Divide and Conquer

Robert Kirkby*

February 6, 2024

Abstract

Gordon & Qiu (2018) introduce a 'binary monotonicity' algorithm that exploits monotonicity to solve value function problems faster. This brief pdf describes the adaption to GPU that is implemented in VFI Toolkit.

Keywords: Value Function Iteration, Divide and Conquer, 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 already read Gordon and Qiu (2018). The basic idea is that for almost all value function problems, we know that the solution will be 'monotone', in the sense that $aprime(a)$ is a (weakly) increasing function of a; where $aprime$ is next period endogenous state and a is this period endogenous state. We can exploit this monotonicity to check less points (as many would fail to be monotone and so cannot be correct) and so can find the solution faster and using less memory.

To start, consider the naive pure discretization approach that is default in VFI Toolkit. Say we put 12 points on the endogenous state. Then we will check all 12-by-12 points, and find the solution (which is to say that for each grid point of a we find the point in $aprime$ that maximizes the value function problem). Figure 1 illustrates this idea, the pale blue indicates the points we have evaluated, and the dark blue indicates the solution. So pure discretization is going to have to try all 144 (12 times 12) points. Because we can use the GPU this is workable as they can all be evaluated in parallel, but it would be very slow on a CPU.

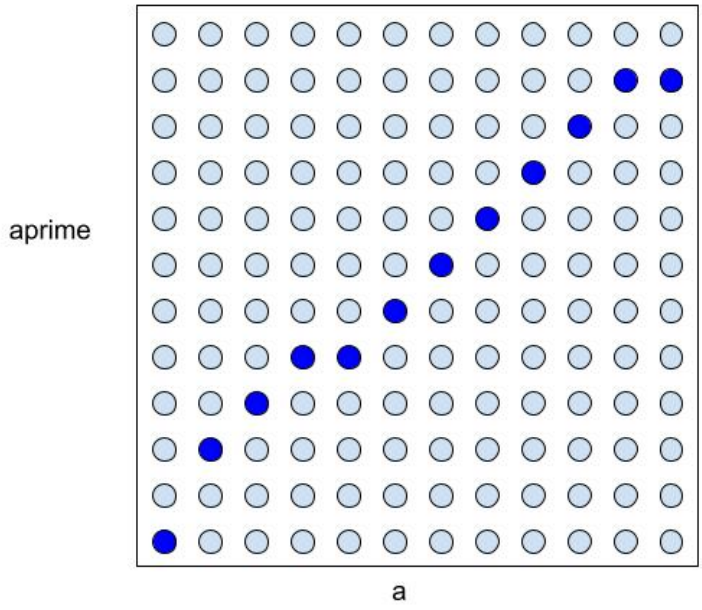

Figure 1: Pure Discretization

'Simple monotonicity' takes advantage of that fact that we know that $aprime$ is weakly increasing in a. We start with the first grid point of a, and look at all the possible $aprime$ to find the solution for that first a grid point. Since we know that the second grid point of a must have an $aprime$ solution for a larger value of $aprime$ we only need to check 'larger' $aprime$ values. Then for the third grid point of a, we only need to consider values of $aprime$ larger than the $aprime$ that was the solution for the second grid point. Figure 2 illustrates this idea, the pale blue indicates the points we have evaluated, and the dark blue indicates the solution. Notice that 'simply monotonicity' has

substantially reduced the number of points we have to evaluate.

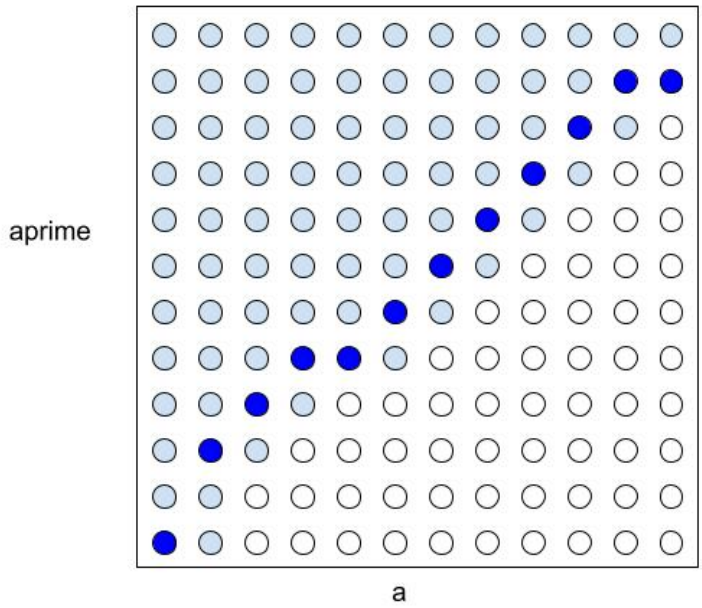

Figure 2: Simple Monotonicity

'Binary monotonicity' introduced by Gordon and Qiu (2018) improves on this further. We first look at the smallest and largest a grid points, check all *aprime*, and find their solutions. We then turn to the middle grid point of a, and only have to check above (or equal to) the *aprime* from the first a grid point, and below the *aprime* of the last a grid point. Next, the lower quarter grid point of a, where we only have to check above *aprime* of the smallest a grid point, and below *aprime* of the middle a grid point. Similarly, the upper quarter grid point of a, where we only have to check above *aprime* of the middle grid point, and below *aprime* of the largest a grid point. Keep subdividing each interval in 2 (this description is a bit rough, see Gordon and Qiu (2018) for precise formulation; hopefully I have communicated the rough idea). Figure 3 illustrates this idea, the pale blue indicates the points we have evaluated, and the dark blue indicates the solution. You can see just how few points binary monotonicity has to evaluate compared to the alternatives.

On the cpu, this is about the end of the story. Fewer points is better. But on the GPU there is more too it. Yes, less points to evaluate will be faster on the GPU than more points to evaluate. But serial on the GPU will be much slower to evaluate than parallel. And binary monotonicity is *very* serial.

So now we come to the idea of 'two-level monotonoicity', which is what this note introduces and is implemented in VFI Toolkit. We choose 'n' points for the first level, let's make it 3 for our example. In the first step we solve the 'first level: we choose $n = 3$ equally spaced points on the a

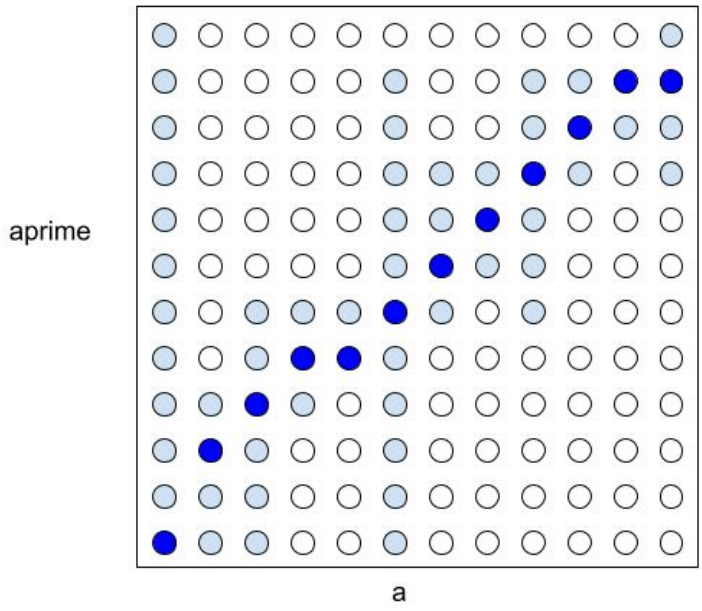

Figure 3: Binary Monotonicity

grid (these n will include both endpoints)[1] and for each of these we evaluate all *aprime* to find the solution. For the second level we use a for-loop counting ii over $n-1$ iterations, and solve all a grid points between the ii-th and $ii+1$-th a grid points from level one, and allow a range for *aprime* from the solutions corresponding to these two points from the first level. Figure 4 illustrates this idea, the pale blue indicates the points we have evaluated, and the dark blue indicates the solution.

Obviously, 'two-level monotonicity' involves evaluating more points than 'binary monotocity'. So on a CPU 'binary monotocity' will be faster. But on a GPU, 'two-level monotonicity' provides a reasonable compromise between evaluating less points, and being more parallel/less serial.

My rough tests suggest that for GPU 'two-level monotonicity' is a good balance. In practice, I often seem to see little gain from using 11 or 15 points in the first level, rather than just 5. This suggests that the costs of serialization are outweighing the benefits of evaluating less points fairly early on.

References

Grey Gordon and Shi Qiu. A divide and conquer algorithm for exploiting policy function monotonicity. Quantitative Economics, 9(2), 2018. doi: https://doi.org/10.3982/QE640.

[1] In Matlab code we choose the indexes $level1ii$ as $level1ii = round(linspace(1, n\_a, vfoptions.level1n))$;, where n_a is number of grid points for a, and $vfoptions.level1n$ is the number n of grid points to be used in the first level.

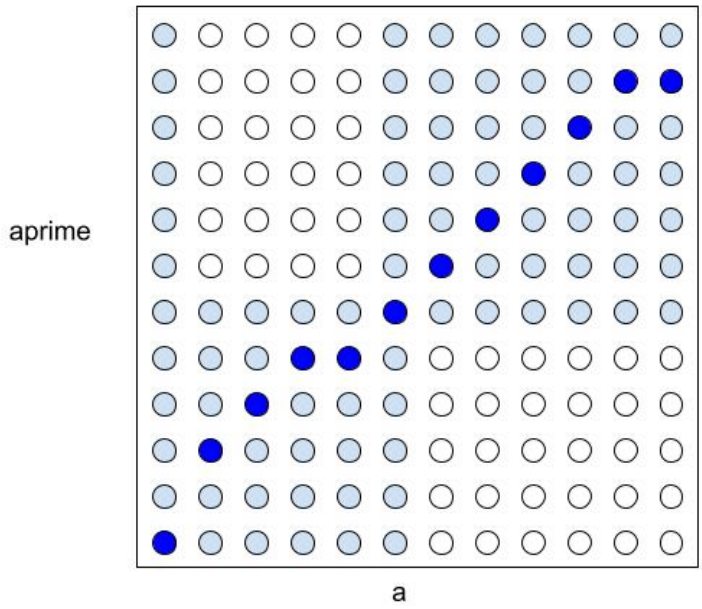

Figure 4: Two-level Monotonicity

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