Lecture 9

Advanced Methods for Numerical Dynamic Models

Ivan Rudik
AEM 7130

Roadmap

1. Regression
2. Endogenous grid method
3. Envelope condition method
4. Modified policy iteration

Chebyshev regression

Chebyshev regression works just like normal regression

Chebyshev regression

Chebyshev regression works just like normal regression

For a degree n polynomial approximation, we choose $m > n + 1$ grid points

Chebyshev regression

Chebyshev regression works just like normal regression

For a degree n polynomial approximation, we choose $m > n + 1$ grid points

We then build our basis function matrix Ψ, but instead of being $n \times n$ it is $m \times n$

Chebyshev regression

Chebyshev regression works just like normal regression

For a degree n polynomial approximation, we choose $m > n + 1$ grid points

We then build our basis function matrix Ψ, but instead of being $n \times n$ it is $m \times n$

Finally we use the standard least-squares equation

$$c = (\Psi'\Psi)^{-1}\Psi'y$$

Chebyshev regression

Chebyshev regression works just like normal regression

For a degree n polynomial approximation, we choose $m > n + 1$ grid points

We then build our basis function matrix Ψ, but instead of being $n \times n$ it is $m \times n$

Finally we use the standard least-squares equation

$$c = (\Psi'\Psi)^{-1}\Psi' y$$

Fun fact: $(\Psi'\Psi)^{-1}\Psi'$ is a pseudoinverse called the Moore-Penrose matrix inverse

Chebyshev regression

Chebyshev regression works just like normal regression

For a degree n polynomial approximation, we choose $m > n + 1$ grid points

We then build our basis function matrix Ψ, but instead of being $n \times n$ it is $m \times n$

Finally we use the standard least-squares equation

$$c = (\Psi'\Psi)^{-1}\Psi' y$$

Fun fact: $(\Psi'\Psi)^{-1}\Psi'$ is a pseudoinverse called the Moore-Penrose matrix inverse

We can apply Chebyshev regression to even our regular tensor approaches, this has the advantage of dropping higher order terms which often oscillate due to error, giving us a smoother approximation

Chebyshev regression: practice

Go back to our original VFI example and convert it to a regression approach

```julia
using LinearAlgebra
using Optim
using Plots
params = (alpha = 0.75, beta = 0.95, eta = 2,
                steady_state = (0.75*0.95)^(1/(1 - 0.75)), k_0 = (0.75*0.95)^(1/(1 - 0.75))*.75,
                capital_upper = (0.75*0.95)^(1/(1 - 0.75))*1.5, capital_lower = (0.75*0.95)^(1/(1 - 0.75))/2,
                num_basis = 7, num_points = 9, tolerance = 0.0001, fin_diff = 1e-6, mpi_start = 5);

coefficients = zeros(params.num_basis);
coefficients[1:2] = [100 5];
```

Chebyshev regression: practice

```
cheb_nodes(n) = cos.(pi * (2*(1:n) .- 1)./(2n))
```

```
## cheb_nodes (generic function with 1 method)
```

```
grid = cheb_nodes(params.num_points) # [-1, 1] grid
```

```
## 9-element Array{Float64,1}:
##   0.984807753012208
##   0.8660254037844387
##   0.6427876096865394
##   0.3420201433256688
##   6.123233995736766e-17
##  -0.3420201433256687
##  -0.6427876096865394
##  -0.8660254037844387
##  -0.984807753012208
```

```
expand_grid(grid, params) = (1 .+ grid)*(params.capital_upper - params.capital_lower)/2 .+ params.capital_lower
```

```
## expand_grid (generic function with 1 method)
```

Chebyshev regression: practice

```
# Chebyshev polynomial function
function cheb_polys(x, n)
    if n == 0
        return 1                    # T_0(x) = 1
    elseif n == 1
        return x                    # T_1(x) = x
    else
        cheb_recursion(x, n) =
            2x.*cheb_polys.(x, n - 1) .- cheb_polys.(x, n - 2)
        return cheb_recursion(x, n) # T_n(x) = 2xT_{n-1}(x) - T_{n-2}(x)
    end
end;

basis_matrix = [cheb_polys.(grid, n) for n = 0:params.num_basis - 1];
basis_matrix = hcat(basis_matrix...);
basis_inverse = inv(basis_matrix'*basis_matrix)*(basis_matrix'); # pre-compute pseudoinverse for regressions
```

```
## 7×9 Array{Float64,2}:
##  0.111111   0.111111      0.111111   …   0.111111    0.111111      0.111111
##  0.218846   0.19245       0.142842      -0.142842   -0.19245      -0.218846
##  0.208821   0.111111     -0.0385885     -0.0385885   0.111111      0.208821
##  0.19245   -5.15976e-17  -0.19245        0.19245     5.02235e-18  -0.19245
##  0.170232  -0.111111     -0.208821      -0.208821   -0.111111      0.170232
```

Chebyshev regression: practice

```
shrink_grid(capital) = 2*(capital - params.capital_lower)/(params.capital_upper - params.capital_lower) - 1;
eval_value_function(coefficients, capital, params) =
    coefficients' * [cheb_polys.(shrink_grid(capital), n) for n = 0:params.num_basis - 1];
```

Chebyshev regression: practice

```
function loop_grid_regress(params, basis_inverse, basis_matrix, grid, capital_grid, coefficients)
    max_value = -.0*ones(params.num_points);
    consumption_store = -.0*ones(params.num_points);

    for (iteration, capital) in enumerate(capital_grid)
        function bellman(consumption)
            capital_next = capital^params.alpha - consumption
            cont_value = eval_value_function(coefficients, capital_next, params)
            value_out = (consumption)^(1-params.eta)/(1-params.eta) + params.beta*cont_value
            return -value_out
        end;

        results = optimize(bellman, 0.00*capital^params.alpha, 0.99*capital^params.alpha)
        max_value[iteration] = -Optim.minimum(results)
        consumption_store[iteration] = Optim.minimizer(results)
    end

    return max_value, consumption_store
end;
```

Chebyshev regression: practice

```julia
function solve_vfi_regress(params, basis_inverse, basis_matrix, grid, capital_grid, coefficients)

    max_value = -.0*ones(params.num_points);
    error = 1e10;
    value_prev = .1*ones(params.num_points);
    coefficients_store = Vector{Vector}(undef, 1)
    coefficients_store[1] = coefficients
    iteration = 1

    while error > params.tolerance
        max_value, consumption_store = loop_grid_regress(params, basis_inverse, basis_matrix, grid, capital_grid
        coefficients = basis_inverse*max_value
        error = maximum(abs.((max_value - value_prev)./(value_prev)))
        value_prev = deepcopy(max_value)
        if mod(iteration, 5) == 0
            println("Maximum Error of $(error) on iteration $(iteration).")
            append!(coefficients_store, [coefficients])
        end
        iteration += 1
    end

    return coefficients, max_value, coefficients_store
end;
```

Chebyshev regression: practice

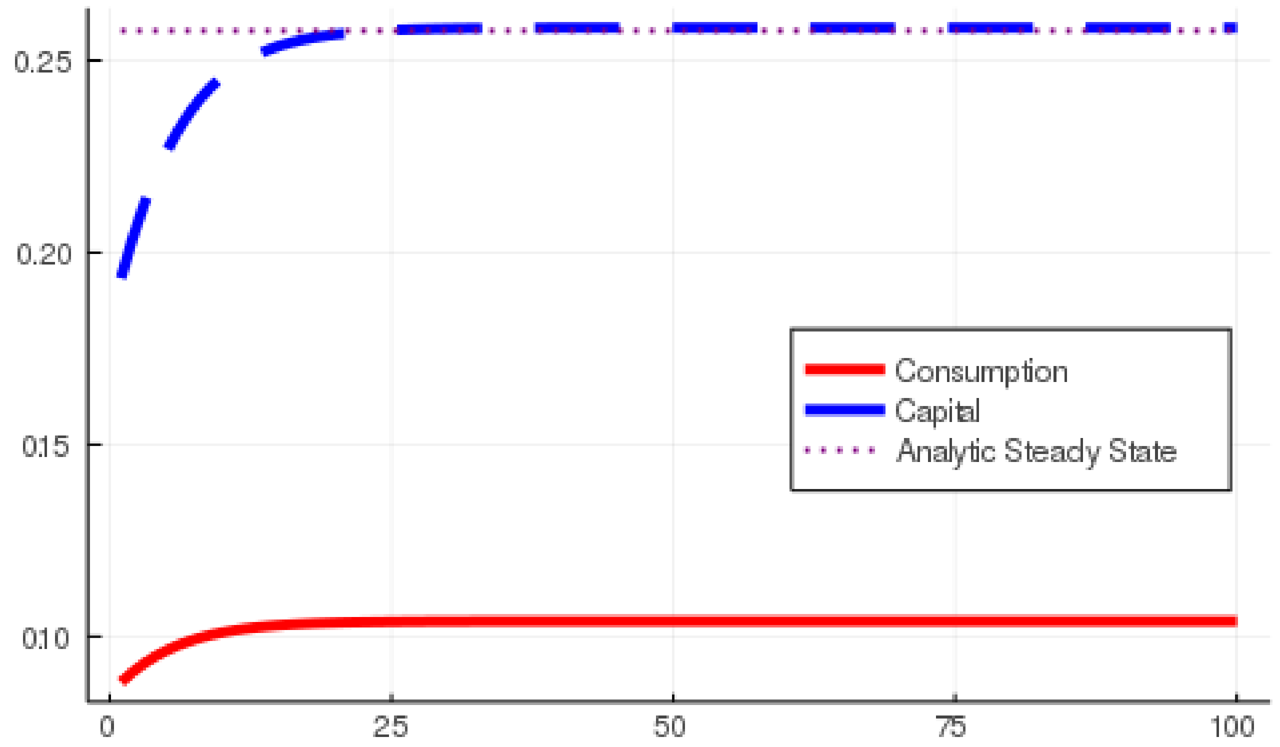
```
@time solution_coeffs, max_value, intermediate_coefficients =
    solve_vfi_regress(params, basis_inverse, basis_matrix, grid, capital_grid, coefficients)
```

```
## Maximum Error of 0.3365646232156377 on iteration 5.
## Maximum Error of 15.324437748784836 on iteration 10.
## Maximum Error of 0.19176452946373068 on iteration 15.
## Maximum Error of 0.07999511358219019 on iteration 20.
## Maximum Error of 0.04557549396818246 on iteration 25.
## Maximum Error of 0.029268260045591604 on iteration 30.
## Maximum Error of 0.02000715481671002 on iteration 35.
## Maximum Error of 0.014198326541472671 on iteration 40.
## Maximum Error of 0.01032384690730612 on iteration 45.
## Maximum Error of 0.007632084134370365 on iteration 50.
## Maximum Error of 0.005708492913566279 on iteration 55.
## Maximum Error of 0.004305925733500575 on iteration 60.
## Maximum Error of 0.003268177593053356 on iteration 65.
## Maximum Error of 0.0024920065993268197 on iteration 70.
## Maximum Error of 0.001906769094882636 on iteration 75.
## Maximum Error of 0.0014628021447215872 on iteration 80.
## Maximum Error of 0.0011244465442097609 on iteration 85.
## Maximum Error of 0.0008656712708535016 on iteration 90.
## Maximum Error of 0.0006672266517799315 on iteration 95.
## Maximum Error of 0.000514733393872197 on iteration 100.
```

Chebyshev regression: practice

```julia
function simulate_model(params, solution_coeffs, time_horizon = 100)
    capital_store = zeros(time_horizon + 1)
    consumption_store = zeros(time_horizon)
    capital_store[1] = params.k_0

    for t = 1:time_horizon
        capital = capital_store[t]

        function bellman(consumption)
            capital_next = capital^params.alpha - consumption
            cont_value = eval_value_function(solution_coeffs, capital_next, params)
            value_out = (consumption)^(1-params.eta)/(1-params.eta) + params.beta*cont_value
            return -value_out
        end;

        results = optimize(bellman, 0.0, capital^params.alpha)
        consumption_store[t] = Optim.minimizer(results)
        capital_store[t+1] = capital^params.alpha - consumption_store[t]
    end

    return consumption_store, capital_store
end;
```

Chebyshev regression: practice



Endogenous grid method (Carroll, 2006)

Suppose now we are working with a model with an inelastic labor supply with logarithmic utility $\eta = 1$, and capital that fully depreciates

Endogenous grid method (Carroll, 2006)

Suppose now we are working with a model with an inelastic labor supply with logarithmic utility $\eta = 1$, and capital that fully depreciates

Leisure does not enter the utility function nor does labor enter the production function, i.e. $B = 0, l = 1$

Endogenous grid method (Carroll, 2006)

Suppose now we are working with a model with an inelastic labor supply with logarithmic utility $\eta = 1$, and capital that fully depreciates

Leisure does not enter the utility function nor does labor enter the production function, i.e. $B = 0, l = 1$

This yields closed form solutions to the model

$$\begin{aligned} k_{t+1} &= \beta\alpha\theta_t k_t^\alpha \\ c_t &= (1-\beta\alpha)\theta_t k_t^\alpha \end{aligned}$$

Endogenous grid method (Carroll, 2006)

Suppose now we are working with a model with an inelastic labor supply with logarithmic utility $\eta = 1$, and capital that fully depreciates

Leisure does not enter the utility function nor does labor enter the production function, i.e. $B = 0, l = 1$

This yields closed form solutions to the model

$$\begin{aligned} k_{t+1} &= \beta\alpha\theta_t k_t^\alpha \\ c_t &= (1-\beta\alpha)\theta_t k_t^\alpha \end{aligned}$$

The endogenous grid method was introduced by Carroll (2006) for value function iteration

Endogenous grid method (Carroll, 2006)

The idea behind EGM is **super simple**

Endogenous grid method (Carroll, 2006)

The idea behind EGM is **super simple**

instead of constructing a grid on the current states, construct the grid on **future states** (making current states endogenous)

Endogenous grid method (Carroll, 2006)

The idea behind EGM is **super simple**

instead of constructing a grid on the current states, construct the grid on **future states** (making current states endogenous)

This works to our advantage because typically it is easier to solve for k given k' than the reverse

Endogenous grid method (Carroll, 2006)

The idea behind EGM is **super simple**

instead of constructing a grid on the current states, construct the grid on **future states** (making current states endogenous)

This works to our advantage because typically it is easier to solve for k given k' than the reverse

Let's see how this works

Endogenous grid method

1. Choose a grid $\{k'_m, \theta_m\}_{m=1,\dots,M}$ on which the value function is approximated
2. Choose nodes ϵ_j and weights ω_j, $j = 1, \dots, J$ for approximating integrals.
3. Compute next period productivity, $\theta'_{m,j} = \theta_m^{\rho} exp(\epsilon_j)$.
4. Solve for b and $\{c_m, k_m\}$ such that
   - (inner loop) The quantities $\{c_m, k_m\}$ solve the following given $V(k'_m, \theta'_m)$:
     - $u'(c_m) = \beta E\left[V_k(k'_m, \theta'_{m,j})\right]$,
     - $c_m + k'_m = \theta_m f(k_m) + (1-\delta)k_m$
   - (outer loop) The value function $\hat{V}(k, \theta; b)$ solves the following given $\{c_m, k'_m\}$:
     - $\hat{V}(k_m, \theta_m; b) = u(c_m) + \beta \sum_{j=1}^{J} \omega_j \left[\hat{V}(k'_m, \theta'_{m,j}; b)\right]$

Endogenous grid method

Focus the inner loop of VFI:

- (inner loop) The quantities $\{c_m, k_m\}$ solve the following given $V(k'_m, \theta'_m)$:
  - $u'(c_m) = \beta E\left[V_k(k'_m, \theta'_{m,j})\right]$,
  - $c_m + k'_m = \theta_m f(k_m) + (1-\delta)k_m$

Notice that the values of k' are fixed since they are grid points

Endogenous grid method

Focus the inner loop of VFI:

- (inner loop) The quantities $\{c_m, k_m\}$ solve the following given $V(k'_m, \theta'_m)$:
    - $u'(c_m) = \beta E\left[V_k(k'_m, \theta'_{m,j})\right]$,
    - $c_m + k'_m = \theta_m f(k_m) + (1-\delta)k_m$

Notice that the values of k' are fixed since they are grid points

This means that we can pre-compute the expectations of the value function and value function derivatives and let $W(k', \theta) = E[V(k', \theta'; b)]$

Endogenous grid method

Focus the inner loop of VFI:

- (inner loop) The quantities $\{c_m, k_m\}$ solve the following given $V(k'_m, \theta'_m)$:
  - $u'(c_m) = \beta E\left[V_k(k'_m, \theta'_{m,j})\right]$,
  - $c_m + k'_m = \theta_m f(k_m) + (1-\delta)k_m$

Notice that the values of k' are fixed since they are grid points

This means that we can pre-compute the expectations of the value function and value function derivatives and let $W(k', \theta) = E[V(k', \theta'; b)]$

We can then use the consumption FOC to solve for consumption, $c = [\beta W_k(k', \theta)]^{-1/\gamma}$ and then rewrite the resource constraint as,

$$(1-\delta)k + \theta k^{\alpha} = [\beta W_k(k', \theta)]^{-1/\gamma} + k'$$

Endogenous grid method

This is easier to solve than the necessary conditions we would get out of standard value function iteration

$$(k' - (1-\delta)k - \theta k^{\alpha})^{-\gamma} = \beta W_k(k', \theta')$$

Endogenous grid method

This is easier to solve than the necessary conditions we would get out of standard value function iteration

$$(k' - (1-\delta)k - \theta k^{\alpha})^{-\gamma} = \beta W_k(k', \theta')$$

Why?

Endogenous grid method

This is easier to solve than the necessary conditions we would get out of standard value function iteration

$$(k' - (1-\delta)k - \theta k^{\alpha})^{-\gamma} = \beta W_k(k', \theta')$$

Why?

We do not need to do any interpolation (k' is on our grid)

Endogenous grid method

This is easier to solve than the necessary conditions we would get out of standard value function iteration

$$(k' - (1-\delta)k - \theta k^{\alpha})^{-\gamma} = \beta W_k(k', \theta')$$

Why?

We do not need to do any interpolation (k' is on our grid)

We do not need to approximate a conditional expectation (already did it before hand and can do it with very high accuracy since it is a one time cost)

Endogenous grid method

This is easier to solve than the necessary conditions we would get out of standard value function iteration

$$(k' - (1-\delta)k - \theta k^{\alpha})^{-\gamma} = \beta W_k(k', \theta')$$

Why?

We do not need to do any interpolation (k' is on our grid)

We do not need to approximate a conditional expectation (already did it before hand and can do it with very high accuracy since it is a one time cost)

Can we make the algorithm better?

Endogenous grid method: turbo speed

Let's make a change of variables

$$Y \equiv (1-\delta)k + \theta k^{\alpha} = c + k'$$

Endogenous grid method: turbo speed

Let's make a change of variables

$$Y \equiv (1-\delta)k + \theta k^{\alpha} = c + k'$$

so we can rewrite the Bellman as

$$V(Y,\theta) = \max_{k'} \left\{ \frac{c^{1-\gamma}-1}{1-\gamma} + \beta E\left[V(Y',\theta')\right] \right\}$$
$$\text{s.t. } \; c = Y - k'$$
$$Y' = (1-\delta)k' + \theta'(k')^{\alpha}$$

Endogenous grid method: turbo speed

This yields the FOC

$$u'(c) = \beta E\left[V_Y(Y', \theta')(1 - \delta + \alpha\theta'(k')^{\alpha-1})\right]$$

Endogenous grid method: turbo speed

This yields the FOC

$$u'(c) = \beta E \left[V_Y(Y', \theta')(1 - \delta + \alpha\theta'(k')^{\alpha-1}) \right]$$

Y' is a simple function of k' (our grid points) so we can compute it, and the entire conditional expectation on the RHS, directly from the endogenous grid points

Endogenous grid method: turbo speed

$$u'(c) = \beta E\left[V_Y(Y', \theta')(1 - \delta + \alpha\theta'(k')^{\alpha-1})\right]$$

This allows us to compute c from the FOC

Endogenous grid method: turbo speed

$$u'(c) = \beta E\left[V_Y(Y', \theta')(1 - \delta + \alpha\theta'(k')^{\alpha-1})\right]$$

This allows us to compute c from the FOC

Then from c we can compute $Y = c + k'$ and then $V(Y, \theta)$ from the Bellman

Endogenous grid method: turbo speed

$$u'(c) = \beta E\left[V_Y(Y', \theta')(1 - \delta + \alpha\theta'(k')^{\alpha-1})\right]$$

This allows us to compute c from the FOC

Then from c we can compute $Y = c + k'$ and then $V(Y, \theta)$ from the Bellman

At no point did we need to use a numerical solver

Endogenous grid method: turbo speed

$$u'(c) = \beta E\left[V_Y(Y', \theta')(1 - \delta + \alpha\theta'(k')^{\alpha-1})\right]$$

This allows us to compute c from the FOC

Then from c we can compute $Y = c + k'$ and then $V(Y, \theta)$ from the Bellman

At no point did we need to use a numerical solver

Once we have converged on some $\hat{V}^*$ we then solve for k via $Y = (1 - \delta)k + \theta k^\alpha$ which does require a solver, but only once and after we have recovered our value function approximant

Endogenous grid method: practice

Let's solve our previous basic growth model using EGM

```
coefficients = zeros(params.num_basis);
coefficients[1:2] = [100 5];
```

Endogenous grid method: practice

```
function loop_grid_egm(params, capital_grid, coefficients)

    max_value = similar(capital_grid)
    capital_store = similar(capital_grid)

    for (iteration, capital_next) in enumerate(capital_grid)

        function bellman(consumption)
            cont_value = eval_value_function(coefficients, capital_next, params)
            value_out = (consumption)^(1-params.eta)/(1-params.eta) + params.beta*cont_value
            return value_out
        end;
        value_deriv = (eval_value_function(coefficients, capital_next + params.fin_diff, params) -
            eval_value_function(coefficients, capital_next - params.fin_diff, params))/(2params.fin_diff)
        consumption = (params.beta*value_deriv)^(-1/params.eta)
        max_value[iteration] = bellman(consumption)
        capital_store[iteration] = (capital_next + consumption)^(1/params.alpha)
    end

    grid = shrink_grid.(capital_store)
    basis_matrix = [cheb_polys.(grid, n) for n = 0:params.num_basis - 1];
    basis_matrix = hcat(basis_matrix...)
    return basis_matrix, capital_store, max_value

end
```

Endogenous grid method: practice

```julia
function solve_egm(params, capital_grid, coefficients)
    iteration = 1
    error = 1e10;
    max_value = -.0*ones(params.num_points);
    value_prev = .1*ones(params.num_points);
    coefficients_store = Vector{Vector}(undef, 1)
    coefficients_store[1] = coefficients
    while error > params.tolerance
        coefficients_prev = deepcopy(coefficients)
        current_poly, current_capital, max_value =
            loop_grid_egm(params, capital_grid, coefficients)
        coefficients = current_poly\max_value
        error = maximum(abs.((max_value - value_prev)./(value_prev)))
        value_prev = deepcopy(max_value)
        if mod(iteration, 5) == 0
            println("Maximum Error of $(error) on iteration $(iteration).")
            append!(coefficients_store, [coefficients])
        end
        iteration += 1
    end

    return coefficients, max_value, coefficients_store
end
```

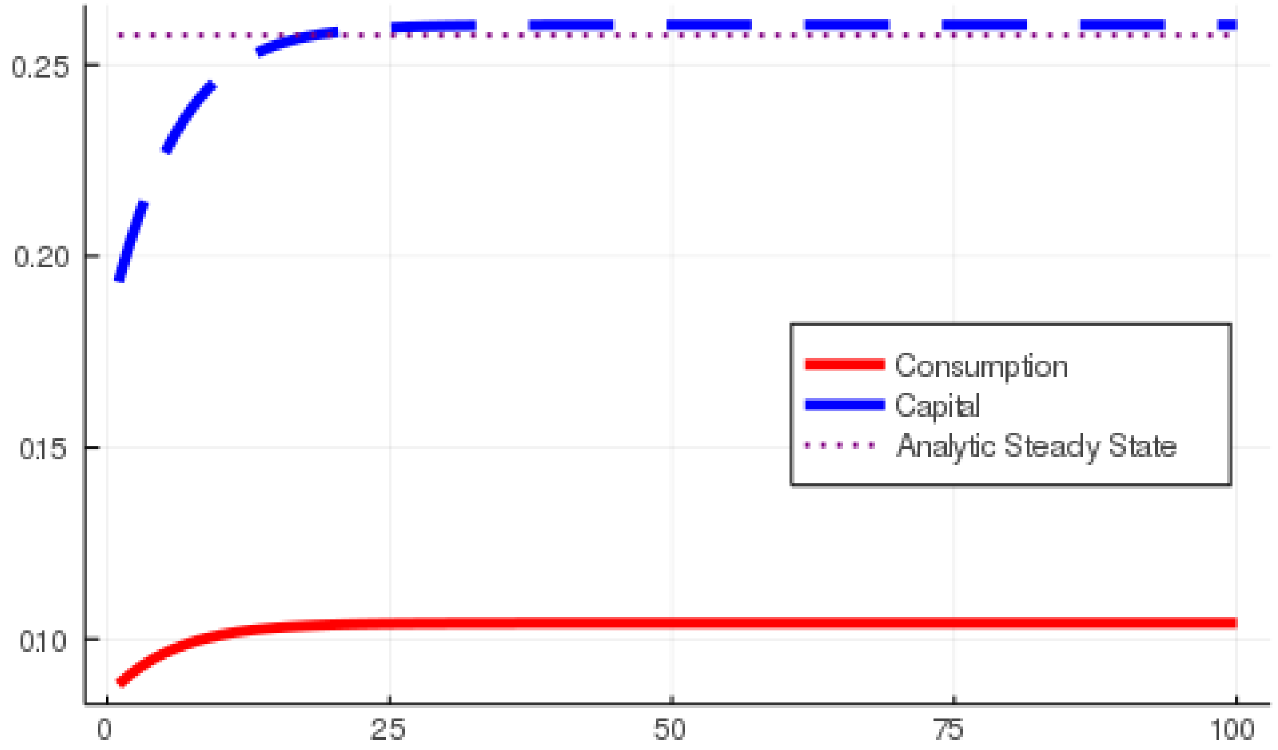
Endogenous grid method: practice

```
@time solution_coeffs, max_value, intermediate_coefficients = solve_egm(params, capital_grid, coefficients)
```

```
## Maximum Error of 0.33984199435067963 on iteration 5.
## Maximum Error of 10.28228426259625 on iteration 10.
## Maximum Error of 0.20342072898478147 on iteration 15.
## Maximum Error of 0.08237320856358903 on iteration 20.
## Maximum Error of 0.046523511041692764 on iteration 25.
## Maximum Error of 0.029761367850834913 on iteration 30.
## Maximum Error of 0.020301746199301834 on iteration 35.
## Maximum Error of 0.014389171942232297 on iteration 40.
## Maximum Error of 0.01045397323252693 on iteration 45.
## Maximum Error of 0.007723897658857289 on iteration 50.
## Maximum Error of 0.005774836832528466 on iteration 55.
## Maximum Error of 0.004354692320749045 on iteration 60.
## Maximum Error of 0.0033044754036150193 on iteration 65.
## Maximum Error of 0.002519276072894919 on iteration 70.
## Maximum Error of 0.0019273992634744547 on iteration 75.
## Maximum Error of 0.001478492015157928 on iteration 80.
## Maximum Error of 0.0011364271043177812 on iteration 85.
## Maximum Error of 0.0008748474878850943 on iteration 90.
## Maximum Error of 0.0006742714538897201 on iteration 95.
## Maximum Error of 0.0005201515925775415 on iteration 100.
## Maximum Error of 0.00040153842119404194 on iteration 105.
```

Endogenous grid method: practice

Endogenous grid method: practice



Envelope condition method

We can simplify rootfinding in an alternative way than an endogenous grid for infinite horizon problems

Envelope condition method

We can simplify rootfinding in an alternative way than an endogenous grid for infinite horizon problems

The idea here is that we want to use the envelope conditions instead of FOCs to construct policy functions

Envelope condition method

We can simplify rootfinding in an alternative way than an endogenous grid for infinite horizon problems

The idea here is that we want to use the envelope conditions instead of FOCs to construct policy functions

These will end up being easier to solve and sometimes we can solve them in closed form

Envelope condition method

For our old basic growth model problem (fully depreciating capital, no tech) the envelope condition (combined with the consumption FOC) is given by

$$V_k(k) = u'(c)f'(k)$$

Envelope condition method

For our old basic growth model problem (fully depreciating capital, no tech) the envelope condition (combined with the consumption FOC) is given by

$$V_k(k) = u'(c)f'(k)$$

Notice that the envelope condition is an intratemporal condition,
it only depends on time t variables

Envelope condition method

For our old basic growth model problem (fully depreciating capital, no tech) the envelope condition (combined with the consumption FOC) is given by

$$V_k(k) = u'(c)f'(k)$$

Notice that the envelope condition is an intratemporal condition,
it only depends on time t variables

We can use it to solve for c as a function of current variables

$$c = \left(\frac{V_k(k)}{\alpha k^{\alpha-1}}\right)^{-1/\eta}$$

Envelope condition method

For our old basic growth model problem (fully depreciating capital, no tech) the envelope condition (combined with the consumption FOC) is given by

$$V_k(k) = u'(c)f'(k)$$

Notice that the envelope condition is an intratemporal condition,
it only depends on time t variables

We can use it to solve for c as a function of current variables

$$c = \left(\frac{V_k(k)}{\alpha k^{\alpha-1}}\right)^{-1/\eta}$$

We can then recover k' from the budget constraint given our current state

Envelope condition method

For our old basic growth model problem (fully depreciating capital, no tech) the envelope condition (combined with the consumption FOC) is given by

$$V_k(k) = u'(c)f'(k)$$

Notice that the envelope condition is an intratemporal condition,
it only depends on time t variables

We can use it to solve for c as a function of current variables

$$c = \left(\frac{V_k(k)}{\alpha k^{\alpha-1}}\right)^{-1/\eta}$$

We can then recover k' from the budget constraint given our current state

We never need to use a solver at any point in time!

Envelope condition method

The algorithm is

1. Choose a grid $\{k_m\}_{m=1,\ldots,M}$ on which the value function is approximated
2. Solve for b and $\{c_m, k'_m\}$ such that
   - (inner loop) The quantities $\{c_m, k'_m\}$ solve the following given $V(k_m)$:
   - $V_k(k_m) = u'(c_m)f'(k_m)$,
   - $c_m + k'_m = f(k_m)$
   - (outer loop) The value function $\hat{V}(k;b)$ solves the following given $\{c_m, k_m\}$:
   - $\hat{V}(k_m;b) = u(c_m) + \beta \sum_{j=1}^{J} \omega_j \left[\hat{V}(k'_m;b)\right]$

Envelope condition method

In more complex settings (e.g. elastic labor supply) we will not necessarily be able to solve for policies without a solver

Envelope condition method

In more complex settings (e.g. elastic labor supply) we will not necessarily be able to solve for policies without a solver

However we will generally be able to solve a system of conditions via function iteration to recover the optimal controls as a function of current states and future states that are perfectly known at the current time

Envelope condition method

In more complex settings (e.g. elastic labor supply) we will not necessarily be able to solve for policies without a solver

However we will generally be able to solve a system of conditions via function iteration to recover the optimal controls as a function of current states and future states that are perfectly known at the current time

Thus at no point in time during the value function approximation algorithm do we need to interpolate off the grid or approximate expectations: this yields large speed and accuracy gains

Envelope condition method: practice

```julia
function loop_grid_ecm(params, basis_inverse, basis_matrix, grid, capital_grid, coefficients)

    max_value = similar(capital_grid);

    for (iteration, capital) in enumerate(capital_grid)

        function bellman(consumption)
            capital_next = capital^params.alpha - consumption
            cont_value = eval_value_function(coefficients, capital_next, params)
            value_out = (consumption)^(1-params.eta)/(1-params.eta) + params.beta*cont_value
            return value_out
        end;

        value_deriv = (eval_value_function(coefficients, capital + params.fin_diff, params) -
            eval_value_function(coefficients, capital - params.fin_diff, params))/(2params.fin_diff)
        consumption = (value_deriv/(params.alpha*capital^(params.alpha-1)))^(-1/params.eta)
        consumption = min(consumption, capital^params.alpha)
        max_value[iteration] = bellman(consumption)

    end

    return max_value
end
```

```
## loop_grid_ecm (generic function with 1 method)
```

Envelope condition method: practice

```
function solve_ecm(params, basis_inverse, basis_matrix, grid, capital_grid, coefficients)
    iteration = 1
    error = 1e10;
    max_value = similar(capital_grid);
    value_prev = .1*ones(params.num_points);
    coefficients_store = Vector{Vector}(undef, 1)
    coefficients_store[1] = coefficients
    while error > params.tolerance
        coefficients_prev = deepcopy(coefficients)
        max_value = loop_grid_ecm(params, basis_inverse, basis_matrix, grid, capital_grid, coefficients)
        coefficients = basis_inverse*max_value
        error = maximum(abs.((max_value - value_prev)./(value_prev)))
        value_prev = deepcopy(max_value)
        if mod(iteration, 5) == 0
            println("Maximum Error of $(error) on iteration $(iteration).")
            append!(coefficients_store, [coefficients])
        end
        iteration += 1
    end
    return coefficients, max_value, coefficients_store
end
```

```
## solve_ecm (generic function with 1 method)
```

Envelope condition method: practice

```
@time solution_coeffs, max_value, intermediate_coefficients =
    solve_ecm(params, basis_inverse, basis_matrix, grid, capital_grid, coefficients)
```

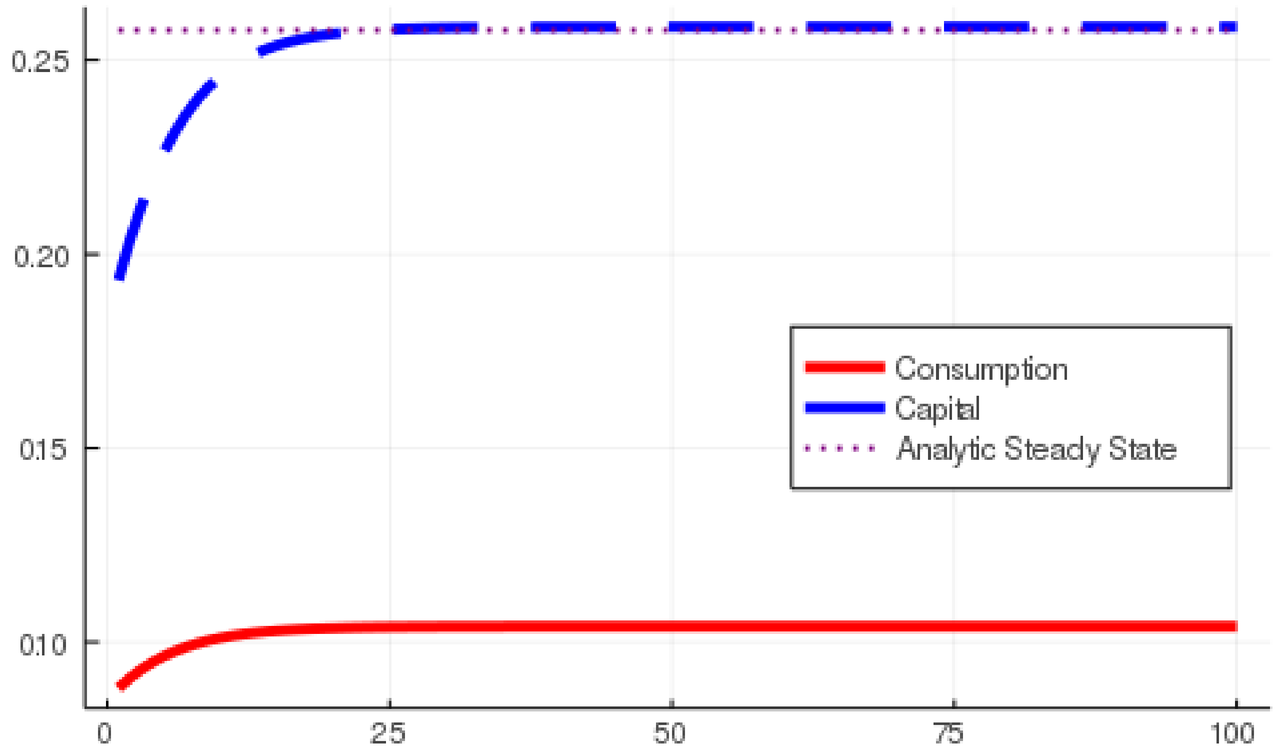
```
## Maximum Error of 0.35270640275290116 on iteration 5.
## Maximum Error of 8.805965931377644 on iteration 10.
## Maximum Error of 0.18965888767404476 on iteration 15.
## Maximum Error of 0.07943447581998832 on iteration 20.
## Maximum Error of 0.04532030555733413 on iteration 25.
## Maximum Error of 0.029127245675450643 on iteration 30.
## Maximum Error of 0.01992067971552436 on iteration 35.
## Maximum Error of 0.014141682933473048 on iteration 40.
## Maximum Error of 0.010285041765191065 on iteration 45.
## Maximum Error of 0.007604645927814628 on iteration 50.
## Maximum Error of 0.005688645480334269 on iteration 55.
## Maximum Error of 0.00429132835259861 on iteration 60.
## Maximum Error of 0.0032573086764123263 on iteration 65.
## Maximum Error of 0.0024838391761790304 on iteration 70.
## Maximum Error of 0.0019005891584954057 on iteration 75.
## Maximum Error of 0.0014581015347963073 on iteration 80.
## Maximum Error of 0.0011208568927165048 on iteration 85.
## Maximum Error of 0.0008629216689646948 on iteration 90.
## Maximum Error of 0.0006651156043508113 on iteration 95.
## Maximum Error of 0.0005131097078957133 on iteration 100.
```

Envelope condition method: practice

```julia
function simulate_model(params, solution_coeffs, time_horizon = 100)
    capital_store = zeros(time_horizon + 1)
    consumption_store = zeros(time_horizon)
    capital_store[1] = params.k_0

    for t = 1:time_horizon
        capital = capital_store[t]

        function bellman(consumption)
            capital_next = capital^params.alpha - consumption
            cont_value = eval_value_function(solution_coeffs, capital_next, params)
            value_out = (consumption)^(1-params.eta)/(1-params.eta) + params.beta*cont_value
            return -value_out
        end;

        results = optimize(bellman, 0.0, capital^params.alpha)
        consumption_store[t] = Optim.minimizer(results)
        capital_store[t+1] = capital^params.alpha - consumption_store[t]
    end

    return consumption_store, capital_store
end;
```

Envelope condition method: practice

```
time_horizon = 100;
consumption, capital = simulate_model(params, solution_coeffs, time_horizon);
plot(1:time_horizon, consumption, color = :red, linewidth = 4.0, label = "Consumption", legend = :right, size =
plot!(1:time_horizon, capital[1:end-1], color = :blue, linewidth = 4.0, linestyle = :dash, label = "Capital");
plot!(1:time_horizon, params.steady_state*ones(time_horizon), color = :purple, linewidth = 2.0, linestyle = :dot
```



Modified policy iteration

When doing VFI what is the most expensive part of the algorithm?

Modified policy iteration

When doing VFI what is the most expensive part of the algorithm?

The maximization step!

Modified policy iteration

When doing VFI what is the most expensive part of the algorithm?

The maximization step!

If we can reduce how often we need to maximize the Bellman we can significantly improve speed

Modified policy iteration

When doing VFI what is the most expensive part of the algorithm?

The maximization step!

If we can reduce how often we need to maximize the Bellman we can significantly improve speed

It turns out that between VFI iterations, the optimal policy does not change all that much

Modified policy iteration

When doing VFI what is the most expensive part of the algorithm?

The maximization step!

If we can reduce how often we need to maximize the Bellman we can significantly improve speed

It turns out that between VFI iterations, the optimal policy does not change all that much

This means that we may be able to skip the maximization step and re-use our old policy function to get new values for polynomial fitting

Modified policy iteration

When doing VFI what is the most expensive part of the algorithm?

The maximization step!

If we can reduce how often we need to maximize the Bellman we can significantly improve speed

It turns out that between VFI iterations, the optimal policy does not change all that much

This means that we may be able to skip the maximization step and re-use our old policy function to get new values for polynomial fitting

This is called **modified policy iteration**

Modified policy iteration

It only change step 5 of VFI:

1. While convergence criterion $>$ tolerance
    - Start iteration p
    - Solve the right hand side of the Bellman equation
    - Recover the maximized values, conditional on $\Gamma(k_{t+1}; b^{(p)})$
    - Fit the polynomial to the values and recover a new vector of coefficients $\hat{b}^{(p+1)}$.
    - Compute $b^{(p+1)} = (1-\gamma)b^{(p)} + \gamma\hat{b}^{(p+1)}$ where $\gamma \in (0,1)$.
    - While MPI stop criterion $>$ tolerance
        - Use policies from last VFI iteration to re-fit the polynomial (no maximizing!)
        - Compute $b^{(p+1)}$ for iteration $p+1$ by $b^{(p+1)} = (1-\gamma)b^{(p)} + \gamma\hat{b}^{(p+1)}$ where $\gamma \in (0,1)$.

Modified policy iteration

Stop criteron can be a few things:

1. Fixed number of iterations
2. Stop when change in value function is sufficient small, QuantEcon suggests stopping MPI when

$$\max(V_p(x;c) - V_{p-1}(x;c)) - \min(V_p(x;c) - V_{p-1}(x;c)) < \epsilon(1-\beta)\beta$$

   where the max and min are over the values on the grid

Modified policy iteration

Stop criteron can be a few things:

1. Fixed number of iterations
2. Stop when change in value function is sufficient small, QuantEcon suggests stopping MPI when

$$\max(V_p(x;c) - V_{p-1}(x;c)) - \min(V_p(x;c) - V_{p-1}(x;c)) < \epsilon(1-\beta)\beta$$

   where the max and min are over the values on the grid

Also note: you should probably start MPI after a few VFI iterations unless you have a good initial guess

Modified policy iteration

Stop criteron can be a few things:

1. Fixed number of iterations
2. Stop when change in value function is sufficient small, QuantEcon suggests stopping MPI when

$$\max(V_p(x;c) - V_{p-1}(x;c)) - \min(V_p(x;c) - V_{p-1}(x;c)) < \epsilon(1-\beta)\beta$$

   where the max and min are over the values on the grid

Also note: you should probably start MPI after a few VFI iterations unless you have a good initial guess

If your early policy functions are bad then starting MPI too early will blow up your problem

Modified policy iteration

```julia
function solve_vfi_regress_mpi(params, basis_inverse, basis_matrix, grid, capital_grid, coefficients)
    max_value = -.0*ones(params.num_points);
    error = 1e10;
    value_prev = .1*ones(params.num_points);
    value_prev_outer = .1*ones(params.num_points);
    coefficients_store = Vector{Vector}(undef, 1)
    coefficients_store[1] = coefficients
    iteration = 1
    while error > params.tolerance
        max_value, consumption_store =
            loop_grid_regress(params, basis_inverse, basis_matrix, grid, capital_grid, coefficients)
        coefficients = basis_inverse*max_value
        if iteration > params.mpi_start # modified policy iteration loop
            mpi_iteration = 1
            while maximum(abs.(max_value - value_prev)) -
                    minimum(abs.(max_value - value_prev)) >
                    (1 - params.beta)/params.beta*params.tolerance
                value_prev = deepcopy(max_value)
```

Modified policy iteration

```
     function bellman(consumption, capital)
        capital_next = capital^params.alpha - consumption
        cont_value = eval_value_function(coefficients, capital_next, params)
        value_out = (consumption)^(1-params.eta)/(1-params.eta) + params.beta*cont_value
        return value_out
      end
      max_value = bellman.(consumption_store, capital_grid) # greedy policy
      coefficients = basis_inverse*max_value
      if mod(mpi_iteration, 5) == 0
        println("MPI iteration $mpi_iteration on VFI iteration $iteration.")
      end
      mpi_iteration += 1
    end
  end
  error = maximum(abs.((max_value .- value_prev_outer)./(value_prev_outer)))
  value_prev_outer = deepcopy(max_value)

  if mod(iteration, 5) == 0
    println("Maximum Error of $(error) on iteration $(iteration).")
    append!(coefficients_store, [coefficients])
  end
  iteration += 1
end
return coefficients, max_value, coefficients_store
```

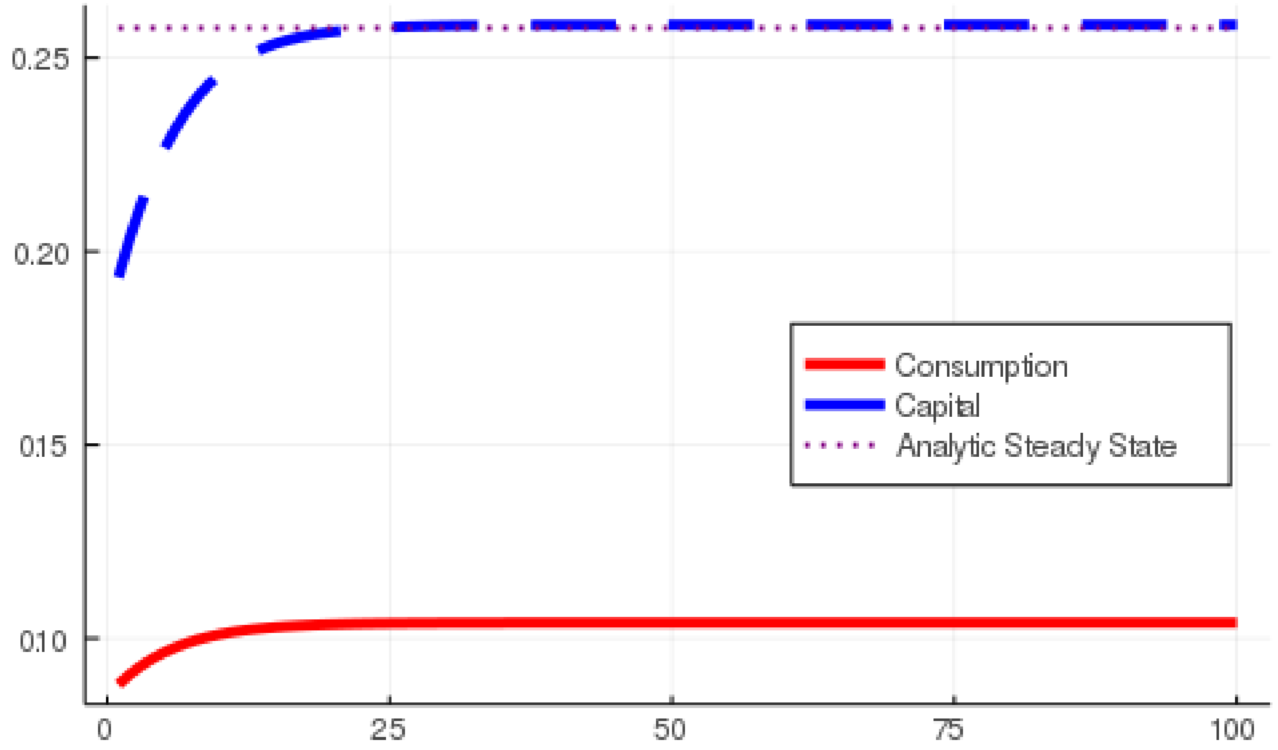
Modified policy iteration

Modified policy iteration

```
@time solution_coeffs, max_value, intermediate_coefficients =
    solve_vfi_regress_mpi(params, basis_inverse, basis_matrix, grid, capital_grid, coefficients)
```

```
## Maximum Error of 0.33656462321563774 on iteration 5.
## MPI iteration 5 on VFI iteration 6.
## MPI iteration 10 on VFI iteration 6.
## MPI iteration 15 on VFI iteration 6.
## MPI iteration 20 on VFI iteration 6.
## MPI iteration 25 on VFI iteration 6.
## MPI iteration 30 on VFI iteration 6.
## MPI iteration 35 on VFI iteration 6.
## MPI iteration 40 on VFI iteration 6.
## MPI iteration 45 on VFI iteration 6.
## MPI iteration 50 on VFI iteration 6.
## MPI iteration 55 on VFI iteration 6.
## MPI iteration 60 on VFI iteration 6.
## MPI iteration 65 on VFI iteration 6.
## MPI iteration 70 on VFI iteration 6.
## MPI iteration 5 on VFI iteration 7.
## MPI iteration 10 on VFI iteration 7.
## MPI iteration 15 on VFI iteration 7.
## MPI iteration 20 on VFI iteration 7.
## MPI iteration 25 on VFI iteration 7.
```

Modified policy iteration



Modified policy iteration

What was your speed up?

I got **6 times**: 0.6s $\rightarrow$ 0.1s