# 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 $\Psi$, 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 $\Psi$, but instead of being $n \times n$ it is $m \times n$

Finally we use the standard least-squares equation

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

Where $(\Psi'\Psi)^{-1}\Psi'$ is 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 $\Psi$, but instead of being $n \times n$ it is $m \times n$

Finally we use the standard least-squares equation

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

Where $(\Psi'\Psi)^{-1}\Psi'$ is the Moore-Penrose matrix inverse

We can apply Chebyshev regression to even our regular tensor approaches,

# Chebyshev regression: practice

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

```
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/
                num_basis = 7, num_points = 15, tolerance = 1e-6, 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
```

```
## 15-element Vector{Float64}:
##   0.9945218953682733
##   0.9510565162951535
##   0.8660254037844387
##   0.7431448254773942
##   0.5877852522924731
##   0.4067366430758004
##   0.20791169081775945
##   2.83276944882399e-16
##  -0.20791169081775912
##  -0.4067366430758001
##  -0.587785252292473
##  -0.7431448254773941
```

# Chebyshev regression: practice

Make the inverse function to shrink from capital to Chebyshev space

```
shrink_grid(capital)
```

# Chebyshev regression: practice

Make the inverse function to shrink from capital to Chebyshev space

`shrink_grid(capital)`

```
shrink_grid(capital) =
  2*(capital - params.capital_lower)/(params.capital_upper - params.capital_lower) - 1;
```

`shrink_grid` will inherit `params` from wrapper functions

# 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;
```

# Chebyshev regression: practice

```
construct_basis_matrix(grid, params) = hcat([cheb_polys.(shrink_grid.(grid), n) for n = 0:params
basis_matrix = construct_basis_matrix(capital_grid, params);
basis_inverse = inv(basis_matrix'*basis_matrix)*(basis_matrix') # pre-compute pseudoinverse for
```

```
## 7×15 Matrix{Float64}:
##  0.0666667   0.0666667     0.0666667    …   0.0666667     0.0666667
##  0.132603    0.126808      0.11547          -0.126808     -0.132603
##  0.13042     0.107869      0.0666667         0.107869      0.13042
##  0.126808    0.0783714     4.46986e-16      -0.0783714    -0.126808
##  0.121806    0.0412023    -0.0666667         0.0412023     0.121806
##  0.11547    -8.47892e-17  -0.11547      …  -1.85126e-16  -0.11547
##  0.107869   -0.0412023    -0.133333         -0.0412023     0.107869
```

# Chebyshev regression: practice

```
eval_value_function(coefficients, grid, params) = construct_basis_matrix(grid, params) * coeffi
```

# Chebyshev regression: practice

```
function loop_grid_regress(params, 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)[1]
            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

```
function solve_vfi_regress(params, basis_inverse, 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, 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, 25) == 0
            println("Maximum Error of $(error) on iteration $(iteration).")
            append!(coefficients_store, [coefficients])
        end
        iteration += 1
    end

    return coefficients, max_value, coefficients_store
```

# Chebyshev regression: practice

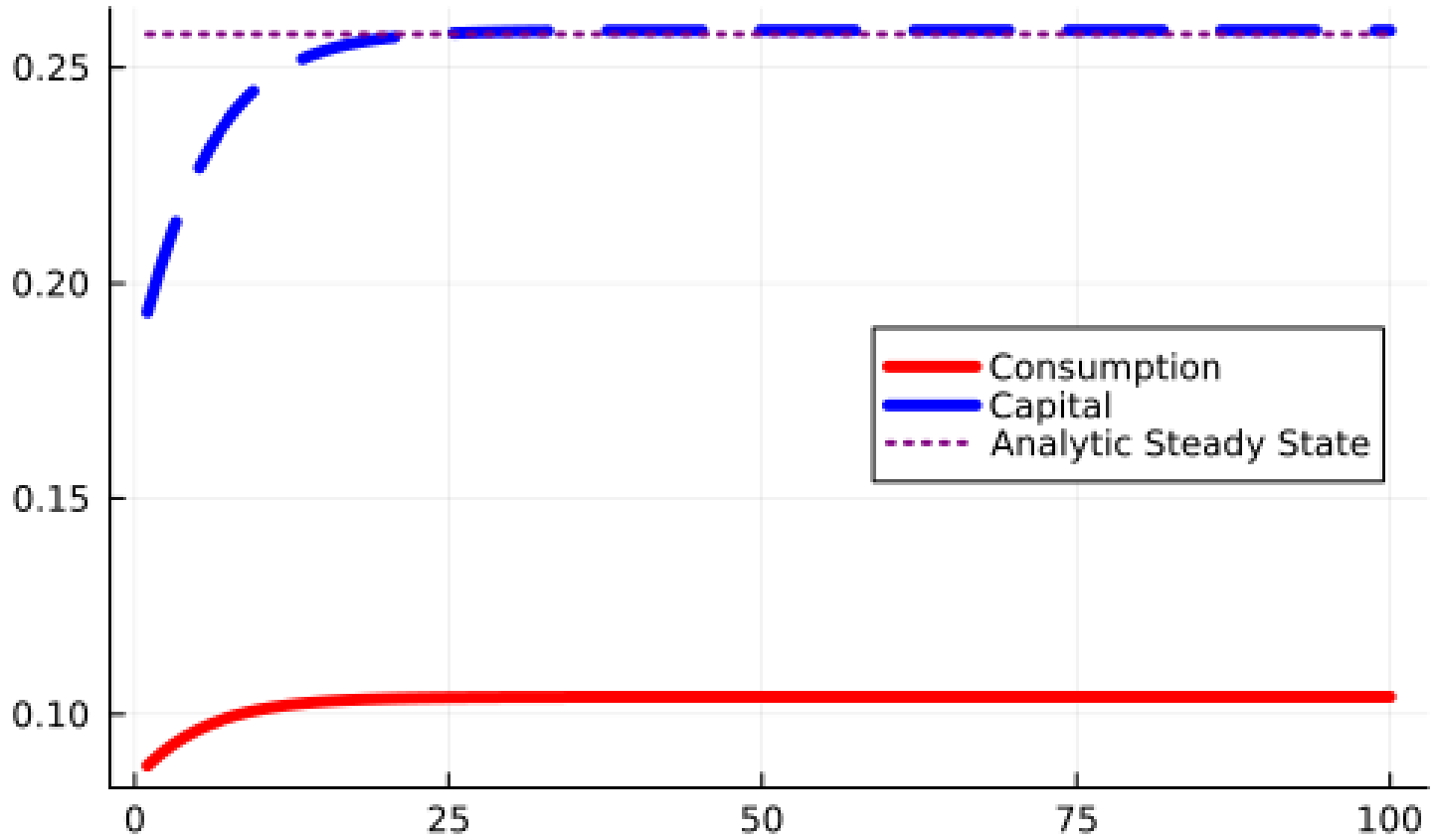
```
@time solution_coeffs, max_value, intermediate_coefficients =
    solve_vfi_regress(params, basis_inverse, capital_grid, coefficients)
```

```
## Maximum Error of 0.045607916784149 23 on iteration 25.
## Maximum Error of 0.007635436575597669 on iteration 50.
## Maximum Error of 0.0019075236037348097 on iteration 75.
## Maximum Error of 0.0005149316099488123 on iteration 100.
## Maximum Error of 0.00014178153569906261 on iteration 125.
## Maximum Error of 3.92482976918936e-5 on iteration 150.
## Maximum Error of 1.0880896630296866e-5 on iteration 175.
## Maximum Error of 3.0177727176252443e-6 on iteration 200.
##   0.847524 seconds (22.49 M allocations: 515.604 MiB, 5.23% gc time, 0.25% compilation time)

## ([-194.85536958622183, 14.142104524187651, -2.664424683176605, 0.5749549884000286, -0.13337251156715
```

# Chebyshev regression: practice

```
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)[1]
            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

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

# 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

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

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 $\epsilon_j$ and weights $\omega_j$, $j = 1, \dots, J$ 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,

# 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)[1]
            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)[1])/(2para
        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...)
```

# Endogenous grid method: practice

```
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, 25) == 0
            println("Maximum Error of $(error) on iteration $(iteration).")
            append!(coefficients_store, [coefficients])
        end
        iteration += 1
    end
```

# Endogenous grid method: practice

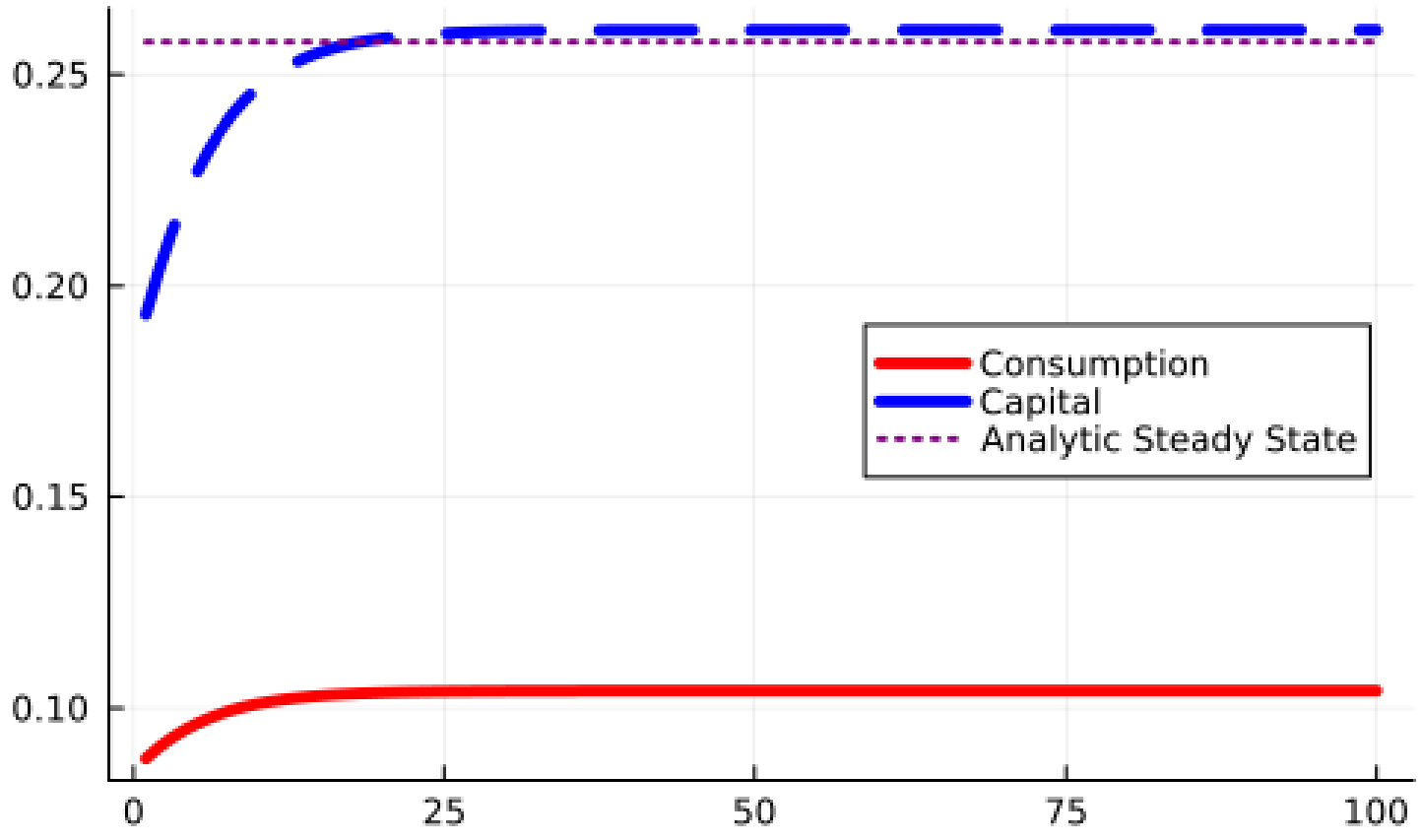
```
@time solution_coeffs, max_value, intermediate_coefficients = solve_egm(params, capital_grid, co
```

```
## Maximum Error of 0.04656312802519048 on iteration 25.
## Maximum Error of 0.0077279558439712175 on iteration 50.
## Maximum Error of 0.0019283105651684272 on iteration 75.
## Maximum Error of 0.0005203908554999235 on iteration 100.
## Maximum Error of 0.00014327347852089944 on iteration 125.
## Maximum Error of 3.966043939472439e-5 on iteration 150.
## Maximum Error of 1.0995091700787616e-5 on iteration 175.
## Maximum Error of 3.0494442960141223e-6 on iteration 200.
##   0.229561 seconds (5.69 M allocations: 144.159 MiB, 9.64% gc time, 0.76% compilation time)

## ([-194.86588167567055, 14.166854450284145, -2.659830643535021, 0.5619970720353987, -0.13632318626428
```

# Endogenous grid method: practice

```
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)[1]
            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;
```

# 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

# Envelope condition method

The algorithm is

1. Choose a grid $\{k_m\}_{m=1,\dots,M}$ 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

```
function loop_grid_ecm(params, 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)[1]
            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)[1] -
            eval_value_function(coefficients, capital - params.fin_diff, params)[1])/(2params.f
        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
```

# Envelope condition method: practice

```
function solve_ecm(params, basis_inverse, 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, 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, 25) == 0
            println("Maximum Error of $(error) on iteration $(iteration).")
            append!(coefficients_store, [coefficients])
        end
        iteration += 1
    end
    return coefficients, max_value, coefficients_store
end
```

# Envelope condition method: practice

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

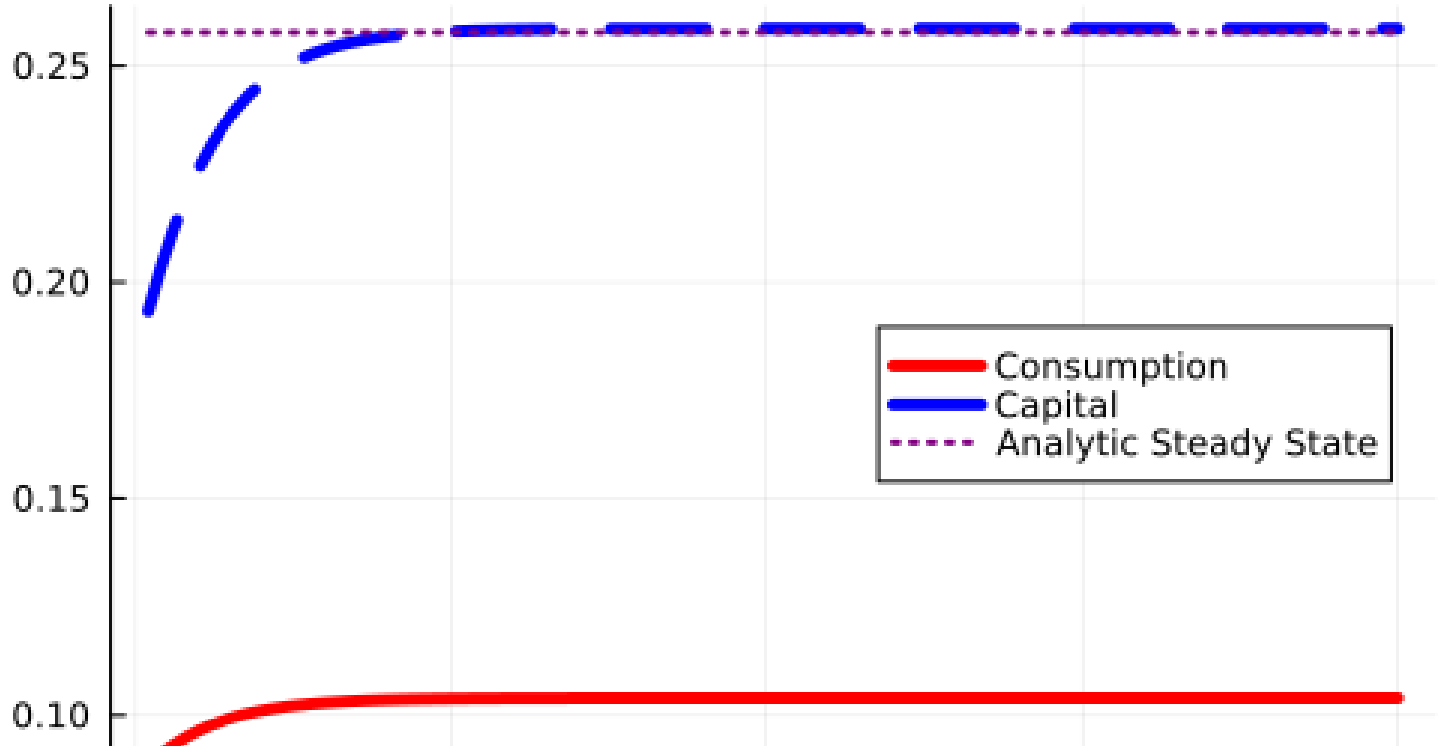
```
## Maximum Error of 0.0453525403650495 on iteration 25.
## Maximum Error of 0.0076079815408730215 on iteration 50.
## Maximum Error of 0.0019013403845842475 on iteration 75.
## Maximum Error of 0.0005133070957501089 on iteration 100.
## Maximum Error of 0.00014133753500579239 on iteration 125.
## Maximum Error of 3.912563883912878e-5 on iteration 150.
## Maximum Error of 1.0846910981672597e-5 on iteration 175.
## Maximum Error of 3.0083484348532563e-6 on iteration 200.
##   0.174326 seconds (4.53 M allocations: 106.518 MiB, 5.97% gc time, 0.99% compilation time)

## ([-194.85531932176127, 14.142062593106905, -2.6644837015279976, 0.5749531960546624, -0.1333743010189
```

# Envelope condition method: practice

```
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)[1]
            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 =
plot!(1:time_horizon, capital[1:end-1], color = :blue, linewidth = 4.0, linestyle = :dash, labe
plot!(1:time_horizon, params.steady_state*ones(time_horizon), color = :purple, linewidth = 2.0,
```



# 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 changes step 5 of VFI:

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 new 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!)

# 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

Only 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

```
function solve_vfi_regress_mpi(params, basis_inverse, basis_matrix, grid, capital_grid, coeffic
    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, 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)[1]
                value_out = (consumption)^(1-params.eta)/(1-params.eta) + params.beta*cont_v
                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
```

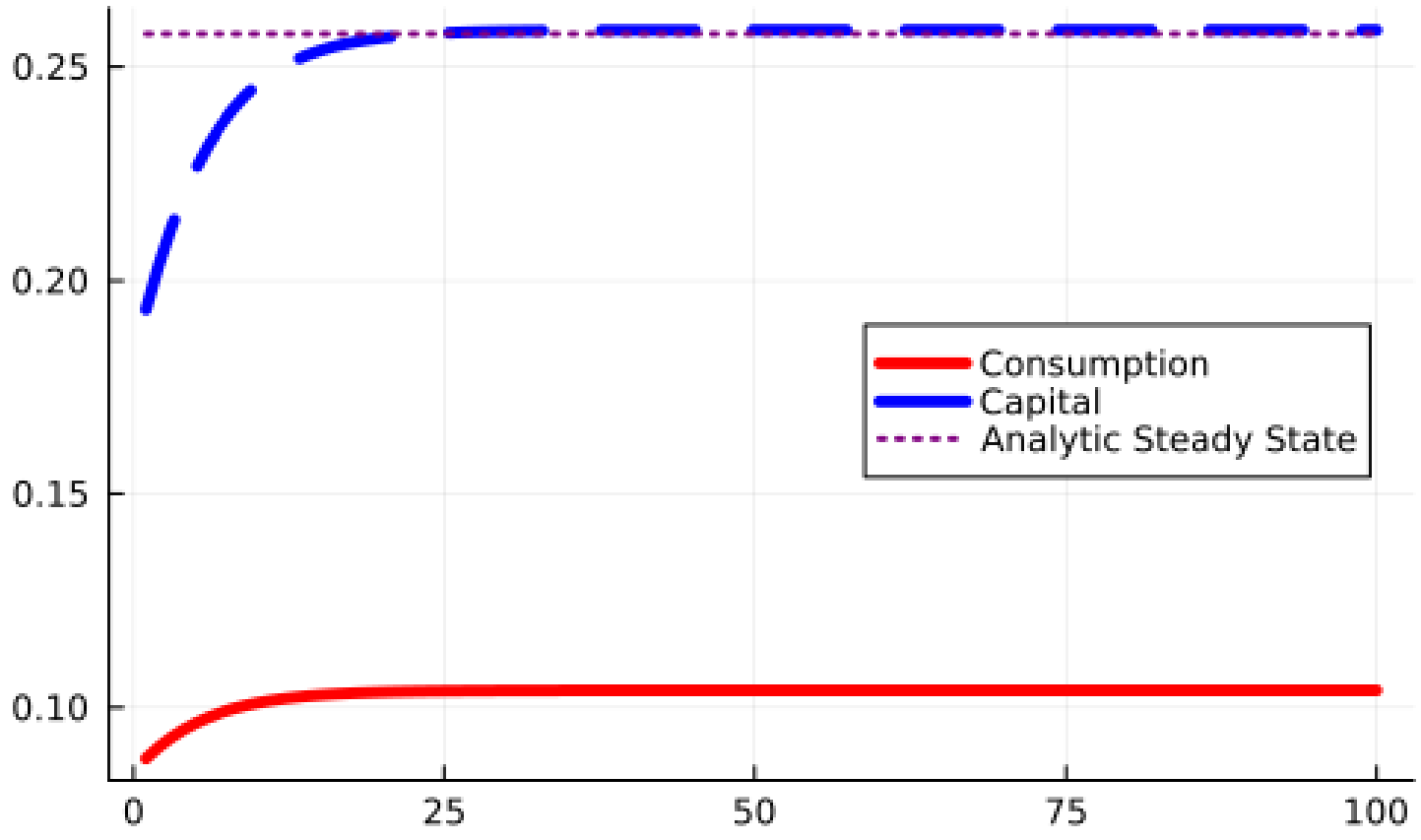
# Modified policy iteration

# 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 0.33871304913135464 on iteration 5.
## MPI iteration 25 on VFI iteration 6.
## MPI iteration 50 on VFI iteration 6.
## MPI iteration 75 on VFI iteration 6.
## MPI iteration 25 on VFI iteration 7.
## MPI iteration 50 on VFI iteration 7.
## MPI iteration 25 on VFI iteration 8.
## Maximum Error of 1.141822859504868e-5 on iteration 10.
## Maximum Error of 4.0892905314530945e-6 on iteration 15.
## Maximum Error of 3.0060755201005212e-6 on iteration 20.
## Maximum Error of 2.3260768798925272e-6 on iteration 25.
## Maximum Error of 1.7998932693973723e-6 on iteration 30.
## Maximum Error of 1.3927345521584257e-6 on iteration 35.
## Maximum Error of 1.0776782693136323e-6 on iteration 40.
##   0.223322 seconds (5.89 M allocations: 136.588 MiB, 4.52% gc time, 0.88% compilation time)

## ([-194.8621441678187, 14.1421045241982, -2.6644246831782934, 0.5749549884003013, -0.13337251156711613
```

# Modified policy iteration



# Solve times

```
## The solve times are:

## Regression: 0.867 seconds

## Endogenous Grid + Regression: 0.216 seconds

## Envelope Condition + Regression: 0.172 seconds

## Modified Policy Iteration + Regression: 0.257 seconds
```

## Individually they give 3-6 times speed up