

VFI Toolkit: Pseudocodes

Robert Kirkby*

April 23, 2025

*Kirkby: Victoria University of Wellington. Please address all correspondence about this article to Robert Kirkby at <robertdkirkby@gmail.com>, robertdkirkby.com, vfitoolkit.com.

Contents

1	Introduction	1
2	Value Function Iteration: Solving for V and Policy	1
2.1	Infinite Horizon: decision variable, markov shock	1
2.1.1	Refinement	2
2.2	Finite Horizon: decision variable, markov shock	3
2.2.1	Finite Horizon: semi-exogenous state	4
2.2.2	Finite Horizon: experienceasset	5
2.2.3	How interpolation is done	6
2.2.4	Finite Horizon: riskyasset	7
2.2.5	Finite Horizon: standard & riskyasset	8
2.2.6	Finite Horizon: Epstein-Zin preferences	9
3	Agent Distribution	12
3.1	Stationary Agent Distribution in Infinite Horizon	12
3.1.1	Stationary Agent Distribution as Eigenvector	12
3.1.2	Simulate Agent Distribution	13
3.1.3	Iterate Agent Distribution	14
3.1.4	Tan Improvement to Iterating Agent Distribution	14
3.2	Agent Distribution in Finite Horizon	15
3.2.1	Simulating the Agent Distribution	15
3.2.2	Iterating the Agent Distribution	16
3.2.3	Tan Improvement Iterating the Agent Distribution	17
3.2.4	Tan Improvement with Semi-Exogenous shocks	17
4	Stationary General Equilibrium	18
5	Transition Path General Equilibrium	19

1 Introduction

This document presents pseudocodes for various commands from VFI Toolkit. The actual implementations in VFI Toolkit often involve parallelized versions of the pseudocodes presented here, and also typically have options to use more-or-less parallelization (versus looping).

This document is a work-in-progress. Please request pseudocode for any specific command you want to see it for (either by email or on discourse.vfitoolkit.com).

Throughout I use the standard notation of VFI Toolkit: so d is a decision variable, a is endogenous state (and a' the next-period endogenous state), z is a markov exogenous state, e is an i.i.d. exogenous state, u is an i.i.d. shock that occurs between periods. V is the value function. For finite horizon problems there are N_j periods, indexed by j .

In many pseudocodes $g(a, z)$ denotes a policy, with $g^d(a, z)$ being the policy for the decision variables and $g^{a'}(a, z)$ being the policy for the next-period endogenous state.

2 Value Function Iteration: Solving for V and Policy

2.1 Infinite Horizon: decision variable, markov shock

Starting from an initial guess for the value function, we iterate on the value function until convergence. To speed things up Howards improvement is used.

Declare initial value V_0 .

Declare iteration count $n = 0$.

while $||V_n - V_{n-1}|| > \text{'tolerance constant'}$

Increment n . Let $V_{old} = V_{n-1}$.

for All values z

Calculate $E[V_{old}(a', z')]$

for All values of a

Calculate $V_n(a, z) = \max_{d, a' \in D(a, z)} F(d, a', a, z) + \beta E[V_{old}(a', z')]$

... and keep the *argmax* $g_n(a, z)$.

end for

end for

if UseHowards==1

for n Howards number of times

Update $V_n(a, z) = F(g_n^d(a, z), g_n^{a'}(a, z), a, z) + \beta E[V_n(g_n^{a'}(a, z), z')]$

end for

end if

end while

return V_n, g_n

Note that by making the outer-loop over z we are able to reduce the number of times we compute the expectations (which depend on z and not on a).

Howards improvement algorithm is to use the current policy to evaluate/update the value function a couple of times. This is faster as it skips the maximization step (which is computationally the most costly) and because the policy function typically is 'pointing' in the direction of the solution, so a few cheap computations to move further in this direction is worthwhile. *UseHowards* is implemented as conditions under which Howards improvement algorithm should be used (which can be controlled using *vfoptions*. *nHowards* is set to 80 by default (based on a trying out a bunch of different values, this seemed to be best at speeding up the 'average' problem). In practice, we avoid using Howards improvement algorithm for the first few iterations (as the initial guess is likely poor, and so Howards does not much help), and also stop using Howards improvement algorithm once we are within one order of magnitude of convergence of the value function.¹

A couple of hints to how this could be done better (the kind of improvements that work for just about any algorithm, rather than things like using endogenous grid method instead of pure discretization): ideally it would use the improved versions of Howards developed in Rendahl (2022), Bakota and Kredler (2022) and Phelan and Eslami (2022) (I've not tried these). And it would use relative VFI, or even endogenous VFI (for these later two, they were tried but don't work in VFI Toolkit because of the pure discretization); Bray (2017).

2.1.1 Refinement

If you set *vfoptions.solnmethode* = 'purediscretization_refinement'; then the algorithm will be the 'refinement'. Refinement is based on the observation that d does not enter the expectations next period value function, only the return function. So we can 'pre-solve', choosing d to maximize the return function $F(d, a_{prime}, a, z)$ to get $d^*(a_{prime}, a, z)$, the decision that maximizes the return function given (a_{prime}, a, z) , and associated maximum values $F^*(a_{prime}, a, z)$. We then perform standard value function iteration, but just using $F^*(a_{prime}, a, z)$, so we have essentially removed d from the problem on which we have to iterate. Once we finish solving we can put the optimal policy for a_{prime} into $d^*(a_{prime}, a, z)$ to get the optimal policy for d . This is faster because we just pre-solve for d once, and thereby avoid having to solve for it as part of every iteration.

for All values z

▷ First, pre-solve for d

¹This is because otherwise Howards leads to very small errors. You won't notice them normally, but when trying to compute a placebo transition path between stationary equilibrium they are problematic as the transition won't use Howards and so tries to go to a very slightly different place. If you are thinking 'but I saw the proof in Sargent & Stachurski that Howards is fine and gives same answer' the trick is that the last line of their proof assumes that Howards evaluates the 'policy greedy' value function, but in practice you will never do this. (Placebo transition=a transition path in which nothing changes, this is actually quite a demanding test of accuracy; this is not a dig at S&S, all textbooks do the same.)

```

for All values  $a$ 
    Calculate  $F^*(a', a, z) = \max_{d \in D(a, z)} F(d, a', a, z)$  and the  $\text{argmax } d^*(a', a, z)$ 
end for
end for
Declare initial value  $V_0$ . ▷ Now do standard VFI without  $d$  variable
Declare iteration count  $n = 0$ .
while  $\|V_n - V_{n-1}\| > \text{'tolerance constant'}$ 
    Increment  $n$ . Let  $V_{old} = V_{n-1}$ .
    for All values  $z$ 
        Calculate  $E[V_{old}(a', z')]$ 
        for All values of  $a$ 
            Calculate  $V_n(a, z) = \max_{a' \in D(a, z)} F^*(a', a, z) + \beta E[V_{old}(a', z')]$ 
            ... and keep the  $\text{argmax } g_n(a, z)$ .
        end for
    end for
end while
if UseHowards==1
    for n Howards number of times
        Update  $V_n(a, z) = F^*(g_n^{a'}(a, z), a, z) + \beta E[V_n(g_n^{a'}(a, z), z')]$ 
    end for
end if
end while
Evaluate  $g_n^d(a, z) = d^*(g_n^{a'}(a, z), a, z)$  ▷ Recover the policy for  $d$ 
return  $V_n, g_n$ 

```

Note that the only difference between the refinement is internal. All inputs and outputs will be identical. Refinement also has the nice property that if we loop when calculating d^* and then parallelize the value function iteration itself, we reduce the memory requirements as well as reducing the runtime making it possible to solve more complex problems (as long as they have a d variable). Obviously the refinement is not possible in models that do not have a d variable to begin with.²

2.2 Finite Horizon: decision variable, markov shock

Just some simple backward iteration. Let Θ be all the parameters of the model, and let θ_j be the values of those parameters which are relevant in period j .³

Solve for final period, N_j : $V_{N_j}(a, z) = \max_{d, a' \in D(a, z)} F(d, a', a, z)$
 ... and keep the $\text{argmax } g_{N_j}$.

²Pure-discretization makes this refinement easy, but in principle it should be possible to do with algorithms that use functional forms to fit the value and policy functions.

³So if a parameter is independent of age, then it is just in θ_j as is. If a parameter depends on age, then only the value relevant to age j is in θ_j .

```

for  $j$  count backward from  $N_j - 1$  to 1
  Get  $\theta_j$  from  $\Theta$ 
  for All values  $z$ 
    Calculate  $E[V_{j+1}(a', z')]$ 
    for All values of  $(a, z)$ 
      Calculate  $V_j(a, z) = \max_{d, a' \in D(a, z)} F(d, a', a, z) + \beta E[V_{j+1}(a', z')]$ 
      ... and keep the argmax  $g_j(a, z)$ .
    end for
  end for
end for
return  $[V_1, V_2, \dots, V_{N_j}], [g_1, g_2, \dots, g_{N_j}]$ 

```

If you were writing custom code for a specific problem, you would likely try to take advantage of things like monotonicity, but because VFI Toolkit is trying to solve generic problems it just uses this simple robust approach.

2.2.1 Finite Horizon: semi-exogenous state

We now add a semi-exogenous state. Semi-exogenous shocks are exogenous states for which the transition probabilities depend on a decision variable (see Life-Cycle Model 30 for an example). So $\pi_{semiz}(semiz'|semiz, d)$ gives the probability of next-period semi-exogenous state based on the current semi-exogenous state and on the decision variable.

The difference the expectations are now more complex.

```

Solve for final period,  $N_j$ :  $V_{N_j}(a, semiz, z) = \max_{d, a' \in D(a, semiz, z)} F(d, a', a, semiz, z)$ 
... and keep the argmax  $g_{N_j}$ .
for  $j$  count backward from  $N_j - 1$  to 1
  Get  $\theta_j$  from  $\Theta$ 
  for All values  $z$ 
    Calculate  $E[V_{j+1}(a', semiz', z')|d]$     ▷ Note conditional on d, because of semi-exog state
    for All values of  $(a, semiz, z)$ 
      Calculate  $V_j(a, semiz, z) = \max_{d, a' \in D(a, semiz, z)} F(d, a', a, semiz, z) + \beta E[V_{j+1}(a', semiz', z')|d]$ 
      ... and keep the argmax  $g_j(a, semiz, z)$ .
    end for
  end for
end for
return  $[V_1, V_2, \dots, V_{N_j}], [g_1, g_2, \dots, g_{N_j}]$ 

```

The $E[V_{j+1}(a', semiz', z')|d]$ term is doing a lot of work here. It adds a whole dimension to the expectations term. The way this is implemented in the codes is actually slightly different, namely

there is a loop over d and the problem is solved conditional on d , then once the loop is finished the maximum over d is taken. That is the lines 5-11 become

```

for A ll values of  $d$ 
  for A ll values  $z$ 
    Calculate  $E[V_{j+1}(a', semiz', z')|d]$     ▷ Note conditional on  $d$ , because of semi-exog state
    for A ll values of  $(a, semiz, z)$ 
      Calculate  $V_j(a, semiz, z|d) = \max_{a' \in D(a, semiz, z)} F(d, a', a, semiz, z) + \beta E[V_{j+1}(a', semiz', z')|d]$ 
      ... and keep the  $argmax$   $g_j(a, semiz, z|d)$ .
    end for
  end for
end for
 $V_j(a, semiz, z) = \max_d V_j(a, semiz, z|d)$ 
and used the  $argmax$   $d$  from this, denoted  $d^*$  to get  $g_j(a, z) = g_j(a, z|d^*)$ .

```

where implicitly for values of d that are not in $D(a, semiz, z)$, the return is $-\infty$ and so these will never be chosen.

2.2.2 Finite Horizon: experienceasset

'experienceasset' is an endogenous state where a_{prime} cannot be chosen directly, but is instead a function of a decision variable and this period endogenous state; $a_{prime}(d, a)$. The basic idea of how VFI Toolkit handles these is to evaluate $a_{prime}(d_i, a_i)$ as a value on the grids of d and a , and then linearly interpolate this $a_{prime}(d_i, a_i)$ onto the grid for a by allocating it to the grid-points above and below that value with linear weights.

So in a finite-horizon model with no other endogenous states, and no decision variables except those influencing the next period experienceasset, the algorithm is

```

Solve for final period,  $N_j$ :  $V_{N_j}(a, z) = \max_{d \in D(a, z)} F(d, a, z)$ 
... and keep the  $argmax$   $g_{N_j}$ .
for  $j$  count backward from  $N_j - 1$  to 1
  Get  $\theta_j$  from  $\Theta$ 
  for A ll values  $z$ 
    Calculate  $E[V_{j+1}(a', z')]$ 
    Evaluate  $a_{prime}(d, a)$  on the grids for  $d$  and  $a$ 
    Interpolate  $a_{prime}(d, a)$  onto the grid for  $a$ ; get  $a_{prime}_i(d, a)$  and  $aProbs(d, a)$ , the index
    for the lower grid point and the probability of the lower grid point.
    Switch  $E[V_{j+1}(a', z')]$  to  $E[V_{j+1}(d, a, z')]$  using  $a_{prime}_i(d, a)$  and  $aProbs(d, a)$  to replace
     $a'$  with  $d$  and  $a$ .
  for A ll values of  $(a, z)$ 

```

```

    Calculate  $V_j(a, z) = \max_{d \in D(a, z)} F(d, a, z) + \beta E[V_{j+1}(d, a, z')]$ 
    ... and keep the argmax  $g_j(a, z)$ .
  end for
end for
end for
return  $[V_1, V_2, \dots, V_{N_j}], [g_1, g_2, \dots, g_{N_j}]$ 

```

notice that a' no longer appears in the return function. The main trick is to replace $E[V_{j+1}(a', z')]$ to $E[V_{j+1}(d, a, z')]$, by interpolating *aprime*(a, z) onto the grid on a .

The way interpolation is done is that for a value of *aprime* that we want to interpolate onto the grid $[a_1, a_2, \dots, a_n]$ we, (i) find the a_i that is the largest grid point that has a value less than (or equal to) *aprime*, call this the lower grid point, (ii) we allocate *aprime* to both the lower grid point a_i and the upper grid point a_{i+1} using weights, (iii) we use linear weights for a_i and a_{i+1} , so if *aprime* is halfway between the two we give a weight of 0.5 to each, this is implemented by giving the lower grid point, a_i a weight of $1 - (\text{aprime} - a_i) / (a_{i+1} - a_i)$ (the weight for the upper grid point is just 1-this). This interpolation can be seen in [CreateExperienceAssetFnMatrix_Case1.m](#), the first part of the code creates *aprime*(a, z) and then the last 30-40ish lines are doing the interpolation onto the grid, note that we only need to keep the lower grid point index and the probability assigned to the lower grid point (as the upper grid point is just adding one to this index, and the probability is one minus this).

2.2.3 How interpolation is done

VFI Toolkit uses linear interpolation as part of alternative endogenous states like *experienceasset*, *experienceassetu*, *riskyasset*, and more. There are two halves to how the toolkit uses interpolation. One half is to put points onto grid (as a lower grid point index and a probability of the lower point; together implicitly with an upper grid point and probability of the upper point). The second half is to use this to 'switch' grids.

For example in the previous pseudocode we saw the first half as:

Interpolate aprime(d, a) onto the grid for a; get $\text{aprime}_i(d, a)$ and $a\text{Probs}(d, a)$, the index for the lower grid point and the probability of the lower grid point.

The way that the toolkit interpolates a value onto the grid is to uses linear interpolation. So say we have that the value we want to interpolate onto the grid is x , and the grid is $\{x_1, x_2, \dots, x_n\}$. The first step is to find the two grid points just either side of x , call them x_i and x_{i+1} , and these will obviously satisfy $x_i \leq x < x_{i+1}$ [note equality on the lower but not upper point; this is arbitrary, just pick one]. Then we call x_i the lower grid point. We assign this lower grid point the probability $1 - \frac{(x - x_i)}{(x_{i+1} - x_i)}$ (notice that this is a number from 0 to 1, and is 1 if $x = x_i$). We only need to store the (index of the) lower grid point and probability of the lower grid point, as we always know that the

upper grid point is just the point one above this, and that the probability of the upper grid point is one minus the probability of the lower grid point (so probability of upper grid point is $\frac{(x-x_i)}{(x_{i+1}-x_i)}$, which again is 0 to 1, and approaches 1 as x approaches x_{i+1}).

While the second half in the previous pseudocode we saw as:

Switch $E[V_{j+1}(a', z')]$ to $E[V_{j+1}(d, a, z')]$ using $aprime_i(d, a)$ and $aProbs(d, a)$ to replace a' with d and a .

The way the toolkit does this is that for every (d, a) we have a value $aprime(d, a)$ which we associated in the first half with a lower grid point index and probability, $aprime_i(d, a)$ and $aProbs(d, a)$.

Looking at $E[V_{j+1}(a', z')]$ and ignoring the $E[\cdot]$ for the moment, we get

$V_{j+1}(a', z') = aProbs(d, a)V_{j+1}(aprime_i(d, a), z') + (1 - aProbs(d, a))V_{j+1}(aprime_{i+1}(d, a), z') \equiv V_{j+1}(d, a, z')$. We just need to do this before we evaluate the expectations term, and we will thus get $E[V_{j+1}(d, a, z')]$.

2.2.4 Finite Horizon: riskyasset

'riskyasset' is an endogenous state where $aprime$ cannot be chosen directly, but is instead a function of a decision variable and a between-period i.i.d. shock; $aprime(d, u)$. The basic idea of how VFI Toolkit handles these is to evaluate $aprime(d_i, u_i)$ as a value on the grids of d and u , and then linearly interpolate this $aprime(d_i, u_i)$ onto the grid for a by allocating it to the grid-points above and below that value with linear weights.

So in a finite-horizon model with no other endogenous states, and no decision variables except those influencing the next period riskyasset, the algorithm is

Solve for final period, N_j : $V_{N_j}(a, z) = \max_{d \in D(a, z)} F(d, a, z)$

... and keep the argmax g_{N_j} .

for j count backward from $N_j - 1$ to 1

 Get θ_j from Θ

for All values z

 Calculate $E[V_{j+1}(a', z')]$

 Evaluate $aprime(d, u)$ on the grids for d and u

 Interpolate $aprime(d, u)$ onto the grid for a ; get $aprime_i(d, u)$ and $aProbs(d, u)$, the index for the lower grid point and the probability of the lower grid point.

 Switch $E[V_{j+1}(a', z')]$ to $E[V_{j+1}(d, u, z')]$ using $aprime_i(d, u)$ and $aProbs(d, u)$ to replace a' with d and u .

 Take expectations over u to convert $E[V_{j+1}(d, u, z')]$ to $E[V_{j+1}(d, z')]$

for All values of (a, z)

 Calculate $V_j(a, z) = \max_{d \in D(a, z)} F(d, a, z) + \beta E[V_{j+1}(d, z')]$

 ... and keep the argmax $g_j(a, z)$.

end for

```

    end for
  end for
  return  $[V_1, V_2, \dots, V_{N_j}], [g_1, g_2, \dots, g_{N_j}]$ 

```

notice that a' no longer appears in the return function. The main trick is to replace $E[V_{j+1}(a', z')]$ to $E[V_{j+1}(d, z')]$, by interpolating $a_{prime}(d, u)$ onto the grid on a , and then taking expectations over u .

Note: An important improvement to runtimes in models with `riskyasset` is the use of 'refinement'. Essentially in many models some decision variables will appear in the `ReturnFn` but not the `aprimeFn`, and vice-versa. We can 'refine' out these decision variables before we combine the return function, $F(d, a, z)$ and the expectations term, $E[V_{j+1}]$. This massively shrinks the size of the combined problem and leads to both faster runtimes and lower memory use. It only makes sense to do on a gpu. Refine is explained for infinite horizon problems in Section 2.1.1. The linear interpolation is explained in Section 2.2.3.

2.2.5 Finite Horizon: standard & riskyasset

When using a standard endogenous state and a `riskyasset` together, the risky asset is always set up as the second endogenous state so we will call them $a1$ and $a2$, respectively. This approach is just to combine the usual approaches to each endogenous state, so the standard endogenous state will have $a1_{prime}$ chosen on the grid while the `riskyasset` will be evaluated on the grids of d and u to get $a2_{prime}(d_i, u_i)$ and then this is linearly interpolated onto the $a2$ grid by allocating it to the grid-points above and below that value with linear weights.

So in a finite-horizon model with no decision variables except those influencing the next period `riskyasset`, the algorithm is

```

Solve for final period,  $N_j$ :  $V_{N_j}(a1, a2, z) = \max_{a1, d \in D(a1, a2, z)} F(d, a1_{prime}, a1, a2, z)$ 
... and keep the argmax  $g_{N_j}$ .
for  $j$  count backward from  $N_j - 1$  to 1
  Get  $\theta_j$  from  $\Theta$ 
  for A ll values  $z$ 
    Calculate  $E[V_{j+1}(a1', a2', z')]$ 
    Evaluate  $a2_{prime}(d, u)$  on the grids for  $d$  and  $u$ 
    Interpolate  $a2_{prime}(d, u)$  onto the grid for  $a2$ ; get  $a2_{prime}_i(d, u)$  and  $a2Probs(d, u)$ , the
    index for the lower grid point and the probability of the lower grid point.
    Switch  $E[V_{j+1}(a1', a2', z')]$  to  $E[V_{j+1}(d, a1', u, z')]$  using  $a2_{prime}_i(d, u)$  and  $a2Probs(d, u)$ 
    to replace  $a2'$  with  $d$  and  $u$ .
    Take expectations over  $u$  to convert  $E[V_{j+1}(d, a1', u, z')]$  to  $E[V_{j+1}(d, a1', z')]$ 
  for A ll values of  $(a1, a2, z)$ 

```

```

    Calculate  $V_j(a1, a2, z) = \max_{d, a1' \in D(a1', a2', z)} F(d, a1', a1, a2, z) + \beta E[V_{j+1}(d, a1', z')]$ 
    ... and keep the argmax  $g_j(a1, a2, z)$ .
  end for
end for
end for
return  $[V_1, V_2, \dots, V_{N_j}], [g_1, g_2, \dots, g_{N_j}]$ 

```

notice that $a2'$ does not appear in the return function. The main trick is to replace $E[V_{j+1}(a1', a2', z')]$ with $E[V_{j+1}(d, a1', z')]$, by interpolating $a2prime(d, u)$ onto the grid on $a2$, and then taking expectations over u .

2.2.6 Finite Horizon: Epstein-Zin preferences

There are various different ways to set up Epstein-Zin preferences, such as in consumption-units, in utility-units with positive utility, in utility-units with negative utility, and 'double Epstein-Zin' which treats the additional risk aversion differently for survival probabilities to other probabilities. VFI Toolkit handles all of these in a single code by taking advantage of the fact that we can set up a generic Epstein-Zin with eight 'Epstein-Zin coefficients' (called *ezc1*, *ezc2*, ..., *ezc8* in the codes), and then all the different Epstein-Zin preferences simply correspond to different values of these eight coefficients.⁴

The generic Epstein-Zin value fn problem for a finite-horizon value fn problem with markov shocks is,

$$V_j(a, z) = \max_{d, a'} ezc1 * ((ezc1 * F(d, a', a, z)^{ezc2}) + ezc3 * \beta * (s_j E[ezc4 * V_{j+1}(a', z')^{ezc5} | z]^{ezc8})^{ezc6})^{ezc7}$$

The algorithm for solving this is,

Solve for final period, N_j : $V_{N_j}(a, z) = \max_{d, a' \in D(a, z)} (ezc1 * F(d, a', a, z)^{ezc2})^{ezc7}$

... and keep the *argmax* g_{N_j} .

for j count backward from $N_j - 1$ to 1

Get θ_j from Θ

for All values z

Calculate $E[ezc4 * V_{j+1}(a', z')^{ezc5} | z]^{ezc8}$

for All values of (a, z)

Calculate $V_j(a, z) = \max_{d, a' \in D(a, z)} ezc1 * ((ezc1 * F(d, a', a, z)^{ezc2}) + ezc3 * \beta (s_j E[ezc4 * V_{j+1}(a', z')^{ezc5} | z]^{ezc8})^{ezc6})^{ezc7}$

... and keep the *argmax* $g_j(a, z)$.

end for

end for

⁴This approach was developed from scratch for VFI Toolkit.

```

end for
return  $[V_1, V_2, \dots, V_{N_j}], [g_1, g_2, \dots, g_{N_j}]$ 

```

Implementing the code for this requires a bit of care because there are issues around things like 0s and negative powers where you have to be careful to treat it correctly.

We now go through different Epstein-Zin setups and explain how the eight Epstein-Zin coefficients are set in each case. We use *ceis* and *crisk* to refer to the elasticity of intertemporal substitution and risk aversion parameters, respectively.

Traditional EZ in consumption units

```

           $ezc1 = 1;$ 
       $ezc2 = 1 - 1./ceis;$ 
           $ezc3 = 1;$ 
           $ezc4 = 1;$ 
       $ezc5 = 1 - risk;$ 
   $ezc6 = (1 - 1./ceis)./(1 - risk);$ 
       $ezc7 = 1./(1 - 1./ceis);$ 
           $ezc8 = 1;$ 

```

EZ in utility-units (in utils): positive utility

```

           $ezc1 = 1;$ 
           $ezc2 = 1;$ 
           $ezc3 = 1;$ 
           $ezc4 = 1;$ 
       $ezc5 = 1 - risk;$ 
   $ezc6 = 1./(1 - risk);$ 
           $ezc7 = 1;$ 
           $ezc8 = 1;$ 

```

EZ in utility-units (in utils): negative utility

$$\begin{aligned}
ezc1 &= 1; \\
ezc2 &= 1; \\
ezc3 &= -1; \\
ezc4 &= -1; \\
ezc5 &= 1 + crisk; \\
ezc6 &= 1./(1 + crisk); \\
ezc7 &= 1; \\
ezc8 &= 1;
\end{aligned}$$

Note: If the utility is negative use 1+crisk instead of 1-crisk. This way the interpretation of risk is identical in both cases. And then we need to multiply it by -1 in two places, which is done by setting *ezc3* and *ezc4* to both be negative.

Double Epstein-Zin: inner EZ is about risk, out EZ is about mortality risk

$$\begin{aligned}
&\text{EZ in consumption units:} \\
ezc6 &= (1 - 1./ceis)./(1 - mrisk); \\
ezc8 &= (1 - mrisk)./(1 - crisk); \\
&\text{EZ in utils with positive utility:} \\
ezc6 &= 1./(1 - mrisk); \\
ezc8 &= (1 - mrisk)./(1 - crisk); \\
&\text{EZ in utils with negative utility:} \\
ezc6 &= 1./(1 + mrisk); \\
ezc8 &= (1 + mrisk)./(1 + crisk);
\end{aligned}$$

Note: this must be used together with one of the other settings, hence just showing which parameters need to change. *mrisk* is the risk aversion parameter for the mortality risk.

Scale returns by $1 - \beta$ or $1 - s_j\beta$: Some people like to scale the return function by $1 - \beta$ or $1 - s_j\beta$ (done with `voptions.EZoneminusbeta`). This just requires setting *ezc1* = $1 - \beta$ or *ezc1* = $1 - s_j\beta$.

There is one aspect of Epstein-Zin preferences that cannot be accomodated by this approach, namely when the elasticity of intertemporal substitution is equal to one. As can clearly be seen in the definitions of the Epstein-Zin coefficients, this would lead to some of them being undefined due to division by zero, or blowing off to infinity. It is possible to solve Epstein-Zin preferences with the elasticity of intertemporal substitution is equal to one, but it cannot be fit into our generic Epstein-Zin problem. For how the problem has to be set up, see Sargent & Lungquist 4ed (2018)

on pages 566-568, specifically looking at eqn (14.7.3) and compare the two cases. In practice, you could use the toolkit to solve with an elasticity of intertemporal substitution equal to 0.99 and again with 1.01, and as long as they give much the same solution, just interpret this as being a good enough approximation to the solution with 1.⁵

3 Agent Distribution

3.1 Stationary Agent Distribution in Infinite Horizon

There are three ways to compute the stationary agent distribution in infinite horizon models. The stationary agent definition is defined as $\mu(a, z)$ satisfying $\mu(a', z') = P(a', z'|a, z)\mu(a, z)$, where P is the markov transition kernel that combines the policy function $g(a'|a, z)$ with the exogenous shock transition kernel $\pi_z(z'|z)$ ($P = g \circ \pi_z$).

The three ways to compute the stationary agent distribution are (i) as the eigenvector of $\mu = P\mu$, (ii) by simulation, and (iii) by iteration.

Which of the three to use? The iteration method is the default as it has a better combination of speed and accuracy than the simulation method. However the iteration method does require more memory and so for large models it is not an option; if you are getting out of memory errors when solving for the agent distribution then turn off iteration by setting `simoptions.iterate=0` and the toolkit will instead use the simulation. The eigenvector method is unstable so is never used.

3.1.1 Stationary Agent Distribution as Eigenvector

In principle this is simple, we know the stationary distribution is defined as satisfying $\mu = P\mu$, and once the problem is discretized P is a matrix. So μ is just an eigenvector of the matrix P and there are plenty of routines existing for finding eigenvectors in all standard software and they are very fast. In practice however the method is unstable as minor numerical rounding errors in computing the eigenvector mean it often contains negative elements. There are ways to 'bend' the eigenvector so that it is a probability distribution (all elements are positive, sum to one) as given, e.g., 'Step 5' on page 178 of Badshah, Beaumont, and Srivastava (2013) describes how to find the normalized eigenvector of the unit eigenvalue of P and use the Perron-Frobenius theorem to 'bend' it.

By default the toolkit will avoid the eigenvector approach due to the instability (if you want to use the eigenvector method, set `simoptions.eigenvector=1`. Neither of the ways to 'bend' it have been implemented (if you want to contribute one, please do :)

⁵Thanks to Louis Daumas for pointing me to where to find this.

3.1.2 Simulate Agent Distribution

We have a markov process P (on the joint space of the endogenous and exogenous states). We know from econometric theory that if we simulate a single time series using P then asymptotically the distribution of this time series will convergence to what we are calling the stationary distribution $\mu = P\mu$.

Two minor points are needed to implement this. First, we have to start from somewhere, and we don't one the point we start from to matter, so we do 'burnin': simulate for B periods, throw these away except for where we end up, and now use this as the starting point for simulating our distribution (because the starting point is random it will not matter on average; it is also likely to be a 'typical' point). Second, simulating a single time series is slow, so in practice we simulate lots of time series in parallel on the cpu; while this is not in line with the underlying theory justifying the approach it works just fine in practice.

The pseudocode for simulating the agent distribution follows. Note everything is done based on the index of (a, z) , not the value.

We will simulate M observations. Done as N simulations of length T ($M = N \times T$)

```

for  $i$  count from 1 to  $N$  (Parallel-for over cpu cores)
  Choose starting point  $(a_0, z_0)$ 
  Create  $Dist_i = \text{zeros}(\text{size of } (a, z) \text{ space})$ 
  for  $t$  counts from 1 to  $B + 1$  ▷  $B$  is number of periods for burnin
    Draw new  $(a_t, z_t)$  from  $P(a', z' | a_{t-1}, z_{t-1})$ 
  end for
  for  $t$  counts from  $B + 1$  to  $T$  ▷ Same as during burnin, but now we will keep them
    Draw new  $(a_t, z_t)$  from  $P(a', z' | a_{t-1}, z_{t-1})$ 
     $Dist_i(a_t, z_t) = Dist_i(a_t, z_t) + 1$  (recall, using indexes, not values)
  end for
   $Dist(:, i) = Dist_i$  ▷ Put the  $Dist_i$  together
end for
 $Dist = \text{sum}(Dist, \text{in the } i \text{ dimension})$  ▷  $Dist$  is now a count of how many times each point in
distribution was visited during the simulations
 $Dist = Dist / \text{sum}(Dist)$  ▷ Normalize to be a probability distribution
return  $Dist$ 

```

the implementation in VFI Toolkit uses the mid-point of the grids on (a, z) as the initial starting point (a_0, z_0) for all simulations (it will anyway be burned away). This can be controlled setting `simoptions.seedpoint` (which has default value of the mid-point of the grids on (a, z)).

3.1.3 Iterate Agent Distribution

If we start from some initial guess for agent distribution μ_0 , and iterate $\mu_t = P\mu_{t-1}$, we know from the theory that this sequence of μ_t will converge to the stationary distribution μ .⁶

Iteration just repeatedly iterates until convergence, so psuedocode is just

Start from initial guess μ_{old}

while $currdist > \text{tolerance}$

$\mu = P\mu_{old}$

▷ Iterate agent distribution

$currdist = \max(\text{abs}(\mu - \mu_{old}))$

$\mu_{old} = \mu$

▷ Update 'old'

end while

return μ

▷ The distribution once convergence was reached

This just leaves the question of what to use as the initial guess μ_{old} . By default VFI Toolkit just put equal weight on every grid point as the initial guess for μ_{old} . You can give an initial guess for μ_{old} as *simoptions.initialdist* (if guess is good, codes will run faster).⁷

The code implementing this contains one trick to speed it up, namely that it computes $\mu = P\mu_{old}$ multiple times before calculating *currdist*; because the distance calculation is non-trivial it is costly to compute it every step. This is controlled by *simoptions.multititer* which is set to 50 by default.

To avoid the risk of just iterating forever if the agent distribution is failing to converge, the codes impose a maximum number of iterations (it is an additional criterion of the while loop). This is controlled by *simoptions.maxit* and is set to 50,000 by default.

You can control 'tolerance' by setting *simoptions.tolerance*.

3.1.4 Tan Improvement to Iterating Agent Distribution

By default VFI Toolkit will perform the 'Tan improvement' Tan (2020) to iteration which we now describe. In standard iteration we use the transition matrix P , which is the composition of the policy function $g(a'|a, z)$ and the transition matrix for the exogenous state $\pi_z(z'|z)$; $P = g \circ \pi_z$. The Tan improvement is the clever observation that actually instead of doing the iteration as $\mu = P\mu_{old}$, we can actually do it as a two-step process, with the first step being $\tilde{\mu} = \Gamma\mu_{old}$ and the second step is $\mu = \pi_z\tilde{\mu}$; where Γ is just g but as a transition from (a, z) to (a', z) rather than just to a (notice that both P and Γ go from (a, z) , but P goes to (a', z') while Γ goes to (a', z)). Conceptually this

⁶Typically P is either a 'stochastic contraction' or, more likely for incomplete markets models, satisfies a 'monotone mixing condition' (or 'splitting condition', or 'monotone order-mixing condition') and either of these means the sequence will converge. Actually analytically proving either of these is non-trivial but has been done for some of the more basic/standard heterogenous agent incomplete market models.

⁷The toolkit used to use a small simulation as the initial guess, but the Tan improvement made the iteration so fast that getting a good initial guess was no longer worth the run time involved.

seems pointless, but computationally Γ is very *sparse* and so the Tan improvement is both vastly faster and uses less memory than standard iteration.

The pseudocode for iteration with the Tan improvement is thus,

Start from initial guess μ_{old}

Compute Γ from g

while $currdist > tolerance$

$\mu = \Gamma \mu_{old}$

▷ First step of Tan improvement

$\mu = \pi_z \mu$

▷ Second step of Tan improvement

$currdist = \max(abs(\mu - \mu_{old}))$

$\mu_{old} = \mu$

▷ Update 'old'

end while

return μ

▷ The distribution once convergence was reached

There is some reshaping of matrices involved in the Tan improvement that I have glossed over here, for a full explanation see Tan (2020). As in standard iteration our implementation does not check the *currdist* every iteration as it is faster not to.

Because the Tan improvement makes the iteration step so much faster it is no longer worth generating a decent initial guess for μ_{old} , and so the default just puts all the mass for the endogenous state on the mid-point of the grid, and for the exogenous state it iterates ten times with π_z from putting equal mass on all grid points. You can give an initial guess for μ_{old} as *simoptions.initialdist* (if guess is good, codes will run faster).

3.2 Agent Distribution in Finite Horizon

There are two ways to compute the agent distribution in finite horizon, simulation and iteration. The user must define/input the period-1 distribution. The iteration method is used by default as it has a better combination of speed and accuracy than simulation, but because the iteration requires more memory it will sometimes not be possible and simulation should then be used instead (default is *vfoptions.iterate=1*, setting it to 0 means simulation will be used instead). When we use simulation we are essentially just doing a panel data simulation, but then converting the result into an agent distribution.

3.2.1 Simulating the Agent Distribution

The pseudocode for simulating the agent distribution follows. Note everything is done based on the index of (a, z) , not the value. We use μ_j to denote the distribution at age/period j , and g denotes policy function, π_z denotes the transition matrix for the exogenous state.

Define initial age-1 distribution $\mu_1(a, z)$

Define age-weights $\omega(j)$

We will perform N simulations of length J

for i count from 1 to N (Parallel-for over cpu cores)

 Draw starting point (a_1, z_1) from $\mu_1(a, z)$

 Create $Dist_i = \text{zeros}(\text{size of } (a, z, j) \text{ space})$

for j counts from 1 to J

 Get $a_j g(a_{j-1}, z_{j-1}, j - 1)$

 Draw z_j from $\pi_z(z_j | z_{j-1}; j)$

end for

$Dist_i(a_j, z_j, j) = Dist_i(a_j, z_j, j) + 1$ (recall, using indexes, not values)

end for

$Dist(:, :, i) = Dist_i$ ▷ Put the $Dist_i$ together

$Dist = \text{sum}(Dist, \text{in the } i \text{ dimension})$ ▷ $Dist$ is now a count of how many times each point in distribution was visited during the simulations

$Dist(:, :, j) = Dist(:, :, j) / \text{sum}(Dist(:, :, j))$, for each j ▷ Normalize conditional on age to be a probability distribution

Multiply ω along the j dimesion of $Dist$.

return $Dist$

3.2.2 Iterating the Agent Distribution

To iterate the agent distribution in finite horizon we need an intial distribution defined by the user, μ_1 , and then construct the transition matrix on (a, z) for each period j denoted P_j . P_j is constructed by combinging the policy function for period j , g_j , with the transition matrix on exogenous shocks for period j , $\pi_{z,j}$.

We just iterate j times,

Define initial age-1 distribution $\mu_1(a, z)$ (must be mass 1)

Define age-weights $\omega(j)$

for j counts from 1 to $J - 1$

 Construct P_j from g_j and $\pi_{z,j}$

$\mu_{j+1} = P_j \mu_j$ ▷ Iterate agent distribution

end for

Store all the μ_j in μ

Multiply ω along the j dimesion of μ .

return μ

In the actual codes the storing of μ_j in μ is done inside the for-loop, and we just have a μ_{new} and μ_{old} rather than all of the μ_j .

VFI Toolkit by default will actually use the Tan improvement when iterating on the agent distribution and we turn to that now. We note that P here takes us directly from (a, z) to (a', z') .

3.2.3 Tan Improvement Iterating the Agent Distribution

The Tan improvement is the observation that instead of using P we can use g and π_z one-by-one. This bit of genius means we do exactly the same, but in a way that is faster and less memory demanding (people spent 30+ years using P before Tan saw that you could split it in two!).

We just iterate j times,

Define initial age-1 distribution $\mu_1(a, z)$ (must be mass 1)

Define age-weights $\omega(j)$

for j counts from 1 to $J - 1$

Construct Γ_j from g_j

$\mu_{temp} = \Gamma_j \mu_j$

▷ First step of Tan improvement

$\mu_{j+1} = \pi_{z,j} \mu_{temp}$

▷ Second step of Tan improvement

end for

Store all the μ_j in μ

Multiply ω along the j dimesion of μ .

return μ

There is some reshaping of matrices involved in the Tan improvement that I have glossed over here, for a full explanation see Tan (2020).

The Tan improvement can be understood as breaking the move from (a, z) to (a', z') into two steps. The first step is from (a, z) to (a', z) , and the second step is from (a', z) to (a', z') .

3.2.4 Tan Improvement with Semi-Exogenous shocks

Semi-exogenous shocks are exogenous states for which the transition probabilities depend on a decision variable (see Life-Cycle Model 30 for an example). So $\pi_{semiz}(semiz'|semiz, d)$ gives the probability of next-period semi-exogenous state based on the current semi-exogenous state and on the decision variable.

Using the Tan improvement requires a minor modification. Note that we cannot treat $semiz$ like we treat z , in the second step of the Tan improvement, because $semiz$ depends on d (and hence on a and z). Instead we have to include $semiz$ in the first step of the Tan improvement.

We just iterate j times,

Define initial age-1 distribution $\mu_1(a, semiz, z)$ (must be mass 1)

Define age-weights $\omega(j)$

```

for  $j$  counts from 1 to  $J - 1$ 
    Construct  $\Gamma_j$  from  $g_j$  and  $\pi_{semiz,j}$ 
     $\mu_{temp} = \Gamma_j \mu_j$  ▷ First step of Tan improvement
     $\mu_{j+1} = \pi_{z,j} \mu_{temp}$  ▷ Second step of Tan improvement
end for
Store all the  $\mu_j$  in  $\mu$ 
Multiply  $\omega$  along the  $j$  dimesion of  $\mu$ .
return  $\mu$ 

```

The difference from the standard Tan improvement is all in the first step, which now includes the semi-exogenous state transitions alongside the standard endogenous state transitions. There is some reshaping of matrices involved in the Tan improvement that I have glossed over here, for a full explanation see Tan (2020). Note that now our distribution is on $(a, semiz, z)$, and Γ_j is used in the first step to change to $(a', semiz', z)$, and then the second step goes to $(a', semiz', z')$.⁸

4 Stationary General Equilibrium

To solve for a stationary general equilibrium, we are looking for some parameters that are determined in general equilibrium, call them θ such that our general equilibrium equations equal zero, $GEeqn(\theta) = 0$.

There are various algorithms for doing this (*heteroagentoptions.fminalgo* determines which is used), but all of them essentially boil down to the following,

```

Guess some initial  $\theta^0$ 
Define  $GeneralEqmConditions = \infty$ 
while  $GeneralEqmConditions > tolerance$ 
    Solve the value fn and policy (given current  $\theta^k$ )
    Solve the agent distribution (given current  $\theta^k$  and policy we just found)
    Solve the aggregate variables at each  $t$  (given current  $\theta^k$  and policy/agent dist we just found)
    Evaluate  $GeneralEqmConditions = GEeqn(AggVars, \theta^k)$  (given current  $\theta^k$  and agg vars)
    Update  $\theta^{k+1}$  (in some way, likely based on  $\theta^k$  and  $GeneralEqmConditions$ )
end while
return  $\theta$ 

```

The difference between the different *fminalgo* is how we update the θ . Note that in almost all of the *fminalgo* the while loop is being done by an optimization algorithm (like Matlab's *fminsearch()*).

Whether the problem is finite or infinite-horizon value functions, and whether or not there are

⁸Getting the Tan improvement to work here does require the order $(a, semiz, z)$, in fact the whole toolkit got rewritten just to be able to do this as it originally had z then $semiz$.

permanent types of different kinds of shocks is unimportant, in the sense that the above steps are all the same steps, just that the details of their implementation is changed.

When there are multiple general equilibrium equations, *GeneralEqmConditions* is (by default) defined as the sum-of-squares of the individual general equilibrium equations.

5 Transition Path General Equilibrium

We consider a transition path of length T time periods. Solving for a transition path general equilibrium, we are looking for some path-of-parameters that are determined in general equilibrium, call them Θ such that our general equilibrium equations equal zero in every time period, $GEeqn(\Theta) = 0$. Note that at some level this is the same thing we did for a stationary equilibrium, except that now $\Theta = \{\theta_1, \dots, \theta_T\}$ is a time-path of general-eqm parameters, rather than just parameters at a single point in time.

There are various algorithms for doing this (*heteroagentoptions.fminalgo* determines which is used), but all of them essentially boil down to the following,

```

Guess some initial  $\Theta^0$  (time-path on general-equilibrium parameters)
Define GeneralEqmConditions =  $\infty$ 
while GeneralEqmConditions > tolerance
    Solve backward,  $t = T - 1, \dots, 2, 1$  for the value fn and policy at each  $t$  (given current  $\theta_t^k$ )
    Solve forward,  $t = 1, 2, \dots, T$  for the agent distribution (given current  $\theta_t^k$  and Policy $t$  we just found)
    Solve the aggregate variables (given current  $\theta_t^k$  and policy/agent dist we just found for  $t$ )
    Evaluate GeneralEqmConditions = [GEeqn1(AggVars1,  $\theta_1^k$ ), GEeqn1(AggVars1,  $\theta_1^k$ ),  $\dots$ , GEeqn $T$ (AggVars $T$ ,  $\theta_T^k$ )] (evaluate general eqm eqns at each time period)
    Update  $\Theta^{k+1}$  (in some way, likely based on  $\Theta^k$  and GeneralEqmConditions)
end while
return  $\theta$ 

```

The difference between the different *fminalgo* is how we update the Θ . The most common way to update is as a shooting algorithm.

In a stationary equilibrium, parameters other than those parameters which were being determined in general equilibrium (the general equilibrium parameters) were all constant (in time) and kept in *Parameters* structure. When we solve transition paths, some parameters might vary over time, and this is done with the *ParamPath* (a path on exogenous parameters), while any other parameters not in this path are assumed to be constant at their values in *Parameters* structure. It is not described in algorithm above, but obviously we have to make sure to use the correct parameters in each time period.

It is important to solve backward for policy and forward for agent distribution. For convenience, aggregate variables and general eqm conditions are often solved forwards while doing agent distribution, but in principle how we deal with these (in terms of timing) is not important in most models (can be important if we use lagged/leading parameter values for something).

Whether the problem is finite or infinite-horizon value functions, and whether or not there are permanent types of different kinds of shocks is unimportant, in the sense that the above steps are all the same steps, just that the details of their implementation is changed.

References

- Muffasir Badshah, Paul Beaumont, and Anuj Srivastava. Computing equilibrium wealth distributions in models with heterogeneous-agents, incomplete markets and idiosyncratic risk. Computational Economics, 41, 2013. doi: <https://doi.org/10.1007/s10614-011-9313-8>.
- Ivo Bakota and Matthias Kredler. Continuous-time speed for discrete-time models: A markov-chain approximation method. MEA Discussion Papers, 2022. doi: <https://dx.doi.org/10.2139/ssrn.4155499>.
- Robert Bray. Markov decision processes with exogenous variables. Management Science, 2017. doi: <https://doi.org/10.1287/mnsc.2018.3158>.
- Robert Kirkby. VFI toolkit, v2. Zenodo, 2022. doi: <https://doi.org/10.5281/zenodo.8136790>.
- Thomas Phelan and Keyvan Eslami. Applications of markov chain approximation methods to optimal control problems in economics. Journal of Economic Dynamics and Control, 2022. doi: <https://doi.org/10.1016/j.jedc.2022.104437>.
- Pontus Rendahl. Continuous vs. discrete time: Some computational insights. Journal of Economic Dynamics and Control, 2022. doi: <https://doi.org/10.1016/j.jedc.2022.104522>.
- Eugene Tan. A fast and low computational memory algorithm for non-stochastic simulations in heterogeneous agent models. Economics Letters, 193, 2020. doi: <https://doi.org/10.1016/j.econlet.2020.109285>.