periods the simulation should run for.
  simoptions.simperiods=10ˆ4;
- Decide for how many periods the simulation should perform a burnin from the seed point before the 'simperiods' begins.
  simoptions.burnin=10ˆ3;
- If you want to parallelize the code on the GPU set to two, parallelize on CPU set to one, single core on CPU set as zero
  simoptions.parallel=0;
  (Each simulation will be *simperiods/ncores* long, and each will begin from *seedpoint* and have a burnin of *burnin* periods.)

- If you want feedback set to one, else set to zero
  simoptions.verbose=0
  (Feedback includes some on the run times of various parts of the code)
- If you are using parallel you need to tell it how many cores you have (this is true both for CPU and GPU parallelization). simoptions.ncores=1;

5.2 SteadyState_Case1

All of the inputs required will already have been created either by running the VFI command, or because they were themselves required as an input in the VFI command. The only exception is an initial guess for *SteadyStateDist*; which can be any matrix of size $[n\_a, n\_z]$ which sums to one.

5.2.1 Inputs and Outputs

To use the toolkit to solve problems of this sort the following steps must first be completed.

- You need an initial guess, *SteadyStateDist*, for the steady-state distribution. This can be any $[n\_a, n\_z]$ matrix which sums to one.
  SteadyStateDist=ones([n_a,n_z]);
- You will need the optimal policy function, *Policy*, as outputed by the VFI commands.
  Policy
- Define n_a, n_z, and n_d. You will already have done this to be able to run the VFI command.
- Create the transition matrices, pi_z for the exogenous z variables. Again you will already have done this to be able to run the VFI command.

That covers all of the objects that must be created, the only thing left to do is simply call the value function iteration code and let it do it's thing.
SteadyStateDist=SteadyState_Case1(SteadyStateDist,PolicyIndexes,n_d,n_a,n_z,pi_z,[simoptions])

The outputs are

- *SteadyStateDist*: The steady state distribution evaluated on the grid (ie. on $a \times z$). It will be a matrix of size $[n_a, n_z]$ and at each point it will be the value of the probability distribution function evaluated at the corresponding point (a, z).

5.2.2 Some Remarks

- Be careful on the interpretation of the limiting distribution of the measure over the endogenous state and the idiosyncratic endogenous state. If the model has idiosyncratic but no aggregate uncertainty then this measure will also represent the steady-state agent distribution. However if there is aggregate uncertainty then it does not represent the measure of agents at any particular point in time, but is simply the unconditional probability distribution [Either of aggregates, or jointly of aggregates and the agents. Depends on the model. For more on this see Imrohoglu (1989) , pg 1374)].

- If the model has idiosyncratic and aggregate uncertainty and individual state is independent of the aggregate state then $pi\_sz = kron(pi\_s, pi\_z)$;.

5.2.3 Options

Optionally you can also input a further argument, a structure called *simoptions*, which allows you to set various internal options. Perhaps the most important of these is *simoptions.parallel* which can be used get the codes to run on the GPU (see the examples). Following is a list of the *simoptions*, the values to which they are being set in this list are their default values.

- Define the tolerance level to which you wish the steady-state distribution convergence to reach
  simoptions.tolerance=10ˆ(-9)
- Decide whether you want the optimal policy function to be in the form of the grid indexes that correspond to the optimal policy, or to their grid values.
  simoptions.maxit=5*10ˆ4;
  (In my experience, after simulating an initial guess, if you need more that 5*10ˆ4 iterations to reach the steady-state it is because something has gone wrong.)
- If you want to parallelize the code on the GPU set to two, parallelize on CPU set to one, single core on CPU set as zero
  simoptions.parallel=0;

6 Simulating Time Series

References

Eric Aldrich, Jesus Fernandez-Villaverde, Ronald Gallant, and Juan Rubio-Ramirez. Tapping the supercomputer under your desk: Solving dynamic equilibrium models with graphics processors. *Journal of Economic Dynamics and Control*, 35(3):386–393, 2011.

S. Aruoba, Jesus Fernandez-Villaverde, and Juan Rubio-Ramirez. Comparing solution methods for dynamic equilibrium economies. *Journal of Economic Dynamics and Control*, 30(12):2477–2508, 2006.

William Brock and Leonard Mirman. Optimal economic growth and uncertainty: The discounted case. *Journal of Economic Theory*, 4(3):479–513, 1972.

C. Burnside. Discrete state-space methods for the study of dynamic economies. In R. Márimon and A. Scott, editors, *Computational Methods for the Study of Dynamic Economies*, chapter 5. Oxford University Press, 2 edition, 2001.

J. Díaz-Gímenez. Linear quadratic approximations: An introduction. In R. Márimon and A. Scott, editors, *Computational Methods for the Study of Dynamic Economies*, chapter 2. Oxford University Press, 2001.

Javier Díaz-Giménez, Edward C. Prescott, Fernando Alvarez, and Terry Fitzgerald. Banking in computable general equilibrium economies. *Journal of Economic Dynamics and Control*, 16: 533–559, 1992.

G. Hansen. Indivisible labor and the business cycle. *Journal of Monetary Economics*, 16(3):309–327, 1985.

Ayse Imrohoglu. Cost of business cycles with indivisibilities and liquidity constraints. *Journal of Political Economy*, 97(6):1368–1383, 1989.

Robert E. Lucas. *Models of Business Cycles.* Wiley-Blackwell, first edition, 1987.

A Markov Chains: Theory

This section explains a couple of useful things about Markov chains. When not otherwise specified Markov chains are assumed to be of first-order.

Let us start with a definition: q_t is a (discrete-time) n-state first-order Markov chain if at any time t, q_t takes a value in $\{q_1, ..., q_n\}$, and the probability that q takes the value q_j at time $t+1$ depends only on the state of q at time t, that is $Pr(q_{t+1} = q_j) = Pr(q_{t+1} = q_i | q_t = q_j)$.

The transition matrix of q is defined as the n-by-n matrix $\Pi^q = [\pi^q_{ij}]$, where $\pi^q_{ij} = Pr(q_{t+1} = q_i | q_t = p_j)$. Thus the element of the transition matrix in row i and column j gives the probability that the state tomorrow is q_j given that the state today is q_i.

For a more comprehensive treatment of Markov Chains see SLP chapters 8, 11, & 12; Sargent & Ljungquist chapter 2; or Grimmett & Stirzaker - Probability and Random Processes.

A.1 How to turn a Markov chain with two variables into a single variable Markov chain

Say you have a Markov chain with two variables, q & p (the most common macroeconomic application for this is in heterogenous models, where q is an idiosyncratic state s and p is the aggregate state z). Let q take the states $q_1, ...q_n$, and p take the states $p_1, ..., p_m$. We start with the simple example of how to combine the two when their transitions are completely independent of each other so as to illustrate the concepts and then treat the general case.

When q and p are independent Markov chains, $Pr(q_{t+1} = q_j) = Pr(q_{t+1} = q_i | q_t = q_j) = \pi^q_{ij}$ $\forall i, j = 1, ...n$ and $Pr(p_{t+1} = p_j) = Pr(p_{t+1} = p_i | p_t = p_j) = \pi^p_{ij}$ $\forall i, j = 1, ...m$, then we can define a new single-variable Markov chain r simply by taking the Kronecker Product of q and p. Thus, r will have n times m states, $[r_1, .., r_{nm}]' = [q_1, ...q_n]' \otimes [p_1, ...p_m]'$, and it's transition matrix will be the Kronecker Product of their transition matrices; $\Pi^r = \Pi^q \otimes \Pi^p$. For the definition of the Kronecker product of two matrices see wikipedia.

For the vector (q, p) to be a (first-order) Markov chain, at least one of q and p must be independent of the current period value of the other. Thus we assume, without loss of generality, that q_t evolves according to the transformation matrix defined by $Pr(q_t) = Pr(q_t | q_{t-1}, p_{t-1})$, while p_t has the transformation matrix $Pr(p_t) = Pr(q_t, q_{t-1}, p_{t-1})$[10]. Again we can define a Markov chain r will have n times m states by $[r_1, .., r_{nm}]' = [q_1, ...q_n]' \otimes [p_1, ...p_m]'$. The transition matrix however is now more complicated; rather than provide a general formula you are referred to the examples of Imrohoroglu (1989) and OTHEREXAMPLE which illustrate some of the common cases you may wish to model.

If you have three or more variables in vector of the first-order Markov chain you can reduce this to a scalar first-order Markov chain simply by iteratively combining pairs. For example with three variables, (q^1, q^2, q^3), start by first defining r^1 as the combination of q^1 & q^2 (combining them as described above), and then r as the combination of r^1 and q^3.

[10]Note that here I switch from describing Markov chains as $t+1|t$ to $t|t-1$, the difference is purely cosmetic.

A.2 How to turn a second-order Markov chain into a first-order Markov chain

Suppose that q, with states $q_1, ...q_n$ is a second-order Markov chain, that is $Pr(q_{t+1} = q_i) = Pr(q_{t+1} = q_i | q_t = q_j, q_{t-1} = q_k) \neq Pr(q_{t+1} = q_i | q_t = q_j)$. Consider now the vector (q_{t+1}, q_t). It turns out that this vector is in fact a vector first-order Markov chain (define this periods state (q_{t+1}, q_t) in terms of last periods state (q_t, q_{t-1}) by defining this periods q_{t+1} as a function of last periods q_t & q_{t-1} following the original second-order Markov chain, and define this periods q_t as being last periods q_t). Now just combine the two elements of this first-order vector Markov chain, (q_{t+1}, q_t), into a scalar first-order Markov chain r in the same way as we did above, that is by the Kronecker product.

For third- and higher-order Markov chains simply turn them into three- and higher-element first-order vector Markov chains, and then combine them into scalars by repeated pairwise Kronecker products as above.

B Markov Chains: Some Toolkit Elements

This section mentions a couple of commands that can be useful with Markov chains. One command related to Markov chains which is not in this section is the Tauchen Method (see Appendix C).

B.1 MarkovChainMoments:

For the markov chain z defined by the states z_grid and the transition matrix pi_z (row i column j gives the transition probability of going from state i this period to state j next period). The command

```
[z_mean,z_variance,z_corr,z_statdist]=MarkovChainMoments(z_grid,pi_z,T, Tolerance)
```

gives the mean, variance, and correlation of the markov chain, as well as it's stationary distribution. $Tolerance$ determines how strict to be in finding the stationary distribution. T is needed as the correlation is determined by simulating the process (for T periods).
Note that should you then wish to calculate the expectation of any function of z, $E[f(z)]$, it will simply be $f(z\_grid')*z\_statdist$. So for example the mean, $E[z]$, is actually just $z\_grid'*z\_statdist$; while $E[exp(z)]$ is $exp(z\_grid')*z\_statdist$.

C Tauchen Method

The toolkit contains a couple of functions for generating the grids and transistion matrices for exogenous variables based on the Tauchen Method for discrete state space approximations of VARs. Currently there is code for creating the approximation directly from the parameters of an AR(1) Functions for higher order AR processes, or for VAR processes are not yet implemented within the toolkit (in theory both can be done using the Tauchen Method).

Namely,

- *TauchenMethod_Param*: Generates a markov chain approximation for AR(1) process defined by the inputed parameters.

C.1 TauchenMethod_Param

To create a discrete state space approximation of the AR(1) process,

$$z_t = \mu + \rho z_{t-1} + \epsilon_t \quad \epsilon_t \sim^{iid} N(0, \sigma^2) \tag{1}$$

the command is,
[z_grid, pi_z]=TauchenMethod_Param(mu,sigmasq,rho,znum,q, [tauchenoptions]); the outputs are the grid of discretized values (*z_grid*) and the transition matrix (*pi_z*). We now describe in more detail the inputs and outputs.

C.1.1 Inputs and Outputs

The inputs are

- Define the constant, autocorrelation, and error variance terms relating to equation 1, eg.
  mu=1; rho=0.9; sigmasq=0.09;
- Set the number of points to use for the approximation,
  znum = 9;
  (When using the value function iteration commands, *znum* will correspond to *n_z* whenever it is the only shock.)
- Set q, roughly q defines the number of standard deviations of z covered by the range of discrete points,
  q = 3;

the optional input is *tauchenoptions*

- Can either not parallelize (tauchenoptions.parallel=0), or parallelize on graphics card (tauchenoptions.parallel=2).
  (If you set tauchenoptions.parallel=1 it will just act as if you set a value of zero, as parallelizing on CPU does not appear to help at all)

The outputs are,

- The discrete grid (of *znum* points). A column vector.
  z_grid
- The transition matrix; the element in row i, column j gives the probability of moving to state j tomorrow, given that today is state i,
  pi_z
  (each row sums to one; size is *znum*-by-*znum*)

D Howards Improvement Algorithm (for Infinite Horizon Value Function Iteration)

Howards improvement algorithm is not used by the infinite horizon value function iteration codes for the first couple of iterations as it can otherwise lead to 'contamination' ofthe -Inf elements. It is set automatically to kick in just after this point (ie. once number of -Inf elements appears stable, this is when *currdist* is finite (¡1), this slowed some applications (in which it didn't matter when it kicked in) but avoids what can otherwise be a fatal error. It is my experience that the greatest speed gains come from choosing $Howards$ to be something in the range of 80 to 100, hence the default value of 80.

E Some Issues to be Aware Of

E.1 A Warning on 'Tolerance' and Grid Sizes

The toolkit provides feedback on the occurance of various errors that may occour. One thing it will not warn you of however (other than obvious things such as incorrect values in the return function matrix) is that if your grids are too rough in comparison to the value you have choosen for Tolerance then it will be impossible for that tolerance level to be reached and the code will never end. This will be evident in the feedback the toolkit provides whilst running in $verbose = 1$ mode as the 'currdist' will begin repeating the same values (or series of values) instead of continuing to converge to zero. In practice I have almost never had this problem, but I mention it just in case.

E.2 Poorly Chosen Grids

Another problem the toolkit does not tell you about is if your model tries to leave the grid (eg. say your grid has a max capital level of 4, but agents with a capital level of 4 this period want to choose a capital level of 4.2 for next period). This can be checked for simply by looking at the output of the policy functions. If people do not choose to move away from the edge, then you may have this error.

In any case it is always worth taking a close look at the value function and policy function matrixes generated by the code to make sure they make sense as a check that you have defined all the inputs correctly (eg. look for things like higher utility in the 'good' states, utility increasing in asset holdings, not too many times that people choose to stay in their current endogenous state (this may be a signal your grid may be too coarse), etc.). An alternative way is to check for mass at the edges of grids in the steady-state distributions.

F Examples

The following is a collection of example models and code based on a number of classic models/-papers. The examples are meant to illustrate how to use the toolkit to code these models, thus they generally ignore all issues of calibration and of interpreting the results; they should be read in conjunction with the papers themselves.

The examples F.1 and F.2 provide two basic examples of how to set up and call the value function iteration.

Example F.3 replicates the paper of Hansen (1985), and in doing so shows how to use the toolkit to easily calculate the standard business cycle statistics (correlations between the different variables).

Examples F.4 and F.5 replicate two early heterogeneous agent models (Imrohoglu (1989) and Díaz-Giménez, Prescott, Alvarez, and Fitzgerald (1992)) (without general equilibrium conditions). They give an example of how to use various commands in the toolkit to calculate statistics relating to the agent distributions.

F.1 Stochastic Neoclassical Growth Model (Diaz-Gimenez, 2001)

The Neoclassical Growth Model was first developed in Brock and Mirman (1972), although our treatment of the model here is based on Díaz-Gímenez (2001). If you are unfamiliar with the model a full description can be found in Section 2.2 of Díaz-Gímenez (2001); he also briefly discusses the relation to the social planners problem, how the problem looks in both sequential and recursive formulation, and how we know that value function iteration will give us the correct solution. In what follows I assume you are familiar with these issues and simply give the value function problem. Our concentration here is on how to solve this problem using the toolkit.

The value function problem to be solved is,

$$V(k,z) = \sup_{k'} \{ \frac{c^{1-\gamma}}{1-\gamma} + \beta * E[V(k',z')|z]$$

subject to

$$c + i = exp(z)k^{\alpha} \tag{2}$$
$$k' = (1-\delta)k + i \tag{3}$$
$$z' = \rho z + \epsilon' \quad , \epsilon \overset{iid}{\sim} N(0,\sigma_{\epsilon}^2) \tag{4}$$

where k is physical capital, i is investment, c is consumption, z is a productivity shock that follows an AR(1) process; $exp(z)k^{\alpha}$ is the production function, δ is the depreciation rate.

We will use the Tauchen Method to discretize the AR(1) shock.

The following code solves this model, using parallelization on GPU, and draws a graph of the value function; it contains many comments describing what is being done at each step. This code is a copy of $StochasticNeoClassicalGrowthModel.m$ which you can run and modify to get a better feel for how to use the toolkit. Below that is the code that defines the return function, $StochasticNeoClassicalGrowthModel\_ReturnFn.m$.

Contents of StochasticNeoClassicalGrowthModel.m

```
% Neoclassical Stochastic Growth Model
% Example based on Diaz-Gimenez (2001) - Linear Quadratic Approximations: An Introduction
% (Chapter 2 in 'Computational Methods for the Study of Dynamic Economies', edited by Marimon & Scott)

% This model is also used by Aldrich, Fernandez-Villaverde, Gallant, & Rubio-Ramirez (2011) - "Tapping the supercomputer under your desk: Solving dynamic equilibrium models with graphics processors,"
% But they do use slightly different parameters to those used here.


tic;

Javier=1;   %If you set this parameter to 0 then the parameters will all be set to those used by Aldrich, Fernandez-Villaverde, Gallant, & Rubio-Ramirez (2011)

%% Set up
tauchenoptions.parallel=2;
vfoptions.returnmatrix=2;
vfoptions.parallel=2;

% The sizes of the grids
n_z=2^2;
n_k=2^12;

%Discounting rate
beta = 0.96;

%Give the arameter values
alpha = 0.33;
gamma=1; %gamma=1 is log-utility
rho = 0.95;
delta = 0.10;
sigmasq_epsilon=0.09;

if Javier==0
    n_z=4;
    beta=0.984;
    gamma=2;
    alpha=0.35;
    delta=0.01;
    rho=0.95;
    sigma_epsilon=0.005;
    sigmasq_epsilon=sigma_epsilon^2;
end
```

```

%% Compute the steady state
K_ss=((alpha*beta)/(1-beta*(1-delta)))^(1/(1-alpha));
X_ss= delta*K_ss;
%These are not really needed; we just use them to determine the grid on
%capital. I mainly calculate them to stay true to original article.

%% Create grids (grids are defined as a column vectors)

q=3; % A parameter needed for the Tauchen Method
[z_grid, pi_z]=TauchenMethod_Param(0,sigmasq_epsilon,rho,n_z,q,
    tauchenoptions); %[states, transmatrix]=TauchenMethod_Param(mew,
    sigmasq,rho,znum,q), transmatix is (z,zprime)

k_grid=linspace(0,20*K_ss,n_k)'; % Grids should always be declared as
    column vectors

%% Now, create the return function
ReturnFn=@(aprime_val, a_val, s_val, gamma, alpha, delta)
    StochasticNeoClassicalGrowthModel_ReturnFn(aprime_val, a_val, s_val,
    gamma, alpha, delta);
ReturnFnParams=[gamma, alpha, delta]; %It is important that these are
    in same order as they appear in '
    StochasticNeoClassicalGrowthModel_ReturnFn'

%% Solve
%Do the value function iteration. Returns both the value function
    itself,
%and the optimal policy function.
d_grid=0; %no d variable
n_d=0;


V0=ones(n_k,n_z,'gpuArray');
[V, Policy]=ValueFnIter_Case1(V0, n_d,n_k,n_z,d_grid,k_grid,z_grid,
    pi_z, beta, ReturnFn, vfoptions, ReturnFnParams);
time=toc;

fprintf('Time to solve the value function iteration was %8.2f seconds.
    \n', time)


%% Draw a graph of the value function
%surf(V)

% Or to get 'nicer' x and y axes use
surf(k_grid*ones(1,n_z),ones(n_k,1)*z_grid',V)
```

Contents of StochasticNeoClassicalGrowthModel_ReturnFn.m

```
function F=StochasticNeoClassicalGrowthModel_ReturnFn(kprime_val,k_val,
    z_val, gamma, alpha, delta)

F=-Inf;
c=exp(z_val)*k_val^alpha-(kprime_val-(1-delta)*k_val);
if c>0
    if gamma==1
        F=log(c);
    else
        F=(c^(1-gamma))/(1-gamma);
    end
end

end
```

F.2 Basic Real Business Cycle Model

The Basic Real Business Cycle Model presented here simply extends the Stochastic Neo-Classical Growth Model to include an endogenous choice of labour supply (of how much to work). This is an important difference in terms of the Toolkit we know have a 'd' variable: a choice variable that does not affect tomorrows state (in the Stochastic Neo-Classical Growth Model the only choice variable was '$aprime$', next periods state). The following code solves the model and then uses the results to reproduce the relevant findings of Aruoba, Fernandez-Villaverde, and Rubio-Ramirez (2006) looking at the sizes of numerical errors — this serves the dual purpose of allowing you to see how measures of the numerical errors change with the choice of grids, and of providing an example of how to use the policy function created by the Value Function (Case 1) command.

The value function problem to be solved is,

$$V(k,z) = \sup_{k'}\{\frac{(c^{\theta}(1-l)^{1-\theta})^{\tau}}{1-\tau} + \beta * E[V(k',z')|z]$$

subject to

$$c + i = exp(z)k^{\alpha}l^{1-\alpha} \tag{5}$$

$$k' = (1-\delta)k + i \tag{6}$$

$$z' = \rho z + \epsilon' \quad , \epsilon \overset{iid}{\sim} N(0,\sigma_{\epsilon}^2) \tag{7}$$

where k is physical capital, i is investment, c is consumption, l is labour supply, z is a productivity shock that follows an AR(1) process; $exp(z)k^{\alpha}$ is the production function, δ is the depreciation rate.

We will use the Tauchen Method to discretize the AR(1) shock.

The following code solves this model, using parallelization on GPU, and then reproduces a number of relevant elements from Tables & Figures in Aruoba, Fernandez-Villaverde, and Rubio-Ramirez (2006) relating to the accuracy of numerical solutions; it contains many comments describing what is being done at each step. This code is a copy of *BasicRealBusinessCycleModel.m* which you can run and modify to get a better feel for how to use the toolkit. Below that is the code that defines the return function, *BasicRealBusinessCycleModel_ReturnFn.m*.

F.3 Hansen (1985) - Indivisible Labor and Business Cycles

Hansen (1985) is an early paper in the RBC literature. At the time it was important for introducing the extensive margin of labour — working or not working — into models (which it did via a trick of using loteries, so that in the aggregate it was back to just being a continuous choice, and because utility was now quasi-linear the implied Frisch elasticity is infinity; highly unrealistic). Thanks to modelling advances we can now model the extensive and intensive margins of labour supply[11] at the same time, and without the need for (highly unrealistic) lotteries, so the model of Hansen (1985) is no longer relevant. It retains historical interest and while now outdated it provides us with a chance to look at how the Toolkit output can be used to generate some of the kinds of model statistics that are standard in the literature.

The value function problem to be solved is,

$$V(k,z)=\sup_{k',l}\{log(c)+Bl+\beta * E[V(k',z')|z]$$

subject to

$$c+i=exp(z)k^{\theta}l^{1-\theta} \tag{8}$$
$$k'=(1-\delta)k+i \tag{9}$$
$$z'=\rho z+\epsilon' \quad ,\epsilon \overset{iid}{\sim} N(0,\sigma_\epsilon^2) \tag{10}$$

where k is physical capital, i is investment, c is consumption, l is aggregate labour supply (for the individual household it represents the probability of working; see Hansen (1985)), z is a productivity shock that follows an AR(1) process; $exp(z)k^{\alpha}l^{1-\theta}$ is the production function, δ is the depreciation rate.

The following code solves this model, using parallelization on GPU, and draws a graph of the value function; it contains many comments describing what is being done at each step. This code is a copy of $Hansen1985.m$ which you can run and modify to get a better feel for how to use the toolkit. Below that is the code that defines the return function, $Hansen1985\_ReturnFn.m$.

Contents of Hansen1985.m

```
%%% Example using the model of Hansen (1985)

% l_z=1; %tech shock
% l_a=1; %assets
% l_d=1; %decide whether to work

n_z=21;
n_a=101;
n_d=21;

%% Some Toolkit options
Parallel=2; % Use GPU

```

[11]The extensive margin is the decision to work or not work, the intensive margin is, having decided to work, how many hours to work.

```
tauchenoptions.parallel=Parallel;

mcmomentsoptions.parallel=tauchenoptions.parallel;

vfoptions.parallel=Parallel;
vfoptions.returnmatrix=Parallel;

simoptions.parallel=Parallel;

%% Setup

%Discounting rate
beta = 0.96;

%Parameter values
alpha = 0.33; % alpha
gamma=-2*log(1-0.53)/0.53;
rho = 0.95; % rho
delta = 0.10; % delta
sigmasq_epsilon=0.09;
%Step 1: Compute the steady state
K_ss=((alpha*beta)/(1-beta*(1-delta)))^(1/(1-alpha));

%Create grids (it is very important that each grid is defined as a
    column
%vector)
q=3;
[z_grid, pi_z]=TauchenMethod_Param(0,sigmasq_epsilon,rho,n_z,q,
    tauchenoptions);
a_grid=linspace(0,2*K_ss,n_a)';
d_grid=linspace(0,1,n_d)';

n_z=length(z_grid);
n_a=length(a_grid);
n_d=length(d_grid);

ReturnFn=@(d_val, aprime_val, a_val, z_val, alpha, delta, gamma)
    Hansen1985_ReturnFn(d_val, aprime_val, a_val, z_val, alpha, delta,
    gamma)
ReturnFnParams=[alpha, delta, gamma];

%% Solve
V0=ones([n_a,n_z],'gpuArray'); %(a,z)
[V, Policy]=ValueFnIter_Case1(V0, n_d,n_a,n_z,d_grid,a_grid,z_grid, pi_z
    , beta, ReturnFn,vfoptions,ReturnFnParams);


SteadyStateDist=SteadyState_Case1_Simulation(Policy,n_d,n_a,n_z,pi_z,
    simoptions);
SteadyStateDist=SteadyState_Case1(SteadyStateDist,Policy,n_d,n_a,n_z,
    pi_z,simoptions);


%% Generate output
% NOT YET DONE
```

Contents of Hansen1985_ReturnFn.m

```
function F=Hansen1985_ReturnFn(d_val, aprime_val, a_val, z_val, alpha,
    delta, gamma)

F=-Inf;
C=exp(z_val)*(a_val^alpha)*(d_val^(1-alpha))-(aprime_val-(1-delta)*
    a_val);
if C>0
    F=log(C)-gamma*d_val;
end

end
```

F.4 Imrohoroglu (1989) - Cost of Business Cycles with Indivisibilities and Liquidity Constraints

Imrohoglu (1989) introduces the use of both idiosyncratic and aggregate shocks; albeit without an aggregate shock. The paper looks at the cost of recessions (business cycles) in a manner following Lucas (1987) introducing into the model the idea that recessions do not affect everyone in the same way, in particular in this model in a recession those how find themselves out of work will be more likely to remain out of work during a recession. A number of different economies (model variants) are considered and the code has an option at the beginning for which one will be implemented. It is of additional interest here as it will illustrate how to use the toolkit commands in the presence of two exogenous shocks.

The value function problem to be solved is,

$$V(a,s,z) = \sup_{a'}\{\frac{c^{1-\sigma}}{1-\sigma} + \beta * E[V(a',s',z')|s,z]$$

subject to

$$c + \frac{a'}{1+r} = a + earnings(s)$$
$$earnings(s) = \begin{cases} y & \text{if s=employed,} \\ \theta y & \text{if s=unemployed} \end{cases} \qquad (s,z') = \pi(s,z)$$

where a is asset holdings, c is consumption, s is an idiosyncratic shock and determines whether a household is employed or unemployed. z is an aggregate shock (recession or expansion). A variety of different economies are considered, with and without aggregate shocks, z, and changing the way in which s changes state (whether or not it depends on aggregate state). See Imrohoglu (1989) for a detailed discussion.

The following code solves this model, using parallelization on GPU, and (IN PROGRESS) replicates the results of Imrohoglu (1989); it contains comments describing what is being done at each step. This code is a copy of *Imrohoroglu1989.m* which you can run and modify to get a better feel for how to use the toolkit. Below that is the code that defines the return function, *Imrohoroglu1989_ReturnFn.m*.

Contents of Imrohoroglu1989.m

```
% Imrohoroglu (1989) - Cost of Business Cycles with Indivisibilities
    and Liquidity Constraints

%Choose one of the following
%EconomyEnvironment pairs
%1: Economy=1, Environment=A
%1: Economy=2, Environment=A
%1: Economy=1, Environment=B
%1: Economy=2, Environment=B
%1: Economy=1, Environment=C
%1: Economy=2, Environment=C
EconomyEnvironment=1

%% Some Toolkit options
```

```
Parallel=2; % Use GPU

tauchenoptions.parallel=Parallel;

mcmomentsoptions.parallel=tauchenoptions.parallel;

vfoptions.parallel=Parallel;
vfoptions.returnmatrix=Parallel;

simoptions.parallel=Parallel;

%% Setup

% num_z_vars=1; %This is 'n' in Imrohoroglu
% num_s_vars=1; %This is 'i' in Imrohoroglu
% num_a_vars=1; %This is 'a' in Imrohoroglu
% num_d_vars=0; %There is none in Imrohorglu (you only choose aprime,
    which Case 1 codes assume is chosen anyway)

%Discounting rate
beta = 0.995;

%Parameters
sigma=6.2; %Lucas (1987) uses 6.2; also compares for value of 1.5
theta=0.25;
r_b=8; %Interest rate on borrowing
r_l=0; %Interest rate on lending (in environment A, this is the only r)
y=1; %Normalization (THIS IS NOT FROM PAPER, I JUST GUESSED)

E1_z_grid=[1,2]';
E2_z_grid=[1]';
s_grid=[y,theta*y]';
E1_pi_sz=[0.9141,0.0234,0.0587, 0.0038; 0.5625, 0.3750, 0.0269, 0.0356;
     0.0608, 0.0016, 0.8813, 0.0563; 0.0375, 0.0250, 0.4031, 0.5344];
E2_pi_s=[0.9565,0.0435;0.5,0.5];
E2_pi_z=[1]';
EA_a_grid=linspace(0,8,301)';
EB_a_grid=linspace(-8,8,601)';


if EconomyEnvironment==1 || EconomyEnvironment==3 || EconomyEnvironment
    ==5
    z_grid=E1_z_grid;
    pi_sz=E1_pi_sz;
elseif EconomyEnvironment==2 || EconomyEnvironment==4 ||
    EconomyEnvironment==6
    z_grid=E2_z_grid;
    pi_sz=kron(E2_pi_s,E2_pi_z);
```

```
end

if EconomyEnvironment==1 || EconomyEnvironment==4
    a_grid=EA_a_grid;
elseif EconomyEnvironment==2 || EconomyEnvironment==5
    a_grid=EB_a_grid;
elseif EconomyEnvironment==3 || EconomyEnvironment==6
    disp('EconomyEnvironment 3 & 6 not yet implemented')
end

%Note that from the point of view of the value function, there is no
%difference between s & z, so we combine them.
sz_grid=[s_grid;z_grid];

n_s=length(s_grid);
n_z=length(z_grid);
n_sz=[n_s,n_z];
n_a=length(a_grid);

n_d=0; % No d variable
d_grid=0;

ReturnFn=@(aprime_val, a_val, s_val,sigma,theta,y,r_l,r_b)
    Imrohoroglu1989_ReturnFn(aprime_val, a_val, s_val,sigma,theta,y,r_l,
    r_b);
ReturnFnParams=[sigma,theta,y,r_l,r_b];

%% Solve the model
%Do the value function iteration. Returns both the value function
    itself,
%and the optimal policy function.
V0=ones(n_a,n_s,n_z,'gpuArray');
[V,Policy]=ValueFnIter_Case1(V0, n_d, n_a, n_sz, d_grid, a_grid,
    sz_grid, pi_sz, beta, ReturnFn, vfoptions, ReturnFnParams);

SteadyStateDist=ones([n_a,n_s,n_z],'gpuArray')./prod([n_a,n_sz]);
SteadyStateDist=SteadyState_Case1(SteadyStateDist,Policy,n_d,n_a,n_sz,
    pi_sz, simoptions);

%% Generate some output following what is reported in Imrohoroglu
    (1989)
% NOT YET DONE
```

Contents of Imrohoroglu1989_ReturnFn.m

```
function F=Imrohoroglu1989_ReturnFn(aprime_val, a_val, s_val,sigma,r_l,
    r_b) %theta,y

F=-Inf;
```

```

if a_val>=0 %If lending
    r=r_l;
else %If borrowing
    r=r_b;
end

C=a_val-aprime_val/(1+r)+s_val;
if C>0
    F=(C^(1-sigma))/(1-sigma);
end

end
```

F.5 Díaz-Gímenez, Prescott, Alvarez, and Fitzgerald (1992) - Banking in Computable General Equilibrium Economies