Value Function Iteration in Matlab

Michael Kotrous

University of Georgia

October 19, 2023

Matlab

- Popular programming language among economists
- Advantages:
  - Efficient matrix math
  - Built-in functionality
  - Simple syntax
  - Documentation
- Disadvantages:
  - Proprietary
  - Limited add-ons
  - Non-numeric data

Getting Started

- UGA installation guide
- Useful add-ons
  - Optimization
  - Parallel Computing
- Useful Links
  - Matlab documentation
  - eLC: Download Matlab > Tutorials > `matlab_tutorial_files.zip`

Neoclassical Growth Model

$$V(k) = \max_{k'}\{\log(zk^{\alpha} + (1-\delta)k - k') + \beta V(k')\}$$

subject to

$$0 \leq k' \leq zk^{\alpha} + (1-\delta)k$$

- When $\delta = 1$, $g(k) = \alpha\beta zk^{\alpha}$
- When $\delta < 1$, solve numerically

Value Function Iteration

- Solve models numerically
- Solution: approximation of $V(k)$
- Method: Iterate on $V(k)$, reach fixed point
- Works b/c Bellman = contraction mapping
- Implement grid search in Matlab

VFI Algorithm

1. Calibrate parameters $(\alpha, \beta, \delta, z)$
2. Set tolerance $\varepsilon > 0$
3. Discretize state space

$$\mathcal{K} = \{k_1, k_2, \dots, k_n\}$$

4. Calculate flow utility $u(k, k')$ for $(k, k') \in \mathcal{K} \times \mathcal{K}$

VFI Algorithm (cntd.)

5. Define initial guess

$$V_0(k) = \{V_0(k_1), V_0(k_2), \ldots, V_0(k_n)\}$$

6. For each $k \in \mathcal{K}$, solve

$$V_1(k) = \max_{k'}\{\log(zk^\alpha + (1-\delta)k - k') + \beta V_0(k')\}$$

subject to

$$0 \leq k' \leq zk^\alpha + (1-\delta)k$$
$$k' \in \mathcal{K}$$

7. Calculate $||V_1(k) - V_0(k)||$
8. Stopping criteria $||V_{n+1}(k) - V_n(k)|| < \varepsilon$

Calibration

Table 1: Calibrated Parameters, Tolerance

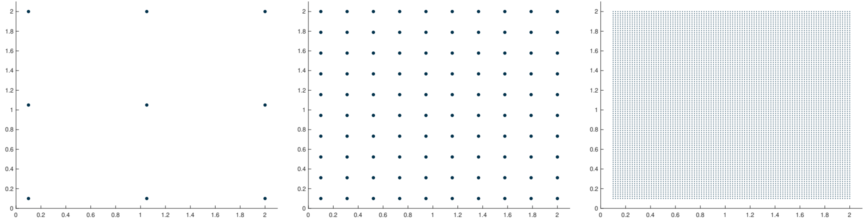
Parameter	**Value(s)**
α	0.39
β	0.95
δ	1
z	274
Tolerance	**Value**
ε	10^{-8}

Calibration (in code)

```
a = 0.39;
b = 0.95;
d = 1;
z = 274;

tol = 1e-8;
```

Discretize State Space



(a) 3 nodes (b) 10 nodes (c) 100 nodes

Increasing nodes

- ⋆ Pro: ↑ precision
- ⋆ Con: ↑ compute time

Discretize State Space (in code)

vfi_lecture.m

```
n = 5;
kss = ((z*a)./(1/b - 1 + d))^(1/(1-a));
kgrid = zeros(1,n);
kgrid(:) = griddle(0.1*kss, 2*kss, n, 1.5);
```

griddle.m

```
function g = griddle(a,b,n,p)
    gr = zeros(1,n);
    gr(1) = a;
    gr(n) = b;
    for k = 2:n-1
        gr(k) = a + (b-a)*((k-1)/(n-1))^p;
    end
    g = gr;
end
```

Flow Utility

$$u(k, k') = \begin{cases} \log(zk^\alpha + (1-\delta)k - k') & \text{if } zk^\alpha + (1-\delta)k - k' > 0 \\ -10^{20} & \text{otherwise} \end{cases}$$

Table 2: Flow Utility for all $(k, k') \in \mathcal{K} \times \mathcal{K}$

		k'				
		k_1	k_2	k_3	k_4	k_5
	k_1	7.5737	7.3024	6.4588	-1e20	-1e20
	k_2	8.0852	7.9315	7.5694	6.7369	-1e20
k	k_3	8.4241	8.3171	8.0857	7.6745	6.7524
	k_4	8.6458	8.5610	8.3844	8.0966	7.5941
	k_5	8.8087	8.7371	8.5912	8.3638	8.0039

Flow Utility (in code)

```
ucgrid = zeros(n,n);
for i = 1:n
    for j = 1:n
        c = z*kgrid(i)^a + (1-d)*kgrid(i) - kgrid(j);
        if c > 0
            ucgrid(i,j) = log(c);
        else
            % exclude infeasible choices
            ucgrid(i,j) = -1e20;
        end
    end
end
```

Define Initial Guess

- Solution: fixed point
- Blackwell's sufficient conditions
  - Discounting
  - Monotonicity
- Bellman operator = contraction mapping
- Any initial guess works!

```
V = linspace(0,1,n);
Tv = zeros(1,n);
g = zeros(1,n);
```

Update Guess

```
for i = 1:n
    [vmax, kmax] = max(ucgrid(i,:) + b*V);
    Tv(i) = vmax;
    g(i) = kgrid(kmax);
end
```

Update Guess (cntd.)

Table 3: Updating Value, Policy Functions

		k'						
		k_1	k_2	k_3	k_4	k_5	Tv	$g(k)$
	k_1	7.5737	7.5399	6.9338	0.7125	0.9500	7.5737	k_1
	k_2	8.0852	8.1690	8.0444	7.4494	0.9500	8.1690	k_2
k	k_3	8.4241	8.5546	8.5607	8.3870	7.7024	8.5607	k_3
	k_4	8.6458	8.7985	8.8594	8.8091	8.5441	8.8594	k_3
	k_5	8.8087	8.9746	9.0662	9.0763	8.9539	9.0763	k_4

Check Convergence

Table 4: Distance between Initial, Updated Guess

V	Tv	$V - Tv$
0	7.5737	-7.5737
0.25	8.1690	-7.9190
0.50	8.5607	-8.0607
0.75	8.8594	-8.1094
1	9.0763	-8.0763

```
err1 = norm(V-Tv);          % 17.7774
err2 = abs(max(V-Tv));      %  8.1094
```

Value Function Iteration

```
V = Tv; % update V
err = err1; % use Euclidean norm for stopping criteria
it = 0; % count iterations
while err > tol && it < 500
    for i = 1:n
        [vmax, kmax] = max(ucgrid(i,:) + b*V);
        Tv(i) = vmax;
        g(i) = kgrid(kmax);
    end

    % check for convergence and update guess
    err = norm(Tv-V);
    V = Tv;
    it = it+1;
end
```

Results

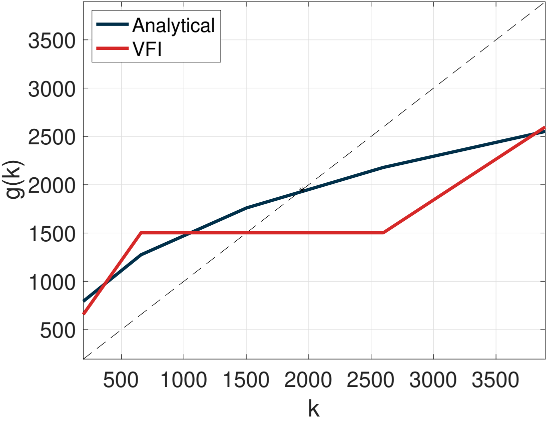


Figure 1: Policy Function, $\delta = 1$, $n = 5$
Solved in 0.000481 seconds

Results (cntd.)

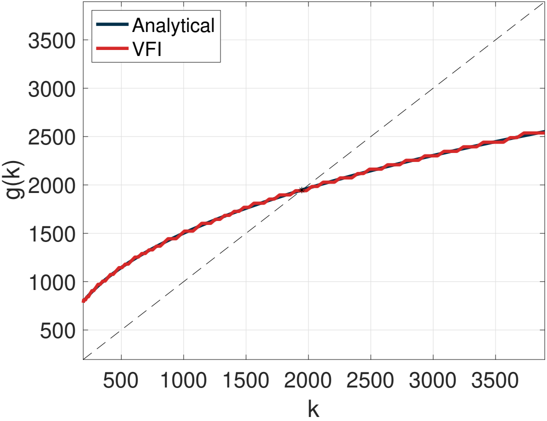


Figure 2: Policy Function, $\delta = 1$, $n = 100$
Solved in 0.003190 seconds

Results (cntd.)

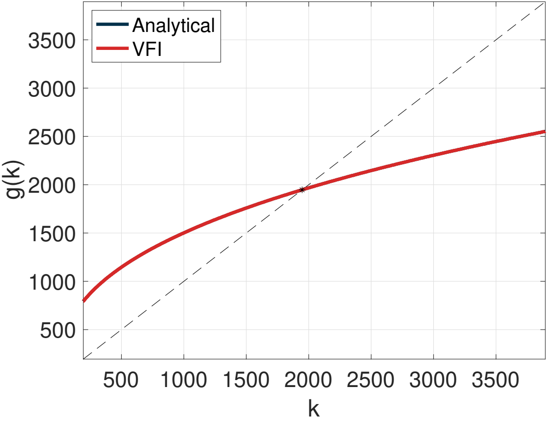


Figure 3: Policy Function, $\delta = 1$, $n = 1,000$
Solved in 0.003492 seconds

Results (cntd.)

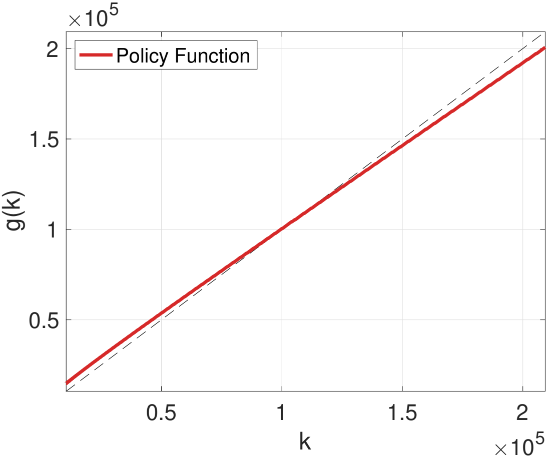


Figure 4: Policy Function, $\delta = 0.04$, $n = 1,000$
Solved in 0.003404 seconds

Conclusion

- ⋆ Many models can't be solved analytically
- ⋆ VFI solves models numerically
- ⋆ We implemented VFI in Matlab
- ⋆ We restricted policy function to grid
  - ⟶ Pros: simpler code, flow utility before iteration
  - ⟶ Cons: inaccurate when n small, high n inefficient
- ⋆ Alternative: interpolate between grid points
  - ⟶ Accurate policy functions with small n
  - ⟶ Useful for multi-dimensional state spaces
  - ⟶ See Karen Kopecky and Eric Sim's VFI notes