VFI Toolkit: Pseudocodes

Robert Kirkby*

September 12, 2023

*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

This document presents pseudocodes for various commands from VFI Toolkit. The actual implementations in VFI Toolkit often involve parallelized versions of the psuedocodes 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 psuedocodes $g(a,z)$ 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 $||V_n - V_{n-1}|| >$ 'tolerance constant'
  Increment n. Let $V_{old} = V_{n-1}$.
  for A ll values z
    Calculate $E[V_{old}(a', z')]$
    for A ll 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 U seHowards==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.solnmethod = '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 $F(d, aprime, a, z)$ to get $d^*(aprime, a, z)$, the decision that maximizes the return function given $(aprime, a, z)$, and associated maximum values $F^*(aprime, a, z)$. We then perform standard value function iteration, but just using $F^*(aprime, a, z)$, 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 $aprime$ into $d^*(aprime, a, z)$ 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 A ll values z ▷ First, pre-solve for d

for A ll values a

[1] This is because otherwise Howards leads to very small errors. You won't notice them normally, but when trying to compute a 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.

Calculate $d^*(aprime, a, z) = max_{d \in D(a,z)} F(d, a', a, z)$.
end for
end for
Declare initial value V_0. ▷ Now do standard VFI without d variable
Declare iteration count $n = 0$.
while $||V_n - V_{n-1}||$ > 'tolerance constant'
Increment n. Let $V_{old} = V_{n-1}$.
for A ll values z
Calculate $E[V_{old}(a', z')]$
for A ll 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 $argmax$ $g_n(a, z)$.
end for
end for
if U seHowards==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 $g_n^d(a, z) = d(g^{a'}(a, z), a, z)$ ▷ 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 parallize 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 argmax g_{N_j}.
for j count backward from $N_j - 1$ to 1

[2]Pure-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.

[3]So 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.

Get θ_j from Θ
for A ll values z
Calculate $E[V_{j+1}(a', z')]$
for A ll values of a
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 $[V_1, V_2, ..., V_{N_j}]$, $[g_1, g_2, ..., g_{N_j}]$

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.

3 Agent Distribution

3.1 Stationary Agent Distribution in Infinite Horizon

There are three was 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 larger 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 eigenvecter 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 computational 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 :)

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 $Dist_i = zeros(size\ of\ (a,z)\ space)$
  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
$Dist = sum(Dist$,in the i dimension) ▷ $Dist$ is now a count of how many times each point in distribution was visited during the simulations

$Dist = Dist/sum(Dist)$ ▷ 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.[4]

Iteration just repeatedly iterates until convergence, so psuedocode is just

Start from initial guess μ_{old}
while currdist>tolerance
$\mu = P\mu_{old}$ ▷ Iterate agent distribution
$currdist = max(abs(\mu - \mu_{old}))$
$\mu_{old} = \mu$ ▷ 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 will do a small simulation of the agent distribution and use this 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).

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.multiiter* 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*.

[4]Typically 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.

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 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).

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.