Grid Interpolation Layer

Robert Kirkby*

January 17, 2026

Abstract

Interpolation is a powerful technique for improving the accuracy-runtime (and accuracy-memory) trade-off when solving value function iteration problems. But standard interpolation routines are highly serial and thus not appropriate for the gpu. This brief pdf describes the 'grid interpolation layer' for the GPU that is implemented in VFI Toolkit.

Keywords: Value Function Iteration, Grid Interpolation Layer, VFI Toolkit.

*Thanks to AAA. Thanks to Rāpoi at Victoria University of Wellington for the use of their computing facilities. Kirkby: Victoria University of Wellington. Please address all correspondence about this article to Robert Kirkby at <robertdkirkby@gmail.com>, robertdkirkby.com.

This document is going to largely assume you are already familiar with the concept of interpolation. The basic idea when used in value function iteration is that we have a grid on assets (the endogenous state) and we know the next period expected value function on that same grid for next period assets. If we just restrict our choice of next period assets to fall on the grid, we get the pure discretization that is the basic algorithm of VFI Toolkit. To make the solution more accurate (without changing the grid we are using) we would like to choose values of next period assets without restricting them to be on the grid. This requires evaluating two things at potential values of next period assets (so we can find the optimal value for next period assets), the return function and the expected next period value function. Evaluating the return function off the grid is trivial, it is a known function and so we can just evaluate this function. Evaluating the next period expected value function is trickier, we only know the value function at the points corresponding to the grid on next period assets, so for values not on this grid we must perform interpolation.

So we will use interpolation to evaluate the expected next period value function in terms of next period assets at points between the grid on next period assets. The question then is how to pick which values for next period assets to try, so as to find the optimal value for next period assets (given a value for todays assets). Standard CPU implementations involve an initial guess for next period assets, call it x^0, and then update this guess repeatedly as x^1, x^2, x^3, until convering to an optimal $x*$; this might be done using bisection search, golden search, Newton-Raphson, or many other optimization algorithms. This would be a bad idea on the GPU, as x^0, x^1,..., x^m are highly serial (we have to finish calculating x^m before we can calculate x^{m+1}) with any of these approaches.

Before proceeding to introduce an approach to interpolation that is highly parallel and hence GPU friendly, let's do a bit of notation to set things up. Let our grid on assets be $\{a_1, a_2, \dots, a_n\}$, and use $i = 1, \dots, n$ as to index these grid points. We will use this same grid on next period assets, but refer to it as $\{aprime_1, aprime_2, \dots, aprime_n\}$.[1] The grid is assumed to be stictly increasing, so $a_i < a_{i+1}$ for all $i = 1, \dots, n-1$.

For interpolation to work on the GPU, we need to use a more parallel/less serial approach. The idea implemented as 'grid interpolation layer' in VFI Toolkit is that we first solve the standard pure discretization problem as the 'first layer'; we already know $E[V(aprime_i)]$ for all the points on the next period assets grid. For a given value of todays assets, this will give us an optimal on-the-grid point for next period assets, call it $aprime_i$. We will then perform a 'second layer' by creating a finer grid around $aprime_i$; based on an assumption of monotonicity we know from the fact that $aprime_i$ was on-the-grid-optimal that the true optimal value will lie in the range $(aprime_{i-1}, aprime_{i+1})$. So our second layer will consider the points $aprime_{i-1}$, $a_{i-1,1}$, $a_{i-1,2}$,...,$a_{i-1,n2}$, $aprime_i$, $a_{i-1,1}$, $a_{i-1,2}$,...,$a_{i-1,n2}$, $aprime_{i+1}$; note that this is the points $aprime_{i-1}$ and $aprime_i$ with $n2$ points between them, and then the points $aprime_i$ (not duplicated) and $aprime_{i+1}$ with $n2$ points between them, for a total of $2 * n2 + 3$ points in the 'second layer' grid on next period assets. We can

[1] In theory we could have these being two distinct grids, and they could have different numbers of grid points. Here we assume they are the same grid as it simplifies the exposition, and as that is what the code does.

simply evaluate the return function at each of these points (by function evaluation) and use linear interpolation to evaluate the expected next period value function at each of these points, we can then find the optimal next period assets from among these 'second layer' grid points. To keep things simple, we will assume that the $n2$ second layer points are always evenly spaced beween the first layer grid points.[2]

So to implement this, we solve the 'first layer' considering only values of *aprime* on the grid. Then in the second layer we consider only values of *aprime* that are on the second layer grid, which puts $n2$ points between the first layer grid points above and below the first layer grid point that was found to be optimal in the first layer. Note that the effect of all of this is that we are implicitly considering the 'fine' grid on aprime that is given by the Matlab code $aprime\_grid = interp1(1 : 1 : n, a\_grid, linspace(1, n, n + n2 * (n - 1)))$, which contains $n2 * (n - 1) + n$ points (the n points on the first layer grid, plus $n2$ points between each of them). But by doing it in two layers we only actually end up considering $n + (3 + 2 * n2)$ points, and so runtimes are fine. As an example, we might use 301 points in the first layer and 20 in the second, meaning we are implicitly considering a grid of 6301 points on next period assets, but only have to actually evaluate 344 points in two layers. Most important for the GPU, there are just two steps in serial, and everything else is done in parallel within these two layers.

0.1 Pseudocode

This is the pseudocode for solving a simple finite horizon value function problem. One endogenous state, no decision variable, one markov shock. Let $a_grid = \{a_1, \ldots, a_n\}$ be the grid on assets (for both today and tomorrow), and $z\_grid = \{z_1, \ldots, z_m\}$ be the grid on the markov shock and let pi_z be the associated markov transition matrix.

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]

for j count backward from N_j to 1 **do**
  Get θ_j from Θ
  for All values z **do**
    Calculate $EV_j(a', z) = E[V_{j+1}(a', z')]$ ▷ For $j = N_j$ this is zero
    for All values of (a, z) **do**
      First Layer
      Solve for optimal *aprime* on grid:
        $g_j^1(a, z) = argmax_{i=1,\ldots,n} F(a\_grid(i), a, z) + \beta EV_j(a\_grid(i), z)$
      Note: $a\_grid(i)$ is $i - th$ grid point in grid on *aprime*, and we

[2] This is obviously not important to being able to do grid interpolation layer, just keeps things simple in the code.

[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 $g_j^1(a,z)$ as an index for the optimal aprime, not a value.

Second Layer

Create second layer grid:

$aL2\_grid = linspace(g_j(a,z)-1, g_j(a,z)+1, 3+2*n2)$ ▷ Given (a,z)

Interpolate $EV_j(a,z)$, which exists on the first layer grid,

onto the second layer grid to get $\hat{EV}_j(a,z)$.

Solve for optimal *aprime* on second layer grid:

$g_j^2(a,z) = argmax_{i=1,...,3+2*n2} F(aL2\_grid(i),a,z) + \beta \hat{EV}_j(aL2\_grid(i),z)$

end for

end for

end for

Turn the two layer optimal indexes $[g_1^1, g_2^1, ..., g_{N_j}^1]$ and $[g_1^2, g_2^2, ..., g_{N_j}^2]$ into some optimal policy $[g_1, g_2, ..., g_{N_j}]$. ▷ Explained below how I do this

return $[V_1, V_2, ..., V_{N_j}]$, $[g_1, g_2, ..., g_{N_j}]$ ▷ V are the max from second stage (g was argmax)

The notation $EV_j(a',z) = E[V_{j+1}(a',z')]$ is capturing that since z is a markov, the expected value $E[V_{j+1}(a',z')]$ can be expressed as a function of $(a'z)$ (by taking expectations over z' using the transition matrix pi_z which depends on z); this follows from z is markov.

One thing worth discussing is what form the optimal policy $[g_1, g_2, ..., g_{N_j}]$ should be stored in. I choose to store it as the 'index of the lower grid point of a_grid' (which is a number from 1 to $n-1$) together with the index of the second layer (which is a number from 1 to $n2+2$). When using the policy to create the agent distribution we will want the 'upper and lower grid points of a_grid' (the upper is just lower plus 1, so that is easy) and the 'weights' which can be trivially calculated from the second layer index (weight of lower grid point is $1-(secondlayerindex-1)/(n2+1)$; because we placed the second layer points evenly spaced). When using the policy to evaluate fns on the agent dist we want the aprime value, which we can trivially do by creating the 'fine' grid, $aprime\_grid = interp1(1:1:n, a\_grid, linspace(1,n,n+n2*(n-1)))$ and getting the index for the fine grid as 'fine grid index'$=n*$('index of the lower grid point of a_grid'-1)+'second layer index', then simply use the index for the fine grid to get the grid value from the fine grid. Since we created the indexes while solving the value fn problem,[4] it makes sense to store these, and only get the weights as needed for the agent dist.

One minor detail I have omitted above. The index we get solving the first layer is a 'midpoint' for the second layer. But if it is at the edges of the grid we cannot use it as a midpoint for the second layer. So just check if it is the first/last grid point, and if so then add/subtract one, respectively, before using it as midpoint for second layer.

[4] When we solve the first layer we got the 'midpoint', switching this to the 'lower grid point' just requires substracting one from it.

Decision Variables and Exogenous Shocks

The obvious approach is to apply this idea 'conditional on' any decision variables and exogenous shocks. This is exactly how VFI Toolkit handles it.

Note that when there is a decision variable this does rather change how the 'first layer' is done, relative to pure discretization. In pure discretization we would find the optimal $(d*, aprime*)$, which is a single value for each of d and $aprime$ (for each point in todays state-space). But now in the first layer we have to find the optimal $aprime$ for each possible d, so $aprime*(d)$, which is a single value of $aprime$ for each possible value of d, and the in the second layer we create a different second layer grid for each possible value of d. The second layer is more standard as now we just want the optimal $(d*, aprime*)$.

Linear interpolation and evenly spaced points

Until now we have not specified how the interpolation should be done. In VFI Toolkit linear interpolation is used.

Everything described above could easily be done without the second layer grids being evenly spaced, and by using any interpolation other than linear. The evenly spaced grids are intuitive and easy, and it is not obvious that there is an alternative choice that would meaningully improve performance (the choice of a good *a_grid* is likely much more important, as are the choice of n and $n2$). As for linear interpolation, you could easily use something else. I have not tested the alternatives: they will increase runtime and improve accuracy, but will the improved accuracy be worth the increased runtime? The code in VFI Toolkit does the interpolation using Matlab's 'interp1()', so if anyone does want to test them it is as trivial as changing *interp1(a_grid,EV,aprime_grid)* to *interp1(a_grid,EV,aprime_grid,'spline')*, or similar (note, not all the interpolation methods in *interp1()* work with GPU).

VFI Toolkit

You turn on the use of grid interpolation layer by setting $vfoptions.gridinterplayer = 1$, and you must also say what the number of points to use in the second layer between the first layer grid points, $n2$ above, which you do by setting, e.g., $vfoptions.ngridinterp = 20$.

Normally the first dimension of 'Policy' (the policy function created by the value fn iteration commands) contains the indexes for each of the policies (so if you have one decision variable and one endogenous state, there would be two policies and the second would correspond to endogenous state). When you use the grid interpolation layer the endogenous state is now two indices, one for the 'lower grid point of first layer' and the second for the 'second layer index'; these are described

above.

Note that because the interpretation of Policy has changed, we need to tell all the other commands about it so they know what to do. Hence you also need to set $simoptions.gridinterplayer = vfoptions.gridinterplayer$ and $simoptions.ngridinterp = vfoptions.ngridinterp$.

Combination with divide-and-conquer

Notice that the first layer is essentially just the pure discretization problem. So we can apply divide-and-conquer to the first layer just as we would to any other problem. This gives us an algorithm where we do two-layer grid interpolation and use divide-and-conquer on the first layer; the second layer just standard. This is what VFI Toolkit does if you set both $vfoptions.divideandconquer = 1$ and $vfoptions.gridinterplayer = 1$.

Agent Dist and EvalFnOnAgentDist

When using the grid interpolation layer ($vfoptions.gridinterplayer = 1$) the $Policy$ is now the 'lower grid point index' and the 'interpolation layer index' for the next period endogenous state. We can simply keep the 'lower grid point index' and convert the 'interpolation layer index' into a probability of the lower grid point. We then use the 'n probs' setting for iterating on the agent distribution. This is explained in the Psuedocodes pdf for VFI Toolkit.

For evaluating functions on the agent distribution we just need to convert $Policy$ to $PolicyVals$ (from indexes to values) and then the rest of the commands are standard. When using the policy to evaluate fns on the agent dist we want the aprime value, which we can trivially do by creating the 'fine' grid, $aprime\_grid = interp1(1:1:n, a\_grid, linspace(1, n, n + n2 * (n-1)))$ and getting the index for the fine grid as 'fine grid index'=$n *$('index of the lower grid point of a_grid'-1)+'second layer index', then simply use the index for the fine grid to get the grid value from the fine grid

Grid Interpolation Layer in Infinite Horizon VFI

How can we implement the grid interpolation layer in infinite horizon problems? Because of the iteration we want to try and pre-compute the 'second layer' so that it can just be reused each iteration. But as we iterate on the first layer the location of the policy will change, and so we will need to recompute the second layer, which will really hurt the runtimes. VFI Toolkit current implements two approaches that reflect compromises being made elsewhere to avoid having to recompute the second layer, these two are called 'pre-GI' and 'post-GI' (GI is short for grid interpolation layer).

In pre-GI we simply compute the second layer everywhere. So we use the whole $aprime\_grid = interp1(1:1:n, a\_grid, linspace(1, n, n + n2 * (n-1)))$ to compute the return function. Note that

this means having more points in *aprime* than in a, but this does not meaningfully change how to do Value Fn Iteration and so we just use the same standard VFI described in the VFI pseudocode document (with 'refinement' to handle decision variables if the model has those). The only change is that we now do the VFI in two stages taking a multi-grid approach. In the first stage we require *aprime* to reside on *a_grid* (so without interpolation) and perform standard pure-discretized VFI. The V that we get from solving this first stage is then used as an initial guess to perform VFI with the 'interploted' *aprime_grid*. Notice that the $ReturnFn$ matrix used in the first stage is just a subset of that from the second matrix, and so we compute the second $ReturnFn$ matrix before the first stage, and just take the smaller $ReturnFn$ matrix for the first stage out of this. This approach is powerful, as it is guaranteed to get the correct solution on the interpolated *aprime_grid*, but it is very compute and memory intensive as it requires creating the $ReturnFn$ matrix for the second stage which is very large. The first stage in pre-GI does not reduce the memory use in any way, it just speeds up the solution by giving a good initial guess for V by taking a multi-grid approach.

In post-GI, we have the same first stage as for pre-GI. That is, we first solve the pure discretized VFI problem on the *a_grid* (so the same VFI problem we would solve if we were not using grid interpolation layer at all; if the problem has a decision varible 'refinement' is used to handle this). As with pre-GI this first stage gives us a V that we use as the initial guess for the second stage. Note that unlike pre-GI, here we just create the first stage $ReturnFn$ matrix directly (which obviously takes the exact same memory and compute as if we were solving without the grid interpolation layer). In post-GI we also use the optimal policy from the first stage $Policy^*_{a'}$. In the second stage we only consider policies that are within $+$-$vfoptions.maxaprimediff$ of the first stage optimal policy $Policy^*_{a'}$, and we further add in the $vfoptions.ngridinterp$ points. So in the second stage we are considering $(1 + vfoptions.ngridinterp) * (2 * vfoptions.maxaprimediff) + 1$ possible points for *aprime*. Notice that this is substantially less points that we were considering in the second stage of pre-GI ($(1 + vfoptions.ngridinterp) * (n\_a - 1) + 1$), and in practice is often also less points that we considered in the first stage (n_a points) so that the memory demands of post-GI are often of the same order as the memory demands of not using grid interpolation layer at all. It should be noted however that we are not guaranteed that the assumption that the first-stage was within $vfoptions.maxaprimediff$ points of the optimal is not certain to hold, so what if it doesn't? The post-GI will always be more accurate than not using grid interpolation layer, but if $vfoptions.maxaprimediff$ is set too small then post-GI will be (marginally) less accurate that pre-GI (post-GI won't quite get the full accuracy that the grid interpolation layer makes possible). In practice, the user can easily try out different settings for $vfoptions.maxaprimediff$, and as long as increasing $vfoptions.maxaprimediff$ does not change the answer, then it is already large enough. Since you are guaranteed that even getting $vfoptions.maxaprimediff$ you are still more accurate than not using grid interpolation layer getting it wrong is not a big deal. Note that the size of $vfoptions.maxaprimediff$ depends on all sort of things, include how big n_a was to begin with; if n_a is large, the first stage should be very close to the 'correct' grid interpolation layer

solution and so $vfoptions.maxaprimediff$ can be small.

References

Robert Kirkby. VFI toolkit, v2. Zenodo, 2022. doi: https://doi.org/10.5281/zenodo.8136790.