VFI Toolkit for Matlab

Robert Kirkby
Victoria University of Wellington

October 7, 2020

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. The core of the toolkit is value function iteration on a discrete state space utilizing the GPU. The broader focus is on heterogenous agent models (infinite horizon and finite horizon) and the toolkit automatically handles calculating general equilibrium and transition paths.

The toolkit is available for download from vfitoolkit.com.

If you want to get straight into using the Toolkit, look briefly at Sections 2 and 3, and then go to Examples 14.1 and 14.2 on how to solve the Stochastic Neo-Classical Growth Model and a Basic Real Business Cycle Model respectively. If you are already familiar with the Stochastic Neoclassical Growth Model and Basic Real Business Cycle Model you can skip to looking at the codes implementing them. The toolkit also handles finite horizon value function problems, see Example 14.3 and OLG models.

More advanced examples cover life-cycle models, and general equilibrium and transition paths in heterogenous agent models.
Examples: https://github.com/vfitoolkit/vfitoolkit-matlab-examples
Replications: https://github.com/vfitoolkit/vfitoolkit-matlab-replication
Note: This document undergoes periodic revision.

Contents

1 Introduction **6**

1.1 Why use value function iteration? 7

1.2 Examples and Classic Papers 7

1.3 What Can The Toolkit Do? 7

1.4 Parallelization 8

1.5 Details on the Algorithms 9

1.6 Theory Behind These Methods 9

2 Getting Started **9**

3 Infinite Horizon Value Function Iteration: Case 1 **9**

3.1 Inputs and Outputs 10

3.2 Some further remarks 11

3.3 Options 12

3.4 Some Examples 13

4 Infinite Horizon Value Function Iteration: Case 2 **13**

4.1 Preparing the model 13

4.2 Some further remarks 15

4.3 Options 15

5 Calculating Stationary Distributions **15**

5.1 Inputs and Outputs 16

5.2 Some further remarks 16

5.3 Options 16

6 General Equilibrium of Heterogeneous Agent Models **17**

6.1 Preparing the model 18

6.2 Some further remarks 19

6.3 Options 19

6.4 Examples 20

7 Transition Path for Heterogeneous Agent Models 20

7.1 Preparing the model 20

7.2 Some further remarks 21

7.3 Options 22

7.4 Examples 22

8 Finite Horizon Value Function Iteration: Case 1 23

8.1 Inputs and Outputs 24

8.2 Some further remarks 25

8.3 Options 25

8.4 Some Examples 27

9 Finite Horizon Tools: Simulate Panel Data and Life Cycle Profiles 28

9.1 Inputs and Outputs 28

9.2 Some further remarks 29

9.3 Options 30

9.4 Some Examples 30

10 General Equilibrium of Overlapping-Generations models 30

10.1 Preparing the model 31

10.2 Some further remarks 33

10.3 Options 33

10.4 Examples 33

11 Permanent/Fixed Agent Types 34

12 Transition Path for OLG models 35

12.1 Preparing the model 35

12.2 Some further remarks 36

12.3 Options 36

12.4 Examples 37

13 OLG models with 'Age-Dependent-Grids' 38

13.1 Inputs and Outputs . . . 39
13.2 Some further remarks . . . 40
13.3 Options . . . 40

14 Examples **41**
14.1 Stochastic Neoclassical Growth Model (Diaz-Gimenez, 2001) . . . 41
14.2 Basic Real Business Cycle Model . . . 42
14.3 Finite Horizon Stochastic Consumption Savings Model . . . 43

A Markov Chains: Theory **45**
A.1 How to turn a Markov chain with two variables into a single variable Markov chain . 45
A.2 How to turn a second-order Markov chain into a first-order Markov chain . . . 46

B Markov Chains: Some Toolkit Elements **47**
B.1 MarkovChainMoments: . . . 47

C Tauchen Method **48**
C.1 TauchenMethod . . . 48
C.1.1 Inputs and Outputs . . . 48

D Howards Improvement Algorithm (for Infinite Horizon Value Function Iteration) **50**

E StationaryDist Sub-commands **51**
E.1 StationaryDist_Case1_Simulation . . . 51
E.1.1 Inputs and Outputs . . . 51
E.1.2 Options . . . 51
E.2 StationaryDist_Case1_Iteration . . . 52
E.2.1 Inputs and Outputs . . . 52
E.2.2 Some Remarks . . . 53
E.2.3 Options . . . 53

F Some Issues to be Aware Of **54**
F.1 A Warning on 'Tolerance' and Grid Sizes . . . 54

F.2 Poorly Chosen Grids 54

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'. I'm not using the term lightly; I'm 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, but 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 takes advantage of parallelization on the GPU and CPU (without requiring the user to know when and which to do). This Toolkit, and value function iteration generally, is most useful for problems where the first-order conditions of the problem are not both necessary and sufficient, and so methods like perturbation are not usable; ie. it is 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 parallelization on the GPU. The GPU makes a large difference and so the Toolkit (and documentation) assumes by default that you have and will be using the GPU.

The toolkit also handles finite horizon value function iteration problems, and includes commands to simulate panel data and solve OLG models and their transition paths. The dependence of parameters on age (period) and agent-fixed-type is handled automatically.

As well as the main value function iteration commands (which can also solve Epstein-Zin preferences) the other main commands are to simulate time series and to calculate the stationary distribution (mainly of use in heterogeneous agent models).

There are also some 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.

The VFI Toolkit is also be available directly from github:
https://github.com/vfitoolkit

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 benefits.[1]

Their other main strength is their (potential) 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. (Bertsekas, 1976; Whitt, 1978; Kirkby, 2017b). Importantly they will still converge in the presence of occasionally binding constraints, non-concave choice sets, and non-concave return functions.

A further possible disadvantage is that they are not always easy to implement with state space variables 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 ;). Two main examples, the Stochastic Neoclassical Growth Model and the Basic Real Business Cycle are describe and shown in Section 14, and the codes are available at github.com/vfitoolkit/vfitoolkit-matlab-examples.

Also available are codes replicating some classic Macro papers (you will need to read the original papers alongside the codes), these can be found at github.com/vfitoolkit/vfitoolkit-matlab-replication. The replications 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 and replications. They give 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

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

Function and optimal policies.

- StationaryDist: Gives the stationary distribution associated with the model. The interpretation of this stationary distribution will depend on the model. 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.
- HeteroAgentStationaryEqm: Used to find the equilibrium price levels for a Bewley-Huggett-Aiyagari heterogeneous-agent model with no aggregate uncertainty.
- TransitionPath: Use to find general eqm transition path for a Bewley-Huggett-Aiyagari heterogeneous-agent model with no aggregate uncertainty.
- 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.
- SimPanelValues_FHorz: Simulates panel data for finite-horizon based on optimal policy.
- SimLifeCycleProfiles_FHorz: Simulates life-cycle profiles for finite-horizon based on optimal policy.
- HeteroAgentStationaryEqm_FHorz: Solve for general equilibrium for heterogeneous agent OLG model.
- TransitionPath_FHorz: Use to find general eqm transition path for a heterogeneous-agent OLG model with no aggregate uncertainty.
- A variety of useful miscellaneous functions are also provided.
  - TauchenMethod: Gives grid and transition function, inputs are parameters of an AR(1) process. (Rouwenquist, and a VAR(1) version of TauchenMethod also exist)
  - StandardBusCycStats: Generates standard business cycle statistics from time series. IMPLEMENTED BUT NOT YET DOCUMENTED. GPU NOT YET SUPPORTED.

1.4 Parallelization

By default the toolkit assumes you have a GPU and four or more CPUs. It then automatically switches back and forth between these based on which typically works faster in practice.

If you wish to overrule the defaults you can use the 'parallel' option, available in many of the commands. If Parallel is set to zero the code will run without any parallelization. If Parallel is set to 1 then codes will run on parallel CPUs (the *ncores* option controls how many CPU cores are used). If Parallel is set to 2 the code will run in parallelized on the GPU.

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).

1.5 Details on the Algorithms

Roughly speaking it performs value function iteration on a discrete state space[2]. To handle multiple dimensions it vectorizes them.

1.6 Theory Behind These Methods

For a discussion of the theory behind Value function iteration see Stokey, Lucas, and Prescott (1989). 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. Kirkby (2017b) provides proof that the discretized value function iteration converges to true solution (for value fn and policy fn) under very general conditions. Kirkby (2019) extends this to general equilibrium of heterogeneous agent models of Bewley-Huggett-Aiyagari class.

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. $./VFIToolkit$).

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,
    ReturnFn, Parameters, DiscountFactorParamNames, ReturnFnParamNames, vfoptions);
This section describes the problem it solves, all the inputs and outputs, and provides some further info on using this command.

The Case 1[3] infinite-horizon value function iteration code can be used to solve any problem that can be written in the form

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

[2]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.

[3]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 $v(x,z) = \sup_{y\in\Gamma(x,z)}\{F(x,y,z) + \beta\int_Z v(y,z')Q(z,dz')\}$

subject to

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

where
z $\equiv$ vector of exogenous state variables
a $\equiv$ vector of endogenous state variables
d $\equiv$ 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 14.1) and the Basic Real Business Cycle model (Appendix 14.2).

3.1 Inputs and Outputs

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

- 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[4]
  `pi_z=[0.3,0,3.0,4; 0.2,0.2,0.6; 0.1,0.2,0.7];`
  (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

[4]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. So each row must sum to one.

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);

- Pass a structure *Parameters* containing all of the model parameters.
Parameters.beta=0.96; Parameters.alpha=0.3;
Parameters.gamma=2; Parameters.delta=0.05;

- *ReturnFnParamNames* is a cell containing the names of the parameters used by the ReturnFn (they must appear in same order as used by the ReturnFn).
ReturnFnParamNames={'alpha','gamma','delta'}

- *DiscountFactorParamNames* is a cell containing the names of the discount factor parameter (it is also used for 'exoticpreferences' like Epstein-Zin).
DiscountFactorParamNames={'beta'}

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(V0, n_d, n_a, n_z, d_grid, a_grid, z_grid, pi_z,
        ReturnFn, Parameters, DiscountFactorParamNames, ReturnFnParamNames, [vfoptions]);
```

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 $[length(n_d) + length(n_a), n_a, n_z]$. 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).[5]

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 14.1).

- 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 exogenous 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 $Pr(z^a_{t+1} = z^a_j) = Pr(z^a_{t+1} = z^a_j | z^a_t, z^b_{t+1}, z^b_t)$.)

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

- 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=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.
  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=2
- 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. When using GPU just ignore this option.)
- By default the toolkit assumes standard von-Neumann-Morgenstern preferences. It is possible to overrule this and use 'exotic' preferences instead, if you do this then $DiscountFactorParamNames$ is used to pass the needed additional parameters names. For example, can use Epstein-Zin parameters.
  vfoptions.exoticpreferences=0
- When using graphics cards with single-precision floating point numbers[6] I once had a problem due to rounding errors, so that the 'Policy' instead of containing all integer values contained

[6]NVIDIA deliberately handicap most gaming GPUs to single-precision so they can charge a higher price for 'scientific computing' GPUs with double-precision floating point numbers.

some numbers that were 10^{-15} away from being an integer. Setting this option to 1 forces them all to round to integers. Typically this is unnecessary and in principle could cause unintended errors, so is off by default.
vfoptions.policy_forceintegertype=0

- *piz_strictonrowsaddingtoone* option determines strictness of the internal check that the rows of *pi_z*, the transition matrix for exogenous shocks,. The default (value of 0) is to make sure row sums are within 10^{-13} of 1, by setting value of 1 the check is instead that the row sums are exactly equal to 1. To strict version is desirable, but often fails due to machine precision errors, and so have used the slightly less strict 10^{-13} as default.

3.4 Some Examples

Example: Stochastic Neoclassical Growth Model

Example: Basic Real Business Cycle Model

4 Infinite Horizon Value Function Iteration: Case 2

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

$$\begin{aligned} z' &= \pi(z) \\ a' &= \phi(d,a,z,z') \end{aligned}$$

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.

[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 but this can be done; eg. Bertsekas (1976) provides the finite-horizon results.

1. 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.

2. 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)

3. Create the transition matrices for the exogenous z variables[9]
   pi_z=[0.3,0,3.0,4; 0.2,0.2,0.6; 0.1,0.2,0.7];
   (Often you will want to use the Tauchen Method to create *z_grid* and *pi_z*.)

4. 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

5. Pass a structure $Parameters$ containing all of the model parameters.
   Parameters.beta=0.96; Parameters.alpha=0.3;
   Parameters.gamma=2; Parameters.delta=0.05;

6. $ReturnFnParamNames$ is a cell containing the names of the parameters used by the ReturnFn (they must appear in same order as used by the ReturnFn).
   ReturnFnParamNames={'alpha','gamma','delta'}

7. $DiscountFactorParamNames$ is a cell containing the names of the discount factor parameter (it is also used for 'exoticpreferences' like Epstein-Zin).
   DiscountFactorParamNames={'beta'}

8. 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
   $Case2\_Type = 1$: $\phi(d, a, z', z)$
   $Case2\_Type = 2$: $\phi(d, z', z)$
   $Case2\_Type = 3$: $\phi(d, z')$
   $Case2\_Type = 4$: $\phi(d, a)$
   $Case2\_Type = 5$: $\phi(a'|d, e')$
   You should use the one which makes $\phi(\cdot)$ the smallest dimension possible for your problem as this will be fastest.

9. 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

[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. So each row must sum to one.

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(V0, n_d, n_a, n_z, pi_z,
        Phi_aprimeKron, Case_2_Type, ReturnFn,
        Parameters, DiscountFactorParamNames, ReturnFnParamNames, [vfoptions]);
```

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 Stationary Distributions

Once you solve the Value Function Iteration problem and get the optimal policy function, *Policy*, you can use the $StationaryDist\_Case1$ (or $Case2$) command to calculate the stationary distribution (over the endogenous and exogenous states). $StationaryDist\_Case1$ essentially works by automatically calling two sub-commands in order. The first is $StationaryDist\_Case1\_Simulation$

(and $Case2$) and 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 $StationaryDist\_Case1\_Iteration$ (and $Case2$) is based on iterating directly on the agents distribution until it converges.[10] In practice a good combination of speed and accuracy often comes from using $StationaryDist\_Case1\_Simulation$ to generate a starting guess, which can then be used as an input to $StationaryDist\_Case1$ to get an accurate answer; the method followed by default when using $StationaryDist\_Case1$. The main weakness of this approach is that $StationaryDist\_Case1\_Iteration$ is very memory intensive and so is often not usable with larger grids. More details on these sub-commands, which can easily be called directly, can be found in Appendix E

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 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.
StationaryDist=StationaryDist_Case1(Policy,n_d,n_a,n_z,pi_z, [simoptions]);

The outputs are

- $StationaryDist$: 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 Some further remarks

- A description of the sub-commands $StationaryDist\_Case1\_Simulation$ and $StationaryDist\_Case1\_Iteration$ can be found in appendix

5.3 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

[10]By default the simulation is done on parallel CPUs, and the iteration on the GPU. This combination appears to be the best for speed.

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 CPUs you need to tell it how many cores you have.
  simoptions.ncores=1;
- You can turn off the iteration by setting $simoptions.iterate = 0$, in this case only simulation is used. simoptions.iterate=1;

6 General Equilibrium of Heterogeneous Agent Models

The command *HeteroAgentStationaryEqm_Case1* (and *_Case2*) allows for easy calculation of the price (vector) associated with the general equilibrium of standard Bewley-Huggett-Aiyagari models[11,12].

As well as the standard inputs required for the Value function iteration command the other inputs needed are related to evaluating the general equilibrium conditions.

By default the command will use matlabs fminsearch command to solve the fixed-point problem represented by the general equilibrium requirement. Alternatively the user can set n_p to be non-zero in which case the general equilibrium condition will be evaluated on a grid for prices. This can be useful if you which to check for the possible existence of multiple equilibria and has the advantage of possessing well understood convergence properties (Kirkby, 2019).

In the notation of the toolkit this is any problem for which the competitive equilibrium can be written as

[11] General equilibrium incomplete market heterogeneous agent models with idiosyncratic, but no aggregate, shocks.

[12] In principle one could use these codes to solve fixed-point problems around value function problems more generally, but this would be more advanced.

Definition 1. *A Competitive Equilibrium is an agents value function V_p; agents policy function g; vector of prices p; measure of agents μ_p; such that*

1. *Given prices p, the agents value function V_p and policy function g_p solve the agents problem given by a Case 1 value function (as described in Section 3).*

2. *Aggregates are determined by individual actions: $B_p = \mathcal{A}(\mu_p)$.*

3. *Markets clear (in terms of prices): $\lambda(B_p, p) = 0$.*

4. *The measure of agents is invariant:*

$$\mu_p(a, z) = \int \int \left[\int 1_{a=g_p^{a'}(\hat{a},z)} \mu_p(\hat{a}, z) Q(z, dz') \right] d\hat{a} dz \tag{1}$$

the p subscripts denote that these objects depend on the price(s) p.

6.1 Preparing the model

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

1. Create all of the variables that are needed for the value function problem. These are exactly the same as those used for $ValueFnIter\_Case1$ and the exact same notation applies (See Section 3).
(Namely V0, n_d, n_a, n_z, d_grid, a_grid, z_grid, pi_z,
ReturnFn, Parameters, DiscountFactorParamNames, ReturnFnParamNames, [vfoptions])

2. Define functions for any aggregate variables that are needed by the general equilibrium equation in terms of the micro-level variables, these are evaluated as the integral of the function over the stationary agents distribution. Store all these functions together.
eg., if the only aggregate variables is the integral of the endogenous state, then
FnsToEvaluateFn_1 = @(aprime_val,a_val,s_val;
(You always need the @(...) defined either in this way, or with @($d\_val, ...$) where appropriate. You also need to include parameters where appropriate, done in same way as parameters are passed to the $GeneralEqmEqn$ below.)
Define the names of any parameters used to evaluate the aggregate variables.
FnsToEvaluateParamNames(1).Names={};
(In our example there were none, but an example might be that you want to evaluate tax revenue, which is a tax rate parameter times an endogenous state variable, then the tax rate parameter name will be given here.)
You then store all of these together:
FnsToEvaluate={FnsToEvaluateFn_1}; (For models with multiple FnsToEvaluateFns simply create more in same manner and store them all in $FnsToEvaluate$.)

3. Define functions for how to calculate general equilibrium conditions for which we need to find general equilibrium. Then store these functions together.
Create General Equilibrium equation:
GeneralEqmEqn_1 = @(AggVars,p,param1, param2) p-param1*AggVars(1)^param2
Give the names of the parameters:

GeneralEqmEqnParamNames(1).Names = {'param1', 'param2'}
($param1$ must exist in $Parameter.param1$; similarly for $param2$. '$param1$' is the name of the parameter, e.g. '$alpha$')
You then store all of these together:
GeneralEqmEqns=GeneralEqmEqn_1;
(For models with multiple General Equilibrium Equations simply create more in same manner and store them all in $GeneralEqmEqns$.)

4. Define the names of the prices which are being used to calculate general equilibrium equations. General equilibrium will be where the inputed values of these correspond to those that lead the general equilibrium equations to be zero-valued.
GEPriceParamNames={'r'};
(Technically, in terms of finding the general equilibrium these could be any quantity, not just prices.)

That covers all of the objects that must be created, the only thing left to do is simply call the heterogeneous agent general equlibrium code.

```
[p_eqm,p_eqm_index,GeneralEqmConditions] =
    HeteroAgentStationaryEqm_Case1(V0, n_d, n_a, n_z, n_p, pi_z,d_grid, a_grid, s_grid,
        ReturnFn, FnsToEvaluate, GeneralEqmEqns, Parameters, DiscountFactorParamNames,
        ReturnFnParamNames, FnsToEvaluateParamNames, GeneralEqmEqnParamNames,
        GEPriceParamNames, [heteroagentoptions], [simoptions], [vfoptions]);
```

Once run p_eqm will contain the general equilibrium price vector. $GeneralEqmConditions$ will be a vector containing the values for each of the market clearance conditions (general equilibrium is where these equals zero). p_eqm_index will simply take the value nan, unless you are using the (non-default) grid on prices options, see 'Some further remarks' below.

6.2 Some further remarks

- The $Case2$ command is the same, except that you also need to input $Phi\_aprimeKron$ and $Case2\_Type$ in exactly the same way as for the $ValueFnIter\_Case2$ command.
- By setting n_p to a non-zero value you tell the command that instead of using $fminsearch$ (which uses simplex methods) to solve for the general equilibrium using the general equilibrium conditions it should instead use a grid on prices. n_p should describe the size of the grids for the price vectors (in exactly the same way as n_d contains the size of the grids for the vector of decision variables d). The price grids themselves should be passed as a stacked column vector in $heteroagentoptions.pgrid$, the format of being exactly the same as is used for d_grid or z_grid, ie. a stacked column vector. In this case the output of p_eqm_index is now the grid index corresponding to p_eqm, and $GeneralEqmConditions$ contains the values for each of the general equilibrium conditions at every point on the price grid.

6.3 Options

The options for the value function and stationary distribution can be set using structures called *vfoptions* and *simoptions* and are exactly the same as those for $ValueFnIter\_Case1$ and $StationaryDist\_Case1$. Further options can be passed in *heteroagentoptions* and include *heteroagentoptions.pgrid*,

$heteroagentoptions.verbose$, $heteroagentoptions.multiGEcriterion$ and $heteroagentoptions.fminalgo$. $heteroagentoptions.pgrid$ is used when you want to evaluate the general equilibrium conditions on a grid on prices, see 'Some Further Remarks' just above. $heteroagentoptions.multiGEcriterion$ and $heteroagentoptions.fminalgo$ allow you to change multiple general equilibrium conditions are evaluated (the former) and which algorithm is used to compute the general equilibrium fixed-point problem (the later); both are currently inactive and simply implemented to allow future extensions of the code.

6.4 Examples

See the Aiyagari (1994) example at
github.com/vfitoolkit/VFItoolkit-matlab-examples/tree/master/HeterogeneousAgentModels

7 Transition Path for Heterogeneous Agent Models

The command *TransitionPath_Case1* (and *_Case2*) allows for easy calculation of the path of the price (vector) associated with the general equilibrium transition path of a standard Bewley-Huggett-Aiyagari models.[13]

As well as the standard inputs required for the general equilibrium command the other inputs needed are related to the (series of) parameter changes to be evaluated, including the final value function and the initial stationary agents distribution.

The algorithm is based on the standard shooting algorithm approach to computing general equilibrium transition paths in heterogeneous agent models of the Bewley-Huggett-Aiyagari type.

7.1 Preparing the model

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

1. Many of the inputs are exactly the same as those used to compute the final general equilibrium with the $HeteroAgentStationaryEqm\_Case1$ command and the exact same notation applies (see Section 6).
   (Namely n_d, n_a, n_z, d_grid, a_grid, z_grid, pi_z,
   ReturnFn, FnsToEvaluate, GeneralEqmEqns, Parameters, DiscountFactorParamNames, ReturnFnParamNames, FnsToEvaluateParamNames, GeneralEqmEqnParamNames, PriceParamNames,)
2. Define the number of time periods which you are allowing for the transition path.
   T=100;
3. Define the names of any of the parameters that change over the path.
   ParamPathNames={'alpha'};
   (All other parameters are assumed to remain constant at their initial values.)

[13]General equilibrium incomplete market heterogeneous agent models with idiosyncratic, but no aggregate, shocks.

4. Define the path of the parameters for which you want to calculate the transition.
   ParamPath=0.4*ones(T,1);
   (If you just want a one off unannounced change then this will simply be the final parameter values for T-periods. See 'Some Further Remarks' below for more complex transition paths.)

5. Define the names of the prices that make up the path.
   PricePathName={'r'};
   (This will almost always be the same as PriceParamNames.)

6. Define an initial guess for the price path. The final price value (at period T) *must* be that associated with the final general equilibrium.
   PricePath0=linspace(0.04,0.03,T);
   (eg., from initial price to final price.)

7. Give the initial agent distribution (often, but not necessarily, the stationary distribution associated with initial parameters). StationaryDist_init

8. Give the final value function (you must calculate this beforehand, see example codes). V_final

That covers all of the objects that must be created, the only thing left to do is simply call the heterogeneous agent general equlibrium code.

```
[PricePathNew] =
   TransitionPath_Case1(PricePath0, PricePathNames, ParamPath, ParamPathNames, T, V_final,
StationaryDist_init,
      n_d, n_a, n_z, pi_z, d_grid,a_grid,z_grid, )
      ReturnFn, FnsToEvaluate, GeneralEqmEqns, Parameters, DiscountFactorParamNames,
      ReturnFnParamNames, FnsToEvaluateParamNames, GeneralEqmEqnParamNames, PriceParam-
Names,
      [transpathoptions]);
```

Once run $PricePathNew$ will contain the general equilibrium transition path for prices.

7.2 Some further remarks

- The $Case2$ command is the same, except that you also need to input $Phi\_aprimeKron$ and $Case2\_Type$ in exactly the same way as for the $ValueFnIter\_Case2$ command.

- The command can solve for the transition path for any finite-length sequence of parameter changes that are announced at time 0. This includes one-off unannounced changes (set the entire parameter path equal to the final parameter value), one-off preannounced changes (set the parameter path equal to the initial parameter values for the first few periods, then to the final parameter values from then on), or even a whole series of preannounced changes (set the parameter path equal to the series of changes, then to the final parameter values from then on).

- It is important to check that the choice of T is large enough to ensure convergence (ie., that the tail of the parameter path has been at the final parameter values long enough). The codes return the price path with the final value forced to remain at whatever was set in the initial price path. One can therefore informally check if T is long enough simply by comparing the period $T-1$ price with the period T price (the $T-1$ price should have already converged to the T price). More formally one should check that the agent distribution and value function themselves have converged to the final general equilibrium values (and not just the prices), and that the result is invariant to increasing T.

7.3 Options

Options relating to computing the transition path can be passed in *transpathoptions* and include, with their default values,

- $transpathoptions.tolerance = 10\hat{}(-5)$; Sets the convergence criterion for the general eqm price path.
- $transpathoptions.parallel = 2$; Default is to use the gpu.
- $transpathoptions.exoticpreferences = 0$; Not yet implemented for transition path. Will be used to implement quasi-hyperbolic discounting and epstein-zin preferences.
- $transpathoptions.oldpathweight = 0.9$; Determines the weights used to update the price path. (Must be between 0 and 1.)
- $transpathoptions.weightscheme = 1$; Determines the weighting scheme used to update the price path.
- $transpathoptions.maxiterations = 1000$; If a general eqm price path has not been found within this number of iterations then command will simply terminate.
- $transpathoptions.verbose = 0$; If set to 1 then command will print regular output on progress.

7.4 Examples

See the example based on finding the transition path for a change in the capital share of output (the parameter in the production function) in the model of Aiyagari (1994)
github.com/vfitoolkit/VFItoolkit-matlab-examples/tree/master/HeterogeneousAgentModels

8 Finite Horizon Value Function Iteration: Case 1

The relevant command is
[V,Policy] = ValueFnIter_Case1_FHorz(n_d, n_a, n_z, N_j, d_grid, a_grid, z_grid, pi_z,
    ReturnFn, Parameters, DiscountFactorParamNames, ReturnFnParamNames, [vfoptions]);
This section describes the problem it solves, all the inputs and outputs, and provides some further info on using this command.

The main difference from the infinite horizon codes (other than the obvious that the horizon is finite) is that parameters are allowed to depend on age. To do this simply declare an (age dependent) parameter as a row vector of length N_j. The command automatically determines which parameters are age dependent and which are not and acts accordingly. (The return function also depends on j. Non-default options allow for further age dependence in the exogenous shock process. It is also possible to solve models in which the agents problem is completely different (including number of state variables, etc.) for every age using *age dependent grids* as described in Section 13.)

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

$$V_j(a,z) = \max_{d,a'}\{F_j(d,a',a,z) + \beta_j E[V_{j+1}(a',z')|a,z]\}$$

for $j = 1,,,J$ subject to

$$z' = \pi(z)$$
$$V_{J+1} = 0$$

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 non-zero termination values for V_{J+1} can be implemented via the definition of F_J (see also, $vfioptions.dynasty$). More complex problems can be solved using 'age-dependent grids', see Section 13.

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).

$vfoptions$ 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 the Finite-Horizon Stochastic Consumption-Savings model (Appendix 14.3).

8.1 Inputs and Outputs

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

- 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.
- Define *N_j* as the number of periods in the finite-horizon value function problem.
- 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[14]
  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 $(d, aprime, a, z, j)$ and the parameters and outputs the corresponding value for the return function.
  ReturnFn=@(d,aprime,a,z,j,alpha,gamma) 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);
- Pass a structure *Parameters* containing all of the model parameters. Age dependent parameters are declared as row vectors.
  Parameters.beta=0.96; Parameters.alpha=0.3;
  Parameters.gamma=[1,2,2,1.8];
- *ReturnFnParamNames* is a cell containing the names of the parameters used by the ReturnFn (they must appear in same order as used by the ReturnFn).
  ReturnFnParamNames={'alpha','gamma'}
- *DiscountFactorParamNames* is a cell containing the names of the discount factor parameter (it is also used for conditional survival probabilities).
  DiscountFactorParamNames={'beta'}

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_FHorz(n_d, n_a, n_z, N_j, d_grid, a_grid, z_grid, pi_z,
        ReturnFn, Parameters, DiscountFactorParamNames, ReturnFnParamNames, vfoptions);
```

[14]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.

The outputs are

- V: The value function evaluated on the grid (ie. on $a \times z$). It will be a matrix of size $[n\_a, n\_z, N\_j]$ and at each point it will be the value of the value function evaluated at the corresponding point (a, z, j).
- $Policy$: This will be a matrix of size $[length(n\_d) + length(n\_a), n\_a, n\_z, N\_j]$. For each point (a, z, j) the corresponding entries in $Policy$, namely $Policy(:, a, z, j)$ will be a vector containing the optimal policy choices for (d, a).[15]

8.2 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.
- 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 implementing 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 exogenous 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 $pi\_z = 1$.

8.3 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.

- 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.)
- 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=2

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

- 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. When using GPU just ignore this option.)

- By default the toolkit assumes standard von-Neumann-Morgenstern preferences. It is possible to overrule this and use 'exotic' preferences instead, if you do this then $DiscountFactorParamNames$ is used to pass the needed additional parameters names. For example, can use Epstein-Zin parameters.
  vfoptions.exoticpreferences=0

- The 'low memory' option reduces the memory use, allowing you to solve larger state spaces, but is notably slower. Thus you should only choose to set the low memory option to one if the default is giving an 'out of memory' error.
  vfoptions.lowmemory=0
  (Set vfoptions.lowmemory=0 for much faster but memory intensive codes, vfoptions.lowmemory=1 to reduce memory use.)

- When using graphics cards with single-precision floating point numbers[16] I once had a problem due to rounding errors, so that the 'Policy' instead of containing all integer values contained some numbers that were 10^{-15} away from being an integer. Setting this option to 1 forces them all to round to integers. Typically this is unnecessary and in principle could cause unintended errors, so is off by default.
  vfoptions.policy_forceintegertype=0

- Have agents care about future generations (based directly on the value function of their decendents, rather than indirectly by, e.g., warm-glow bequests) by setting the dynasty option equal to 1. Be aware that in this case the transition from one generation to the next alway occours at the final period of the finite horizon and is based on the decisions (and exogenous shocks) between the final period and the first period (representing the decendent). Notice also that this does not consider the possibility of 'decendent on stochastic death' in periods prior to the final period.
  vfoptions.dynasty=0
  (If using this option you also need to set *simoptions.dynasty*)

- Define the tolerance level to which you wish the value function convergence to reach. Note that this is only actually relevant if you are using *vfoptions.dynasty=1*.
  vfoptions.tolerance=10ˆ(-9)

[16]NVIDIA deliberately handicap most gaming GPUs to single-precision so they can charge a higher price for 'scientific computing' GPUs with double-precision floating point numbers.

8.4 Some Examples

Example: Finite-Horizon Stochastic Consumption Savings

9 Finite Horizon Tools: Simulate Panel Data and Life Cycle Profiles

Commands for simulating panel data and life-cycle profiles from a Finite Horizon Value Fn problem are given by

SimPanelValues=SimPanelValues_FHorz_Case1(InitialDist, Policy, FnsToEvaluate,
    FnsToEvaluateParamNames, Parameters, n_d, n_a, n_z, N_j, d_grid, a_grid, z_grid,
    pi_z, [simoptions]);

and

SimLifeCycleProfiles=SimLifeCycleProfiles_FHorz_Case1(InitialDist, Policy, FnsToEvaluate,
    FnsToEvaluateParamNames, Parameters, n_d, n_a, n_z, N_j, d_grid, a_grid, z_grid,
    pi_z, [simoptions]);

This section describes the simulations involved, the inputs and outputs, and provides some further info on using these commands. They are covered together here as their inputs and functionality have much in common.

Both of these commands are intended to be run based on a $Policy$ that has been output by a $ValueFnIter\_FHorz\_Case1()$ command. $PType$ versions of these commands also exist that allow for different agent permanent/fixed types. Many of the inputs are simply the same as those originally used to solve for $Policy$, namely $Parameters$, n_d, n_a, n_z, N_j, d_grid, a_grid, z_grid, pi_z, and so these are not redescribed here.

simoptions allows you to set some internal options (including parallization), if *simoptions* 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 the replication of Hubbard, Skinner, and Zeldes (1994).

9.1 Inputs and Outputs

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

- To be able to simulate a panel you must first declare in what state agents will be 'born'. This is the role of $InitialDist$. Typically it will be a probability distribution across the state space (a, z) at age $j = 1$. This will take the form of a matrix of dimensions $n\_a - by - n\_z$ the elements of which sum to one.
  InitialDist=zeros(n_a,n_z); InitialDist(1,ceil(n_z)/2)=1;
  (In this example it is assumed agents are born with least number of assets (n_a one-dimensional) and median values of the shocks.)
  ($InitialDist$ is allowed to further depend on j, in which case it will be $n\_a-by-n\_z-by-N\_j$. When the N_j dimension is not declare it is simply assumed by default that all agents are 'born' with $j = 1$.)
- The policy function $Policy$ is simply that created as an output by $ValueFnIter\_Case1\_FHorz()$. It contains the 'indexes' for the optimal policy (choices for d and a' variables) at each point in the discretized state space.
- Define functions for the variables you wish to create panel data (or life-cycle profiles) for.

Store all these functions together.
eg., for a panel which is just the values of the endogenous state, then
`FnsToEvaluateFn_1 = @(aprime_val,a_val,s_val) a_val;`
(You always need the @(...) defined either in this way, or with @(d_val, ...) where appropriate. You also need to include parameters where appropriate.)
Define the names of any parameters used to evaluate the aggregate variables.
`FnsToEvaluateParamNames(1).Names={};`
(In our example there were none, but an example might be that you want to evaluate tax revenue, which is a tax rate parameter times an endogenous state variable, then the tax rate parameter name will be given here.)
You then store all of these together:
`FnsToEvaluate={FnsToEvaluateFn_1};`

- For how to declare *Parameters*, *n_d*, *n_a*, *n_z*, *N_j*, *d_grid*, *a_grid*, *z_grid*, *pi_z* check the descriptions of the finite horizon value function iteration command in Section 8.

That covers all of the objects that must be created, the only thing left to do is simply call the commands.

```
SimPanelValues=SimPanelValues_FHorz_Case1(InitialDist, Policy, FnsToEvaluate,
        FnsToEvaluateParamNames, Parameters, n_d, n_a, n_z, N_j, d_grid, a_grid, z_grid,
        pi_z, [simoptions]);
```

and

```
SimLifeCycleProfiles=SimLifeCycleProfiles_FHorz_Case1(InitialDist, Policy, FnsToEvaluate,
        FnsToEvaluateParamNames, Parameters, n_d, n_a, n_z, N_j, d_grid, a_grid, z_grid,
        pi_z, [simoptions]);
```

The outputs are

- *SimPanelValues*: A simulated panel data set containing the values of the 'FnsToEvaluate'. It will be a matrix of size [$length(FnsToEvaluate)$,N_j, $NSims$] and at each point it will be the value of the 'FnsToEvaluateFn' for an age and individual,
- *SimLifeCycleProfiles*: Creates life-cycle profiles for the values of the 'FnsToEvaluate'. It will be a matrix of size [$length(FnsToEvaluate)$, N_j, 23]. The first dimension indexes the variable, the second is the age, and the third indexes mean/median/ventile. For example, $SimLifeCycleProfiles(3,:,1)$ will be the life-cycle profile for the mean of the third variable in 'FnsToEvaluate'.
  ($simoptions.lifecyclepercentiles = 20$ is ventiles, and can be used to control the number of percentiles reported (always including the min and max values), so $simoptions.lifecyclepercentiles = 4$ will return quartiles, and $simoptions.lifecyclepercentiles = 0$ then only the mean and median will be returned.)

9.2 Some further remarks

- Models where d is unnecessary (only a' need be chosen): set $n\_d = 0$ and $d\_grid = 0$, the code will take care of the rest.

- *SimLifeCycleProfiles* simply performs the same panel data simulation as *SimPanelValues* but then returns the age-conditional mean (median/percentile/etc.) rather than returning the actual panel simulation.
- Panel simulation commands presently just ignore stochastic survivial probabilities (the value function and OLG commands do account for them).

9.3 Options

Optionally you can also input a further argument, a structure called *simoptions*, which allows you to set various internal options. Following is a list of the *simoptions*, the values to which they are being set in this list are their default values.

- For reasons of speed simulations are always performed on CPU (or parallel CPUs). Setting *simoptions.parallel* = 2 simply means that the output will be transferred to the GPU before it is returned (and that inputs are assumed to be on the GPU).
  simoptions.parallel=2
- If you want feedback set to one, else set to zero
  simoptions.verbose=0
- Number of periods for which each individual agent simulation lasts.
  simoptions.simperiods=N_j
- Number of individual agents simulations that make up the panel data.
  simoptions.numbersims=10ˆ4
  (Note: Life-cycle profiles are themselves based on a simulated panel so this *simoptions* is relevant)
- Number of percentiles for which life-cycle profiles are given (not relevant to *SimPanelValues* command), in addition to the mean, median, and minimum (note that maximum will always be the p-th of the p percentiles)
  simoptions.lifecyclepercentiles=20;

9.4 Some Examples

See the replication of Hubbard, Skinner, and Zeldes (1994) at github: https://github.com/vfitoolkit/vfitoolkit-matlab-replication/tree/master/HubbardSkinnerZeldes1994.

10 General Equilibrium of Overlapping-Generations models

The command *HeteroAgentStationaryEqm_Case1_FHorz* (and *_Case2_FHorz*) allows for easy calculation of the price (vector) associated with the general equilibrium of standard Overlapping-Generations models[17,18].

[17]General equilibrium finite-horizon-value-function incomplete market heterogeneous agent models with idiosyncratic, but no aggregate, shocks.

[18]In principle one could use these codes to solve fixed-point problems around finite-horizon value function problems more generally, but this would be more advanced.

As well as the standard inputs required for the Value function iteration command the other inputs needed are related to evaluating the general equilibrium conditions and an initial agents distribution (at birth).

By default the command will use matlabs fminsearch command to solve the fixed-point problem represented by the general equilibrium requirement. Alternatively the user can set n_p to be non-zero in which case the general equilibrium condition will be evaluated on a grid for prices. This can be useful if you which to check for the possible existence of multiple equilibria.

In the notation of the toolkit this is any problem for which the competitive equilibrium can be written as

Definition 2. *A Competitive Equilibrium is an agents finite-horizon value function $V_p = \{V_{1,p}, ...V_{J,p}\}$; agents policy function $g_p = \{g_{1,p}, ..., g_{J,p}\}$; vector of prices p; measure of agents $\mu_p = \{\mu_{1,p}, ..., \mu_{J,p}\}$; such that*

1. *Given prices p, the agents value function V_p and policy function g_p solve the agents problem given by a Case 1 finite-horizon value function problem (as described in Section 8).*
2. *Aggregates are determined by individual actions: $B_p = \mathcal{A}(\mu_p)$.*
3. *Markets clear (in terms of prices): $\lambda(B_p, p) = 0$.*
4. *The measure of agents is given by their decisions:*
$$\mu_{j+1,p}(a, z) = \int\int \left[\int 1_{a=g^{a'}_{j,p}(\hat{a},z)} \mu_{j,p}(\hat{a}, z) Q_j(z, dz')\right] d\hat{a}dz, \quad j=1,...,J-1 \tag{2}$$

the p subscripts denote that these objects depend on the price(s) p.[19]

10.1 Preparing the model

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

1. Create all of the variables that are needed for the value function problem. These are exactly the same as those used for $ValueFnIter\_Case1\_FHorz$ and the exact same notation applies (See Section 8).
(Namely jequaloneDist,AgeWeights,n_d, n_a, n_z, N_j, d_grid, a_grid, z_grid, pi_z, ReturnFn, Parameters, DiscountFactorParamNames, ReturnFnParamNames, [vfoptions])
2. Define the distribution of agents at birth
jequaloneDist=ones(n_a,n_z)/(prod(n_a)*prod(n_z));
Must sum to one. This example just assumes a uniform distribution of all agents across all possible states at 'birth'.

[19]Note that the present definition requires agents to be 'reset' at birth, rather than inheriting all the characteristics of the deceased. This is done as in most situations where inheritance is desired a degree of altruism from parents to decendents is likely also desired and so the value function problem becomes infinite-horizon (modelling a dynasty) and so the *HeteroAgentStationaryEqm_Case1* can be used to solve these. The case in which inheritance of characteristics is desired in combination with the utility of one generation not depending directly on the utility of the next generation to which it will be passing on it's characteristics is not covered by the VFI Toolkit but could be implemented as a simple fixed point problem wrapped around *HeteroAgentStationaryEqm_Case1_FHorz* that looks for the fixed point that makes the initial age j=1 distribution 'match' the distribution of 'deceased' agents.

3. Define the mass of agents for each age group
   AgeWeights=ones(1,J)/J;
   Must sum to one. This is needed for population growth and age-dependent mortality. See Example codes for how to use it. If you have neither population growth nor age-dependent mortality it is just the example vector given here.

4. Define functions for any aggregate variables that are needed by the general equilibrium equation in terms of the micro-level variables, these are evaluated as the integral of the function over the stationary agents distribution. Store all these functions together.
   eg., if the only aggregate variables is the integral of the endogenous state, then
   FnsToEvaluateFn_1 = @(aprime_val,a_val,s_val;
   (You always need the @(...) defined either in this way, or with $@(d\_val, ...)$ where appropriate. You also need to include parameters where appropriate, done in same way as parameters are passed to the *GeneralEqmEqn* below.)
   Define the names of any parameters used to evaluate the aggregate variables.
   FnsToEvaluateParamNames(1).Names={};
   (In our example there were none, but an example might be that you want to evaluate tax revenue, which is a tax rate parameter times an endogenous state variable, then the tax rate parameter name will be given here.)
   You then store all of these together:
   FnsToEvaluate={FnsToEvaluate_1}; (For models with multiple FnsToEvaluate simply create more in same manner and store them all in *FnsToEvaluate*.)

5. Define functions for how to calculate general equilibrium conditions for which we need to find general equilibrium. Then store these functions together.
   Create General Equilibrium equation:
   GeneralEqmEqn_1 = @(AggVars,p,param1, param2) param1*AggVars(1)^param2
   Give the names of the parameters:
   GeneralEqmEqnParamNames(1).Names = {'param1', 'param2'}
   (*param*1 must exist in *Parameter.param*1; similarly for *param*2. '*param*1' is the name of the parameter, e.g. '*alpha*')
   You then store all of these together:
   GeneralEqmEqns=GeneralEqmEqn_1;
   (For models with multiple General Equilibrium Equations simply create more in same manner and store them all in *GeneralEqmEqns*.)

6. Define the names of the prices which are being used to calculate general equilibrium equations. General equilibrium will be where the inputed values of these correspond to those that lead the general equilibrium equations to be zero-valued.
   GEPriceParamNames={'r'};
   (Technically, in terms of finding the general equilibrium these could be any quantity, not just prices.)

That covers all of the objects that must be created, the only thing left to do is simply call the heterogeneous agent general equlibrium code.

```
[p_eqm,p_eqm_index,GeneralEqmConditions] =
  HeteroAgentStationaryEqm_Case2(V0, n_d, n_a, n_z, n_p, pi_z,d_grid, a_grid, s_grid,
    ReturnFn, FnsToEvaluate, GeneralEqmEqns, Parameters, DiscountFactorParamNames,
    ReturnFnParamNames, FnsToEvaluateParamNames, GeneralEqmEqnParamNames, GEPri-
```

ceParamNames,
    [heteroagentoptions], [simoptions], [vfoptions]);

Once run p_eqm will contain the general equilibrium price vector. $GeneralEqmConditions$ will be a vector containing the values for each of the market clearance conditions (general equilibrium is where these equals zero). p_eqm_index will simply take the value nan, unless you are using the (non-default) grid on prices options, see 'Some further remarks' below.

10.2 Some further remarks

- The $Case2$ command is the same, except that you also need to input $Phi\_aprimeKron$ and $Case2\_Type$ in exactly the same way as for the $ValueFnIter\_Case2$ command.
- By setting n_p to a non-zero value you tell the command that instead of using $fminsearch$ (which uses simplex methods) to solve for the general equilibrium using the general equilibrium conditions it should instead use a grid on prices. n_p should describe the size of the grids for the price vectors (in exactly the same way as n_d contains the size of the grids for the vector of decision variables d). The price grids themselves should be passed as a stacked column vector in $heteroagentoptions.pgrid$, the format of being exactly the same as is used for d_grid or z_grid, ie. a stacked column vector. In this case the output of p_eqm_index is now the grid index corresponding to p_eqm, and $GeneralEqmConditions$ contains the values for each of the general equilibrium conditions at every point on the price grid.

10.3 Options

The options for the value function and stationary distribution can be set using structures called *vfoptions* and *simoptions* and are exactly the same as those for $ValueFnIter\_Case1$ and $StationaryDist\_Case1$. Further options can be passed in *heteroagentoptions* and include *heteroagentoptions.pgrid*, *heteroagentoptions.verbose*, $heteroagentoptions.multiGEcriterion$ and $heteroagentoptions.fminalgo$. *heteroagentoptions.pgrid* is used when you want to evaluate the general equilibrium conditions on a grid on prices, see 'Some Further Remarks' just above. $heteroagentoptions.multiGEcriterion$ and $heteroagentoptions.fminalgo$ allow you to change multiple general equilibrium conditions are evaluated (the former) and which algorithm is used to compute the general equilibrium fixed-point problem (the later); both are currently inactive and simply implemented to allow future extensions of the code.

10.4 Examples

See simple example at See the example based on Huggett (1996) at:
github.com/vfitoolkit/VFItoolkit-matlab-examples/tree/master/OLG.
Also, replication of Huggett (1996) at:
github.com/vfitoolkit/vfitoolkit-matlab-replication/tree/master/Huggett1996.
See replication at Huggett and Ventura (2000) at FORTHCOMING

11 Permanent/Fixed Agent Types

Many Finite Horizon commands also allow for different permanent (a.k.a) fixed types of agents. Parameters that depend on permanent type are easy to implement, simply give the parameter as a vector (or if it depends on both age —in a finite horizon model— and type then as a matrix). The VFI Toolkit automatically recognises such parameters as depending on permanent type. Other aspects of the model that might depend on type, such as grids, or exogenous shock processes can also be easily implemented. The replication codes for Hubbard, Skinner, and Zeldes (1994) provide a good example of how these work in practice. Typically you just call the standard command but with '_PType' included in the command name.

12 Transition Path for OLG models

The command *TransitionPath_Case1_Fhorz* (*_Case2* is not yet implemented) allows for easy calculation of the path of the price (vector) associated with the general equilibrium transition path of a standard Overlapping-Generation (OLG) models.[20]

As well as the standard inputs required for the general equilibrium command the other inputs needed are related to the (series of) parameter changes to be evaluated, including the final value function and the initial stationary agents distribution.

The algorithm is based on the standard shooting algorithm approach to computing general equilibrium transition paths in heterogeneous agent models of the OLG type.

12.1 Preparing the model

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

1. Many of the inputs are exactly the same as those used to compute the final general equilibrium with the $HeteroAgentStationaryEqm\_Case1\_FHorz$ command and the exact same notation applies (see Section 10).
(Namely jequaloneDist,AgeWeights,n_d, n_a, n_z, N_j, d_grid, a_grid, z_grid, pi_z, ReturnFn, FnsToEvaluate, GeneralEqmEqns, Parameters, DiscountFactorParamNames, ReturnFnParamNames, FnsToEvaluateParamNames, GeneralEqmEqnParamNames, PriceParamNames,)

2. Define the number of time periods which you are allowing for the transition path.
T=100;

3. Define the names of any of the parameters that change over the path.
ParamPathNames={'alpha'};
(All other parameters are assumed to remain constant at their initial values.)

4. Define the path of the parameters for which you want to calculate the transition.
ParamPath=0.4*ones(T,1);
(If you just want a one off unannounced change then this will simply be the final parameter values for T-periods. See 'Some Further Remarks' below for more complex transition paths.)

5. Define the names of the prices that make up the path.
PricePathName={'r'};
(This will almost always be the same as PriceParamNames.)

6. Define an initial guess for the price path. The final price value (at period T) *must* be that associated with the final general equilibrium.
PricePath0=linspace(0.04,0.03,T);
(eg., from initial price to final price.)

7. Give the initial agent distribution (often, but not necessarily, the stationary distribution associated with initial parameters). StationaryDist_init

[20]General equilibrium incomplete market heterogeneous agent models with idiosyncratic, but no aggregate, shocks. Finitely lived households.

8. Give the final value function (you must calculate this beforehand, see example codes). V_final

That covers all of the objects that must be created, the only thing left to do is simply call the heterogeneous agent general equlibrium code.

```
[PricePathNew] =
  TransitionPath_Case1_Fhorz(PricePath0, PricePathNames, ParamPath, ParamPathNames, T,
    V_final, StationaryDist_init, n_d, n_a, n_z, N_j, pi_z, d_grid,a_grid,z_grid,
    ReturnFn, FnsToEvaluate, GeneralEqmEqns, Parameters, DiscountFactorParamNames,
    ReturnFnParamNames, FnsToEvaluateParamNames, GeneralEqmEqnParamNames,
    [transpathoptions]);
```

Once run, $PricePathNew$ will contain the general equilibrium transition path for prices.

12.2 Some further remarks

- The $Case2$ command has not yet been implemented.
- A changing initial distribution (from which agents are born) is not yet allowed for.
- The command can solve for the transition path for any finite-length sequence of parameter changes that are announced at time 0. This includes one-off unannounced changes (set the entire parameter path equal to the final parameter value), one-off preannounced changes (set the parameter path equal to the initial parameter values for the first few periods, then to the final parameter values from then on), or even a whole series of preannounced changes (set the parameter path equal to the series of changes, then to the final parameter values from then on).
- It is important to check that the choice of T is large enough to ensure convergence (ie., that the tail of the parameter path has been at the final parameter values long enough). The codes return the price path with the final value forced to remain at whatever was set in the initial price path. One can therefore informally check if T is long enough simply by comparing the period $T-1$ price with the period T price (the $T-1$ price should have already converged to the T price). More formally one should check that the agent distribution and value function themselves have converged to the final general equilibrium values (and not just the prices), and that the result is invariant to increasing T.

12.3 Options

Options relating to computing the transition path can be passed in *transpathoptions* and include, with their default values,

- $transpathoptions.tolerance = 10\hat{}(-5)$; Sets the convergence criterion for the general eqm price path.
- $transpathoptions.parallel = 2$; Default is to use the gpu.
- $transpathoptions.exoticpreferences = 0$; Not yet implemented for transition path. Will be used to implement quasi-hyperbolic discounting and epstein-zin preferences.

- *transpathoptions.oldpathweight* = 0.9; Determines the weights used to update the price path. (Must be between 0 and 1.)
- *transpathoptions.weightscheme* = 1; Determines the weighting scheme used to update the price path.
- *transpathoptions.maxiterations* = 1000; If a general eqm price path has not been found within this number of iterations then command will simply terminate.
- *transpathoptions.verbose* = 0; If set to 1 then command will print regular output on progress.

12.4 Examples

See the example based on finding the transition path for a change in the capital share of output (the parameter in the production function) in the model of Aiyagari (1994) github.com/vfitoolkit/VFItoolkit-matlab-examples/tree/master/HeterogeneousAgentModels

13 OLG models with 'Age-Dependent-Grids'

Some Finite-Horizon value function problems involve solving different problems at each age. The VFI toolkit handles this with 'age-dependent grids'. As an example, suppose each household lives for three periods. When young they make decisions on whether to go to university. When middle-age they make decisions on how many hours to work and how much to save. When retired they make decisions on how much to save. Clearly the problem, and the number of state variables, changes with age. The age dependent grids allows these problems to be easily solved, and not just the value function problem, but also all the other standard OLG codes such as finding the stationary agents distribution, computing the general equilibrium, and simulating panel data, model statistics, and life-cycle profiles.[21]

The following explantion largely assumes that you know how to set up an OLG model *without* age-dependent grids, as explained in Section 8.

Actually using the age-dependent grids is done using the existing command relevant command ValueFnIter_Case2_FHorz, but the inputs that are passed for the grids and transition matrix are slightly different, and it is important to use vfoptions.agedependentgrids as explained below. The relevant command is thus

```
[V,Policy] = ValueFnIter_Case2_FHorz(n_d, n_a, n_z, N_j, d_gridfn, a_gridfn, z_gridfn,
    AgeDepGridsParamNames, ReturnFn, Parameters, DiscountFactorParamNames,
    ReturnFnParamNames, vfoptions);
```

This section describes the problem it solves, all the inputs and outputs, and provides some further info on using this command.

The main difference from the standard Case 2 finite horizon codes is that the problem solved by agents can be completely different for each age. In particular, the number of grid points, and hence implicitly the dimensions, of the decision variables d, the endogenous states a, and the exogenous states z (and their transitions) are all allowed to depend on age j.

Using age-dependent grids the Case 2 finite-horizon value function iteration code can be used to solve any problem that can be written in the form

$$V_j(a_j, z_j) = \max_{d_j}\{F_j(d_j, a_j, z_j) + \beta_j E[V_{j+1}(a_{j+1} = \phi'_a(d_j, a_j, z_j, z_{j+1}), z_{j+1})|a_j, z_j]\}$$

for $j = 1,,, J$ subject to

$$z_{j+1} = \pi_j(z_j)$$
$$V_{J+1} = 0$$

where
$z_j \equiv$ vector of exogenous state variables
$a_j \equiv$ vector of endogenous state variables
$d_j \equiv$ vector of decision variables
notice that any constraints on d_j, a_j, & a_{j+1} can easily be incorporated into this framework by building them into the return function. Note that non-zero termination values for V_{J+1} can be implemented via the definition of F_J (also see $vfioptions.dynasty$).

[21]Simulated panel data, model statistics, and life-cycle profiles should be treated with care, as depending on how the model is set up the age-dependent problems may mean that some 'variables' change meaning depending on age. This is discussed in more detail below.

The main inputs the value function iteration command requires are the discount rate; the return function F; and the ϕ function that determines next periods state given the decisions.

It also requires to info on how many variables make up d, a and z (and the grids onto which they should be discretized).

$vfoptions$ 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.

13.1 Inputs and Outputs

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

- Define which of the decision variables, endogenous variables, and exogenous variables depend on age. For example vfioptions.agedependentgrids=[1,0,1] means that the decision variables change with age (the first 1), the endogenous variables do not (the zero), and the exogenous variables do (the final 1).
- Define n_a, n_z, and n_d as follows. In this example n_a is age independent and so can be defined as normal except it must now be a column vector rather than a row vector. Since n_d and n_z depend on age they are declared with each row representing a varible, and the columns representing age. So $n\_a = [10, 10; 5, 2]$ would mean there are two a variables, the first endogenous has 10 grid points at age 1 and 10 grid points at age 2 (the first row, note that the interpretation of those 10 grid points could still differ by age), while the second endogenous variable has 5 grid points at age 1 and 2 grid points at age 2. By setting the number of grid points to 1 a variable essentially disappears, so if there is an exogenous variable at age 1 with 15 grid points but no exogenous variable at age 2 we would have $n\_z = [15, 1]$.
- Define N_j as the number of periods in the finite-horizon value function problem.
- Create the (discrete state space) grids for each of the d, a & z variables, for those which do not depend on age you can define them as usual (stacked column vectors), for those that depend on age you must instead define a function that has age (and potentially other parameters) as an input, and returns the grid for that age (in case of exogenous variables z the function must also return the transition matrix for that age),
  d_gridfn, a_gridfn, z_gridfn
  See example codes for more detail.
- AgeDepGridsParamNames must be defined to contain the names of all the parameters used by each of d_gridfn, a_gridfn, and z_gridfn. This is done for each of d, a, and z as follows
  AgeDepGridsParamNames.d_grid={ };
  AgeDepGridsParamNames.a_grid={'agej','theta','gamma'};
  AgeDepGridsParamNames.z_grid={'agej','theta','sigma','rho'};
  (In the current example I am imagining that the d variables do not actually depend on age; this would actually mean that the (empty) parameter names for d_grid are unused and you could actually skip declaring them.)
- 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 illus-

trations of how to do this. It should be a Matlab function that takes as inputs various values for $(d, aprime, a, z, j)$ and the parameters and outputs the corresponding value for the return function. Notice that the 'age-dependent' aspects of the return function are all coded inside the *ReturnFn*, there is only one function.
ReturnFn=@(d,aprime,a,z,j,alpha,gamma) ReturnFunction_AMatlabFunction

- Pass a structure *Parameters* containing all of the model parameters. Age dependent parameters are declared as row vectors.
  Parameters.beta=0.96; Parameters.alpha=0.3;
  Parameters.gamma=[1,2,2,1.8];

- *ReturnFnParamNames* is a cell containing the names of the parameters used by the ReturnFn (they must appear in same order as used by the ReturnFn).
  ReturnFnParamNames={'alpha','gamma'}

- *DiscountFactorParamNames* is a cell containing the names of the discount factor parameter (it is also used for conditional survival probabilities).
  DiscountFactorParamNames={'beta'}

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_Case2_FHorz(n_d, n_a, n_z, N_j, d_gridfn, a_gridfn, z_gridfn,
      AgeDepGridsParamNames, ReturnFn, Parameters, DiscountFactorParamNames,
      ReturnFnParamNames, vfoptions);
```

The *outputs* are substantially more complicated with age dependent grids. The two outputs are still the value function V and the policy function *Policy*, but now these are structures and the age 7 value function is $V.j007$ while the age 12 policy function is $V.j012$. That is, the value function for any age can be found by 'V dot j (three digit number for age)', analagously for the optimal policy function. The age-dependent value functions are themselves in exactly the same form that the VFI toolkit stores standard value functions; same for the age-dependent optimal policy functions.

13.2 Some further remarks

- Age-dependent grids are only available with *Case_2*. This is because it seems unlikely there are many applications of a *Case_1* that is age-dependent. They can anyway be solved by setting them up as if they were *Case_2*, they would just be able to be coded to run a little faster with a dedicated command.

- Almost all of the remarks for when the grids do not depend on age remain relevant.

13.3 Options

When using age dependent grids inputting *vfoptions* is a required input, and as well as telling the toolkit to use the age dependent grids it also 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). Other than *vfoptions.agedependentgrids* all the other options are exactly as for the standard *ValueFnIter_Case2_FHorz* command, see Section 8 for details.

14 Examples

The examples 14.1 and 14.2 provide two basic examples of how to set up and call the value function iteration.

The example 14.3 and provides a basic example of how to set up and call the finite horizon value function iteration.

The codes implementing these and other examples can be found at github.com/vfitoolkit/vfitoolkit-matlab-examples, you can find codes using the VFI Toolkit to replicate some classic papers at github.com/vfitoolkit/vfitoolkit-matlab-replication. These replictions include doing things like simulating time series, computing standard business cycle statistics, and solving heterogeneous agent models with general equilibrium.

14.1 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

$$c + i = exp(z)k^{\alpha} \tag{3}$$
$$k' = (1-\delta)k + i \tag{4}$$
$$z' = \rho z + \epsilon' \quad , \epsilon \overset{iid}{\sim} N(0,\sigma_{\epsilon}^2) \tag{5}$$

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.

You can find a copy of the codes at github.com/vfitoolkit/VFItoolkit-matlab-examples

The code $StochasticNeoClassicalGrowthModel.m$ 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 and uses $StochasticNeoClassicalGrowthModel_ReturnFn$ which implements the return function.

14.2 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. Below that are two further codes that providing examples of what one can then do with the solution of the value function iteration problem. The first demonstrates how to use the policy function created by the Value Function (Case 1) command to reproduce the relevant findings of Aruoba, Fernandez-Villaverde, and Rubio-Ramirez (2006) looking at the sizes of numerical errors; this serves the added purpose of allowing you to easily experiment with how measures of the numerical errors change with the choice of grids. The second provide examples of how to use further Toolkit commands to simulate time series and calculate Standard Business Cycle Statistics.

The value function problem to be solved is,

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

subject to

$$c + i = exp(z)k^{\alpha}l^{1-\alpha} \tag{6}$$

$$k' = (1-\delta)k + i \tag{7}$$

$$z' = \rho z + \epsilon' \quad , \epsilon \overset{iid}{\sim} N(0,\sigma_{\epsilon}^2) \tag{8}$$

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 codes implementing this model are available at github.com/vfitoolkit/VFItoolkit-matlab-examples

The code *BasicRealBusinessCycleModel.m* solves the model, using parallelization on GPU, and uses *BasicRealBusinessCycleModel_ReturnFn.m* which defines the return function.
There are also *BasicRealBusinessCycleModel_BusinessCycleStatistics.m* which reimplements solving the model and then goes on to use some of the other Toolkit commands to simulate time series and calculate the Standard Business Cycle statistics.
Also *BasicRealBusinessCycleModel_NumericalErrors.m* which reimplements solving the model and then goes on to use the output to reproduces a number of relevant elements from Tables & Figures in Aruoba, Fernandez-Villaverde, and Rubio-Ramirez (2006) relating to the accuracy of numerical solutions.

14.3 Finite Horizon Stochastic Consumption Savings Model

A basic Finite-Horizon Stochastic Consumption Savings problem is given. The household lives $J = 10$ periods. Wage income, W consists of two parts, a deterministic function of age W_j and a stochastic component W_z. The household makes a simple choice between consumption c and savings a. Savings earn interest rate r, and the future is discounted at age-independent rate β.

The value function problem to be solved is,

$$V_j(a, z) = \sup_{a'} \{ \frac{c^{1-\gamma} - 1}{1 - \gamma} + \beta E[V_{j+1}(a', z')|z]\}$$

$j = 1, ..., J$, subject to

$$c + a' = W + a(1 + r) \tag{9}$$

$$W = W_j + W_z \tag{10}$$

$$log(W_z') = \rho log(W_z) + \epsilon_z' \quad , \epsilon \overset{iid}{\sim} N(0, \sigma_\epsilon^2) \tag{11}$$

$$V_{11} = 0 \tag{12}$$

Notice that the parameters defining the deterministic component of income as a function of age, W_j, are parameters that depend on age and so are created as a row vector. The VFI Toolkit automatically understands that this row vector represents an age-dependent parameters, while the other parameters are not age dependent.

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

The codes implementing this model are available at github.com/vfitoolkit/VFItoolkit-matlab-examples

The code $FiniteHorzStochConsSavings.m$ solves the model, using parallelization on GPU, and uses $FiniteHorzStochConsSavings\_ReturnFn.m$ which defines the return function.

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.

Dimitri Bertsekas. *Dynamic Programming and Stochastic Control*. Academic Press, 1976.

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.

Glenn Hubbard, Jonathan Skinner, and Stephen Zeldes. The importance of precautionary motives in explaining individual and aggregate saving. *Carnegie-Rochester Conference Series on Public Policy*, 40(1):59–125, 1994.

Mark Huggett. Wealth distribution in life-cycle economies. *Journal of Monetary Economics*, 38: 469–494, 1996.

Mark Huggett and Gustavo Ventura. Understanding why high income households save more than low income households. *Journal of Monetary Economics*, 45(2):361–397, 2000.

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

Robert Kirkby. Transition paths for Bewley-Huggett-Aiyagari models: Comparison of some solution algorithms. *VUW-SEF Working Paper*, 01-2017:1–27, 2017a.

Robert Kirkby. Convergence of discretized value function iteration. *Computational Economics*, 49(1):117–153, 2017b.

Robert Kirkby. Bewley-Huggett-Aiyagari models: Computation, simulation, and uniqueness of general equilibrium. *Macroeconomic Dynamics*, 23(6):2469–2508, 2019.

Nancy Stokey, Robert E. Lucas, and Edward C. Prescott. *Recursive Methods in Economic Dynamics*. Harvard University Press, 1989.

Whitt. Approximations of dynamic programs, i. *Mathematics of Operations Research*, 3:231–243, 1978. There is also a II (1979), but I do not refer to it.

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 $Pr(p_t) = Pr(q_t, q_{t-1}, p_{t-1})$[22]. 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.

[22]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 transition matrices for exogenous variables based on the Tauchen Method for discrete state space approximations of VARs. The main command $TauchenMethod$ is for creating the approximation directly from the parameters of an AR(1), and this is now described.

The Toolkit also includes commands $TauchenMethodVAR$ for approximation (quadrature) of a VAR(1), and *RouwenhorstMethod* which provides an alternative approximations of AR(1) processes that outperforms the Tauchen method when the autocorrelation coefficient (referred to below as ρ) takes a value near one.

Namely,

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

C.1 TauchenMethod

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

$$z_t = \mu + \rho z_{t-1} + \epsilon_t \quad \epsilon_t \sim^{iid} N(0, \sigma^2) \tag{13}$$

the command is,
[z_grid, pi_z]=TauchenMethod(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 13, 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. To ensure that Howards Improvement Algorithm does not interfere with the robustness of the algorithm convergence it is turned off after a number of uses (this is not just a theoretical issue, it can break convergence with small grids).

Turning off Howards Improvement Algorithm (which toolkit does automatically) is essential if you plan to use that solution for the final value function in a general equilibrium transition path computation. (See pg 15 of Kirkby (2017a) on transition paths for practical example of what goes wrong otherwise). Proofs that Howards Improvement Algorithm does not interfere with the convergence of the value function iteration are technically correct but not practically relevant: they are based on 'asymptotics' which the algorithm will never actually reach. The issue can be seen looking at second last line of proof in Stachurski textbook, the proof shows that Howards' won't matter, but this depends on using Howards an infinite number of times at each iteration of the value function, while in practice you will obviously only ever do a finite number of iterations.

E StationaryDist Sub-commands

E.1 StationaryDist_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.

E.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.
StationaryDist=StationaryDist_Case1_Simulation(Policy,n_d,n_a,n_z,pi_z, [simoptions]);

The outputs are

- *StationaryDist*: 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).

E.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=1;
  (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;

E.2 StationaryDist_Case1_Iteration

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 *StationaryDist*; call it *InitialDist* which can be any matrix of size $[n\_a, n\_z]$ which sums to one.

E.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, *InitialDist*, for the steady-state distribution. This can be any $[n\_a, n\_z]$ matrix which sums to one.
  InitialDist=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.
StationaryDist=StationaryDist_Case1_Iteration(InitialDist,Policy,n_d,n_a,n_z,pi_z,[simoptions])

The outputs are

- *StationaryDist*: The stationary 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).

E.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 Imrohoroglu (1989) , pg 1374)].
- If the model has idiosyncratic and aggreate uncertainty and individual state is independent of the aggregate state then $pi\_sz = kron(pi\_s, pi\_z)$;.

E.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)
- Set a maximum number of iterations that will be taken when trying to converge to the stationary distribution. If convergence is not reached before this number of iterations the convergence will be terminated and a warning message printed to screen to effect that this maximum number of iterations has been reached.
  simoptions.maxit=5*10ˆ4;
  (In my experience, after simulating an initial guess, if you need more that 5*10ˆ4 iterations to reach the stationary distribution 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=2;

F Some Issues to be Aware Of

F.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.

F.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.