Lecture 7

Solution methods for discrete time dynamic models

Ivan Rudik
AEM 7130

Roadmap

1. Intuition for solving dynamic models
2. Value function iteration
3. Fixed point iteration
4. Time iteration
5. VFI + discretization

Things to do

1. Install: `LinearAlgebra, Optim, Plots, Roots`

Things to do

1. Install: `LinearAlgebra, Optim, Plots, Roots`
2. Keep in mind that for VFI and TI we will be using optimization/rootfinding packages
   - This matters because these packages typically only let the functions they work on have one input: the guesses for the maximizing input or root
   - We get around this by expressing the function as a *closure*
   - i.e. declare the function inside of a wrapper function that does the maximization/rootfinding so it can access the parameters in the wrapper function

Things to do

1. Keep in mind we will be working with about the simplest example possible, more complex problems will be more difficult to solve in many ways

How do we solve dynamic models?

Solutions to economic models

How do we solve economic models?

Solutions to economic models

How do we solve economic models?

First, what do we want?

Solutions to economic models

How do we solve economic models?

First, what do we want?

We want to be able to compute things like optimal policy trajectories, welfare, etc

Solutions to economic models

There are generally two objects that can deliver what we want:

1. Value functions
2. Policy functions

Solutions to economic models

There are generally two objects that can deliver what we want:

1. Value functions
2. Policy functions

The idea behind the most commonly used solution concepts is to recover good approximations to one of these two functions

Solutions to economic models

We recover these functions by exploiting two things:

1. Dynamic equilibrium conditions incorporating these functions
    - Bellman
    - Euler
2. Fixed points

Our general example

Consider the following problem we will be using for all the solution methods:

$$\max_{\{c_t\}_{t=0}^{\infty}} \sum_{t=1}^{\infty} \beta^t u(c_t)$$
$$\text{subject to:} \quad k_{t+1} = f(k_t) - c_t$$

where both consumption and time $t+1$ capital are positive,
$k(0) = k_0$, $\alpha > 0$, and $\beta \in (0, 1)$

Our general example

Represent the growth model as a Bellman equation

$$V(k_t) = \max_{c_t} u(c_t) + \beta V(k_{t+1})$$
$$\text{subject to:} \quad k'_{t+1} = f(k_t) - c_t$$

Our general example

Represent the growth model as a Bellman equation

$$V(k_t) = \max_{c_t} u(c_t) + \beta V(k_{t+1})$$
$$\text{subject to:} \quad k'_{t+1} = f(k_t) - c_t$$

We can then express the value function in terms of itself, the current state, and the current consumption choice:

$$V(k_t) = \max_{c_t} u(c_t) + \beta V(f(k_t) - c_t)$$

Our general example

$$V(k_t) = \max_{c_t} u(c_t) + \beta V(f(k_t) - c_t)$$

How do we solve this?

Main idea:

1. Guess $V(k_t)$
2. Given guess, do the maximization on the right hand side at some set of states $\mathbf{k_t^i}$
3. Maximized right hand side gives us new values of $V(k_t)$
4. Repeat

Our general example

Another equilibrium condition is the **Euler equation**

Our general example

Another equilibrium condition is the **Euler equation**

For our problem it is

$$u'(c_t) = \beta u'(c_{t+1}) f'(k_{t+1})$$

Plug in the policy function $c_t = C(k_t)$:

$$u'(C(k_t)) = \beta u'(C(k_{t+1})) f'(k_{t+1})$$

Our general example

Recognize k_{t+1} is a function of the current policy and state $(C(k_t), k_t)$:

$$k_{t+1} = f(k_t) - C(k_t)$$

Use this to express the Euler equation in terms of k_t and C

$$C(k_t) = u'^{(-1)} \left[\beta u' \left[C(k_{t+1}(C(k_t), k_t))\right] f' \left[k_{t+1}(C(k_t), k_t)\right]\right]$$

Our general example

$$C(k_t) = u'^{(-1)}\left[\beta u'\left[C(k_{t+1}(C(k_t), k_t))\right] f'\left[k_{t+1}(C(k_t), k_t)\right]\right]$$

How do we solve this?

Main idea:

1. Guess $C(k_t)$
2. Given guess, evaluate the right hand side at some set of states $\mathbf{k_t^i}$
3. Evaluated right hand side gives us new values of $C(k_t)$
4. Repeat

Value function iteration

Method 1: Value function iteration

In VFI we approximate the **value function** with some flexible functional form $\Gamma(k_t; b)$ where b is a vector of coefficients

The algorithm has 6 steps

Method 1: Value function iteration

Step 1: Select the number of collocation points in each dimension and the domain of the approximation space

Method 1: Value function iteration

Step 1: Select the number of collocation points in each dimension and the domain of the approximation space

Step 2: Select an initial vector of coefficients b_0 with the same number of elements as the collocation grid, and initial guesses for consumption for the solver

Method 1: Value function iteration

Step 1: Select the number of collocation points in each dimension and the domain of the approximation space

Step 2: Select an initial vector of coefficients b_0 with the same number of elements as the collocation grid, and initial guesses for consumption for the solver

Step 3: Select a rule for convergence

Method 1: Value function iteration

Step 1: Select the number of collocation points in each dimension and the domain of the approximation space

Step 2: Select an initial vector of coefficients b_0 with the same number of elements as the collocation grid, and initial guesses for consumption for the solver

Step 3: Select a rule for convergence

Step 4: Construct the grid and basis matrix

Method 1: Value function iteration

Step 5: While convergence criterion $>$ tolerance **(outer loop [fixed point])**

- Start iteration p
- For each grid point **(inner loop [right hand side maximization])**
  - Maximize the right hand side of the Bellman equation at each grid point using the approximating value function $\Gamma(k_{t+1}; b^{(p)})$ in place of $V(k_{t+1})$
  - Recover the maximized values $V^{(p)}$ at each grid point, conditional on $\Gamma(k_{t+1}; b^{(p)})$

...

Method 1: Value function iteration

Step 5: While convergence criterion $>$ tolerance **(outer loop, continued)**

- Fit the polynomial to the maximized values $V^{(p)}$ and recover a new vector of coefficients $\hat{b}^{(p+1)}$.
- Compute the vector of coefficients $b^{(p+1)}$ for iteration $p+1$ by $b^{(p+1)} = (1-\gamma)b^{(p)} + \gamma\hat{b}^{(p+1)}$ where $\gamma \in (0,1)$. (damping)

Method 1: Value function iteration

Step 5: While convergence criterion $>$ tolerance **(outer loop, continued)**

- Fit the polynomial to the maximized values $V^{(p)}$ and recover a new vector of coefficients $\hat{b}^{(p+1)}$.
- Compute the vector of coefficients $b^{(p+1)}$ for iteration $p+1$ by $b^{(p+1)} = (1-\gamma)b^{(p)} + \gamma\hat{b}^{(p+1)}$ where $\gamma \in (0,1)$. (damping)

Step 6: Error check your approximation

Functions we will code up

We will need to code up six key functions for all of the algorithms

- `cheb_nodes(n)`: construct degree n collocation grid
- `cheb_polys(x, n)`: evaluate degree n Chebyshev polynomials
- `construct_basis_matrix(grid, params)`: construct full $n \times n$ Chebyshev basis matrix
- `eval_approx_function(coefficients, grid, params)`: evaluate approximating function at grid points `grid`
- `loop_grid(params, capital_grid, coefficients)`: loop over the collocation grid
- `solve_algorithm(params, basis_inverse, capital_grid, coefficients)`: iterate on the fixed point

Functional forms and parameters

Functional forms

- $u(c_t) = c_t^{1-\eta}/(1-\eta)$
- $f(k_t) = k_t^{\alpha}$

Parameters

- $\alpha = 0.75$
- $\beta = 0.95$
- $\eta = 2$

Initial capital value: $k_0 = (\alpha\beta)^{1/(1-\alpha)}/2$

Step 1: Select the number of points and domain

If $k_0 = (\alpha\beta)^{1/(1-\alpha)}/2$ what are a logical set of bounds for the capital state?

Step 1: Select the number of points and domain

If $k_0 = (\alpha\beta)^{1/(1-\alpha)}/2$ what are a logical set of bounds for the capital state?

k^0 and the steady state level $(\alpha\beta)^{1/(1-\alpha)}$

Step 1: Select the number of points and domain

Put everything in a **named tuple** to make passing things easier

```
using LinearAlgebra
using Optim
using Plots
params = (alpha = 0.75, # capital share
          beta = 0.95, # discount
          eta = 2, # EMUC
          steady_state = (0.75*0.95)^(1/(1 - 0.75)),
          k_0 = (0.75*0.95)^(1/(1 - 0.75))/2, # initial state
          capital_upper = (0.75*0.95)^(1/(1 - 0.75))*1.01, # upper bound
          capital_lower = (0.75*0.95)^(1/(1 - 0.75))/2, # lower bound
          num_points = 7, # number of grid points
          tolerance = 0.0001)
```

```
## (alpha = 0.75, beta = 0.95, eta = 2, steady_state = 0.25771486816406236, k_0 = 0.12885743408203118,
```

Step 2: Select an initial vector of coefficients b_0

In some cases you might have a good guess (e.g. increasing and concave so you know the second value is positive, third value is negative, rest maybe set to zero)

Step 2: Select an initial vector of coefficients b_0

In some cases you might have a good guess (e.g. increasing and concave so you know the second value is positive, third value is negative, rest maybe set to zero)

Other cases you might not, guessing zeros effectively turns the initial iteration into a static problem, the second iteration into a 2 period problem, and so on

Step 2: Select an initial vector of coefficients b_0

In some cases you might have a good guess (e.g. increasing and concave so you know the second value is positive, third value is negative, rest maybe set to zero)

Other cases you might not, guessing zeros effectively turns the initial iteration into a static problem, the second iteration into a 2 period problem, and so on

```
coefficients = zeros(params.num_points) # # coeffs = # grid points in collocation
```

```
## 7-element Vector{Float64}:
##  0.0
##  0.0
##  0.0
##  0.0
##  0.0
##  0.0
##  0.0
```

Step 3: Select a convergence rule

There's a lot of potential options here to determine convergence of the function

Step 3: Select a convergence rule

There's a lot of potential options here to determine convergence of the function

Relative or absolute change? Or both?

Step 3: Select a convergence rule

There's a lot of potential options here to determine convergence of the function

Relative or absolute change? Or both?

Change in the value function? Change in the policy function?

Step 3: Select a convergence rule

There's a lot of potential options here to determine convergence of the function

Relative or absolute change? Or both?

Change in the value function? Change in the policy function?

Which norm?

Step 3: Select a convergence rule

There's a lot of potential options here to determine convergence of the function

Relative or absolute change? Or both?

Change in the value function? Change in the policy function?

Which norm?

Our rule for class: convergence is when the maximum relative change in value on the grid is < 0.001%

Step 4: Construct the grid and basis matrix

The function `cheb_nodes` from last lecture constructs the grid on $[-1, 1]$

Step 4: Construct the grid and basis matrix

The function `cheb_nodes` from last lecture constructs the grid on $[-1, 1]$

$$x_k = cos\left(\frac{2k-1}{2n}\pi\right),\ k = 1, \ldots, n$$

Step 4: Construct the grid and basis matrix

```
cheb_nodes(n) = cos.(pi * (2*(1:n) .- 1)./(2n));
grid = cheb_nodes(params.num_points) # [-1, 1] grid with n points
```

```
## 7-element Vector{Float64}:
##   0.9749279121818236
##   0.7818314824680298
##   0.4338837391175582
##   6.123233995736766e-17
##  -0.43388373911755806
##  -0.7818314824680297
##  -0.9749279121818236
```

Our actual capital domain isn't on $[-1, 1]$, we need to expand the grid to some arbitrary $[a, b]$ using a function `expand_grid(grid, params)`

Step 4: Construct the grid and basis matrix

```
expand_grid(grid, params) = # function that expands [-1,1] to [a,b]
   (1 .+ grid)*(params.capital_upper - params.capital_lower)/2 .+ params.capital_lower
```

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

```
capital_grid = expand_grid(grid, params)
```

```
## 7-element Vector{Float64}:
##  0.2586443471450049
##  0.2459545728087113
##  0.2230883895732961
##  0.19457472546386706
##  0.16606106135443804
##  0.14319487811902284
##  0.13050510378272925
```

Step 4: Construct the grid and basis matrix

Make the inverse function to shrink from capital to Chebyshev space

```
shrink_grid(capital)
```

Step 4: Construct the grid and basis matrix

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.(capital_grid)
```

```
## 7-element Vector{Float64}:
##   0.9749279121818237
##   0.7818314824680297
##   0.43388373911755806
##  -2.220446049250313e-16
##  -0.43388373911755806
##  -0.7818314824680297
##  -0.9749279121818236
```

`shrink_grid` will inherit `params` from wrapper functions

Step 4: Construct the grid and basis matrix

Step 4: Construct the grid and basis matrix

`cheb_polys(x, n)` from last lecture gives us the nth degree Chebyshev polynomial at point x

Step 4: Construct the grid and basis matrix

`cheb_polys(x, n)` from last lecture gives us the nth degree Chebyshev polynomial at point x

```
# 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;
```

Step 4: Construct the grid and basis matrix

`cheb_polys.(grid, n)` gives us the nth degree Chebyshev polynomial at all points on our grid

Step 4: Construct the grid and basis matrix

`cheb_polys.(grid, n)` gives us the nth degree Chebyshev polynomial at all points on our grid

```
cheb_polys.(grid, 2) # 2nd degree Cheb poly at each grid point
```

```
## 7-element Vector{Float64}:
##   0.9009688679024193
##   0.22252093395631434
##  -0.6234898018587334
##  -1.0
##  -0.6234898018587336
##   0.22252093395631412
##   0.9009688679024193
```

Step 4: Construct the grid and basis matrix

In our basis matrix, rows are grid points, columns are basis functions, make a function `construct_basis_matrix(grid, params)` that makes the basis matrix for some arbitrary grid of points

```
construct_basis_matrix(grid, params) = hcat([cheb_polys.(shrink_grid.(grid), n) for n = 0:params
basis_matrix = construct_basis_matrix(capital_grid, params)
```

```
## 7×7 Matrix{Float64}:
##  1.0   0.974928      0.900969  …   0.62349    0.433884      0.222521
##  1.0   0.781831      0.222521     -0.900969  -0.974928     -0.62349
##  1.0   0.433884     -0.62349      -0.222521   0.781831      0.900969
##  1.0  -2.22045e-16  -1.0           1.0       -1.11022e-15  -1.0
##  1.0  -0.433884     -0.62349      -0.222521  -0.781831      0.900969
##  1.0  -0.781831      0.222521  …  -0.900969   0.974928     -0.62349
##  1.0  -0.974928      0.900969      0.62349   -0.433884      0.222521
```

Step 4: Pre-invert your basis matrix

Pro tip: you will be using the *exact same* basis matrix in each loop iteration to recover the coefficients: just pre-invert it to save time because inverting the same matrix every loop is costly (especially when large)

```
basis_inverse = basis_matrix \ I # pre-invert
```

```
## 7×7 Matrix{Float64}:
##  0.142857    0.142857    0.142857   …   0.142857    0.142857    0.142857
##  0.278551    0.22338     0.123967      -0.123967   -0.22338    -0.278551
##  0.25742     0.0635774  -0.17814       -0.17814     0.0635774   0.25742
##  0.22338    -0.123967   -0.278551       0.278551    0.123967   -0.22338
##  0.17814    -0.25742    -0.0635774     -0.0635774  -0.25742     0.17814
##  0.123967   -0.278551    0.22338    …  -0.22338     0.278551   -0.123967
##  0.0635774  -0.17814     0.25742       0.25742    -0.17814     0.0635774
```

Pre-Step 5: Evaluate the continuation value

Pre-Step 5: Evaluate the continuation value

To maximize the Bellman at each grid point we need to evaluate the value function

We need to make a function `eval_value_function(coefficients, capital, params)` that lets us evaluate the continuation value given a vector of coefficients `coefficients`, a vector of capital nodes `capital`, and the model parameters `params`

Pre-Step 5: Evaluate the continuation value

To maximize the Bellman at each grid point we need to evaluate the value function

We need to make a function `eval_value_function(coefficients, capital, params)` that lets us evaluate the continuation value given a vector of coefficients `coefficients`, a vector of capital nodes `capital`, and the model parameters `params`

It needs to:

1. Scale capital back into $[-1, 1]$ (the domain of the Chebyshev polynomials)
2. Use the coefficients and Chebyshev polynomials to evaluate the value function

Pre-Step 5: Evaluate the continuation value

```
# evaluates V on the [-1,1]-equivalent grid
eval_value_function(coefficients, grid, params) = construct_basis_matrix(grid, params) * coeffi
```

Step 5: Inner loop over grid points

Construct a function `loop_grid(params, capital_grid, coefficients)` that loops over the grid points `capital_grid` and solves the Bellman given $\Gamma(x; b^{(p)})$

Pseudocode:

```
for each grid point i:
define the Bellman as a closure so it can take in parameters
maximize the Bellman by choosing consumption
store maximized value in a vector v[i]
end

return vector of maximized values v
```

Step 5: Inner loop over grid points

```
function loop_grid(params, capital_grid, coefficients)
    max_value = similar(coefficients); # initialized max value vector

    # Inner loop over grid points
    for (iteration, capital) in enumerate(capital_grid)

        # Define Bellman as a closure
        function bellman(consumption)
            capital_next = capital^params.alpha - consumption # Next period state
            cont_value = eval_value_function(coefficients, capital_next, params)[1] # Continuat
            value_out = (consumption)^(1-params.eta)/(1-params.eta) + params.beta*cont_value # l
            return -value_out
        end;

        results = optimize(bellman, 0.00*capital^params.alpha, 0.99*capital^params.alpha) # max
        max_value[iteration] = -Optim.minimum(results) # Store max value in vector
    end
    return max_value
end
```

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

Step 5: Outer loop iterating on Bellman

Construct a function `solve_vfi(params, basis_inverse, capital_grid, coefficients)` that iterates on `loop_grid` and solves for the coefficient vector b until the maximized values on the grid converge

Pseudocode:

```
while error > tolerance
call loop_grid to get maximized values
use maximized values and basis matrix to get new coefficients
error is maximum relative difference between current and previous maximized values
end

return vector of maximized values v
```

Step 5: Outer loop iterating on Bellman

```
function solve_vfi(params, basis_inverse, capital_grid, coefficients)
    iteration = 1
    error = 1e10;
    max_value = similar(coefficients);
    value_prev = .1*ones(params.num_points);
    coefficients_store = Vector{Vector}(undef, 1)
    coefficients_store[1] = coefficients
    while error > params.tolerance # Outer loop iterating on Bellman eq
        max_value = loop_grid(params, capital_grid, coefficients) # Inner loop
        coefficients = basis_inverse*max_value # \Psi \ y, recover coefficients
        error = maximum(abs.((max_value - value_prev)./(value_prev))) # compute error
        value_prev = deepcopy(max_value) # save previous values
        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
```

Step 5: Outer loop iterating on Bellman

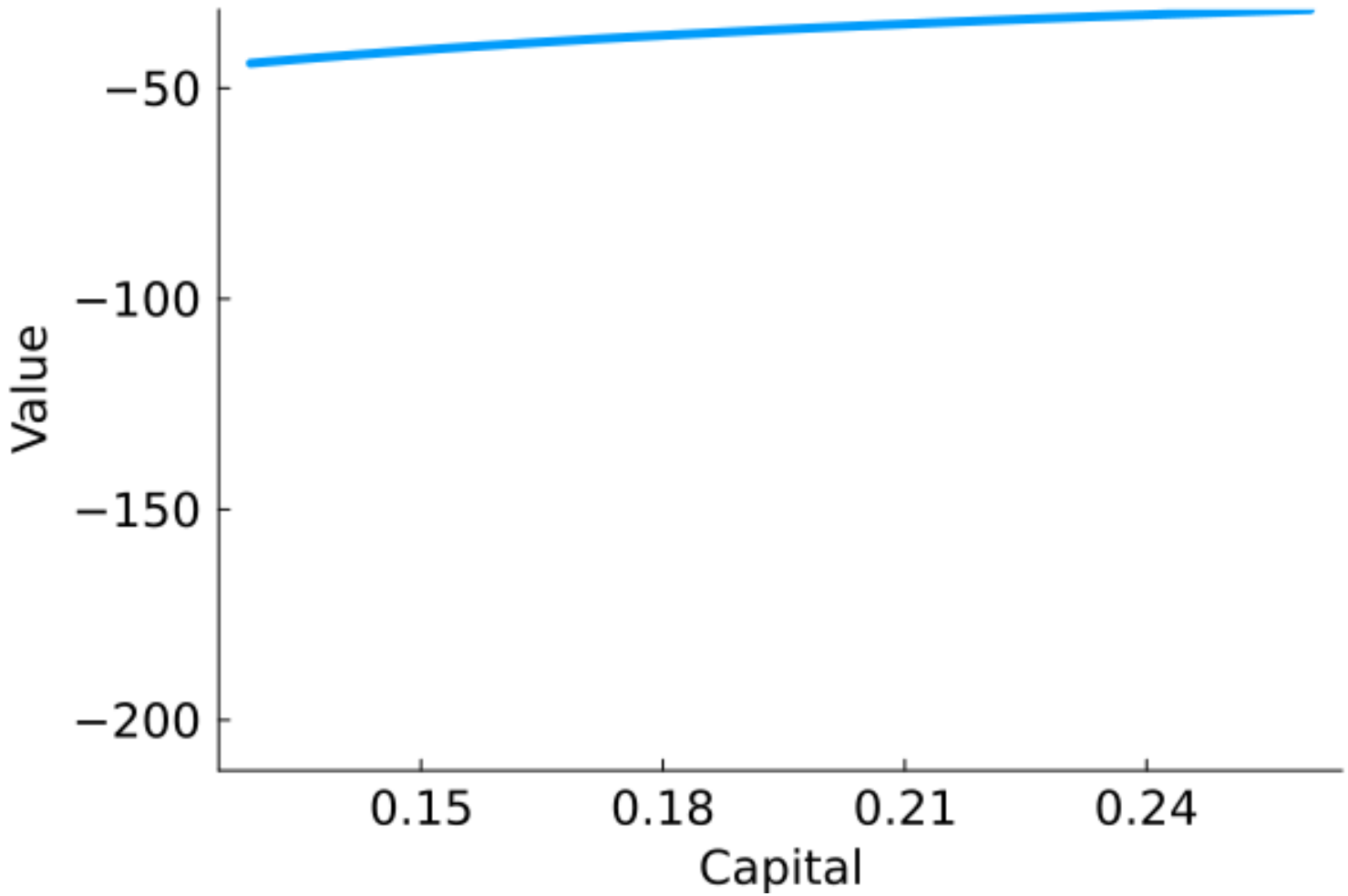
```
solution_coeffs, max_value, intermediate_coefficients =
    solve_vfi(params, basis_inverse, capital_grid, coefficients)
```

```
## Maximum Error of 0.3301919884226089 on iteration 5.
## Maximum Error of 0.10801399197451178 on iteration 10.
## Maximum Error of 0.05647917855011511 on iteration 15.
## Maximum Error of 0.034833389245083446 on iteration 20.
## Maximum Error of 0.023286433761111308 on iteration 25.
## Maximum Error of 0.016301543092581618 on iteration 30.
## Maximum Error of 0.011747480470413624 on iteration 35.
## Maximum Error of 0.008631245645920323 on iteration 40.
## Maximum Error of 0.006427690604127157 on iteration 45.
## Maximum Error of 0.004833073684245354 on iteration 50.
## Maximum Error of 0.003659714890062249 on iteration 55.
## Maximum Error of 0.002785692376976501 4 on iteration 60.
## Maximum Error of 0.002128686710212813 3 on iteration 65.
## Maximum Error of 0.001631424967737030 7 on iteration 70.
## Maximum Error of 0.001253116057138970 3 on iteration 75.
## Maximum Error of 0.000964170879127253 7 on iteration 80.
## Maximum Error of 0.000742816675076792 4 on iteration 85.
```

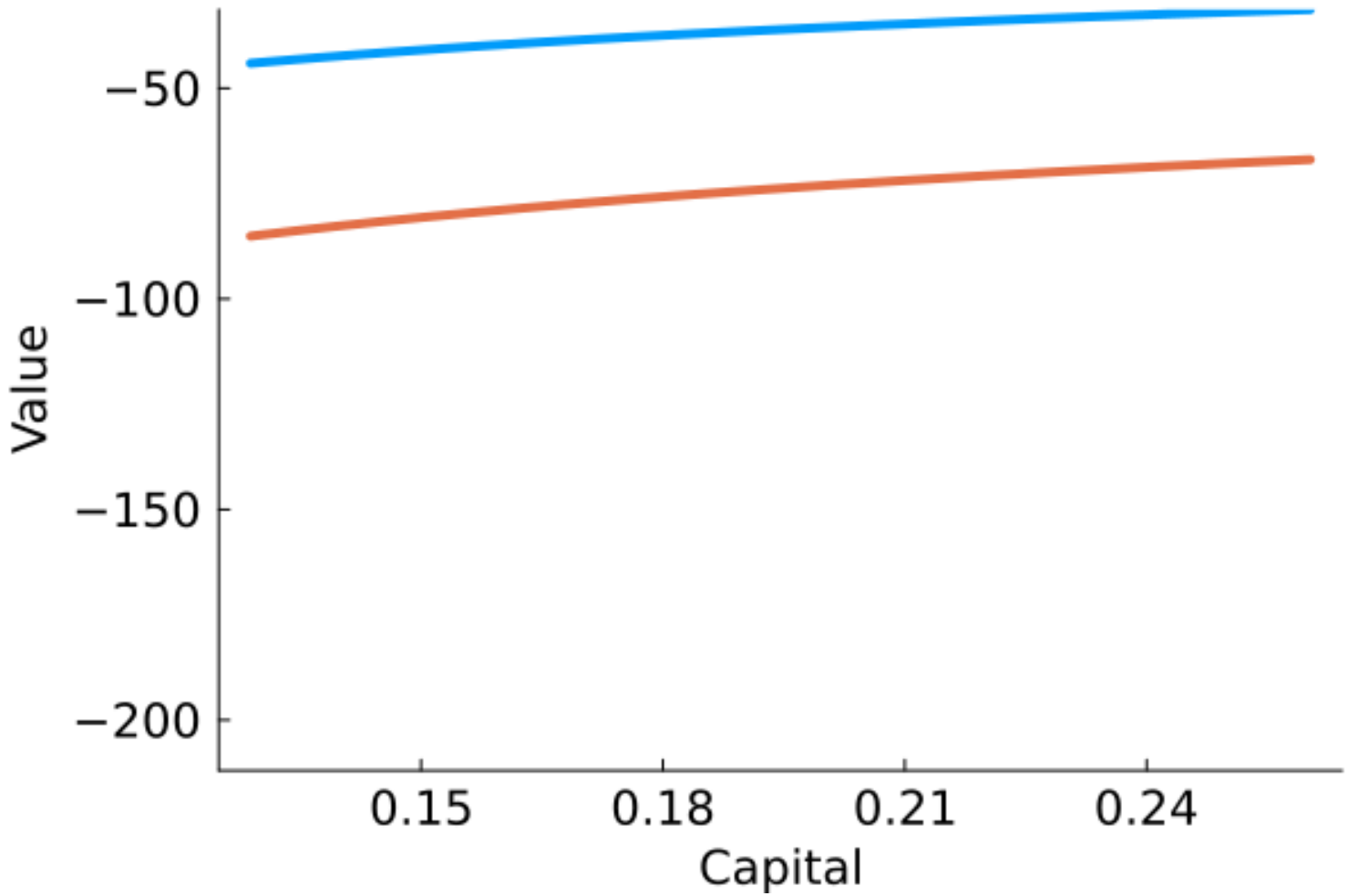
Now lets plot our solutions

```
capital_levels = range(params.capital_lower, params.capital_upper, length = 100);
eval_points = shrink_grid.(capital_levels);

solution = similar(intermediate_coefficients);

# Compute optimal value at all capital grid points
for (iteration, coeffs) in enumerate(intermediate_coefficients)
    solution[iteration] = [coeffs' * [cheb_polys.(capital, n) for n = 0:params.num_points - 1] f
end
```

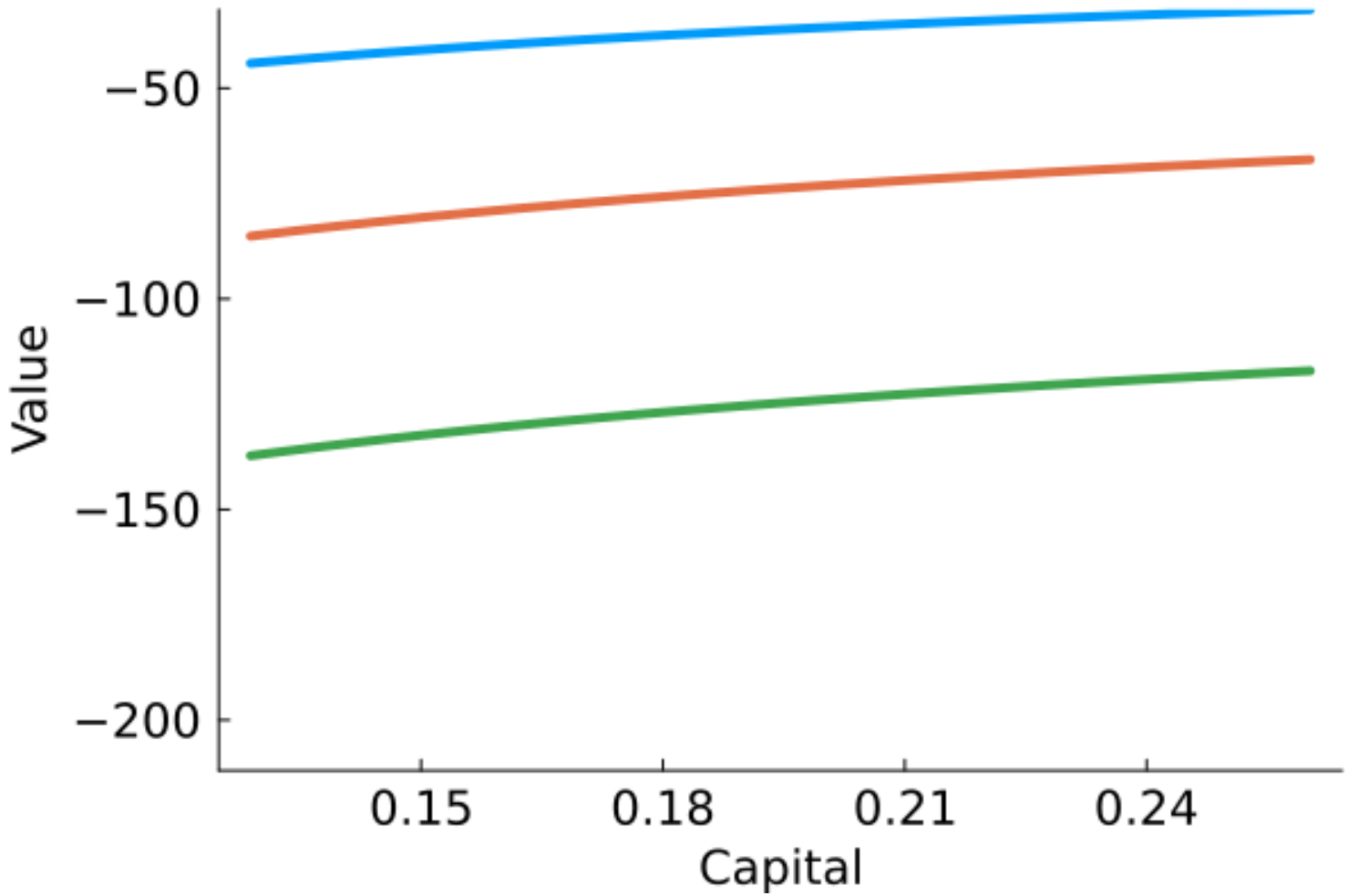
Plot the value function iterations



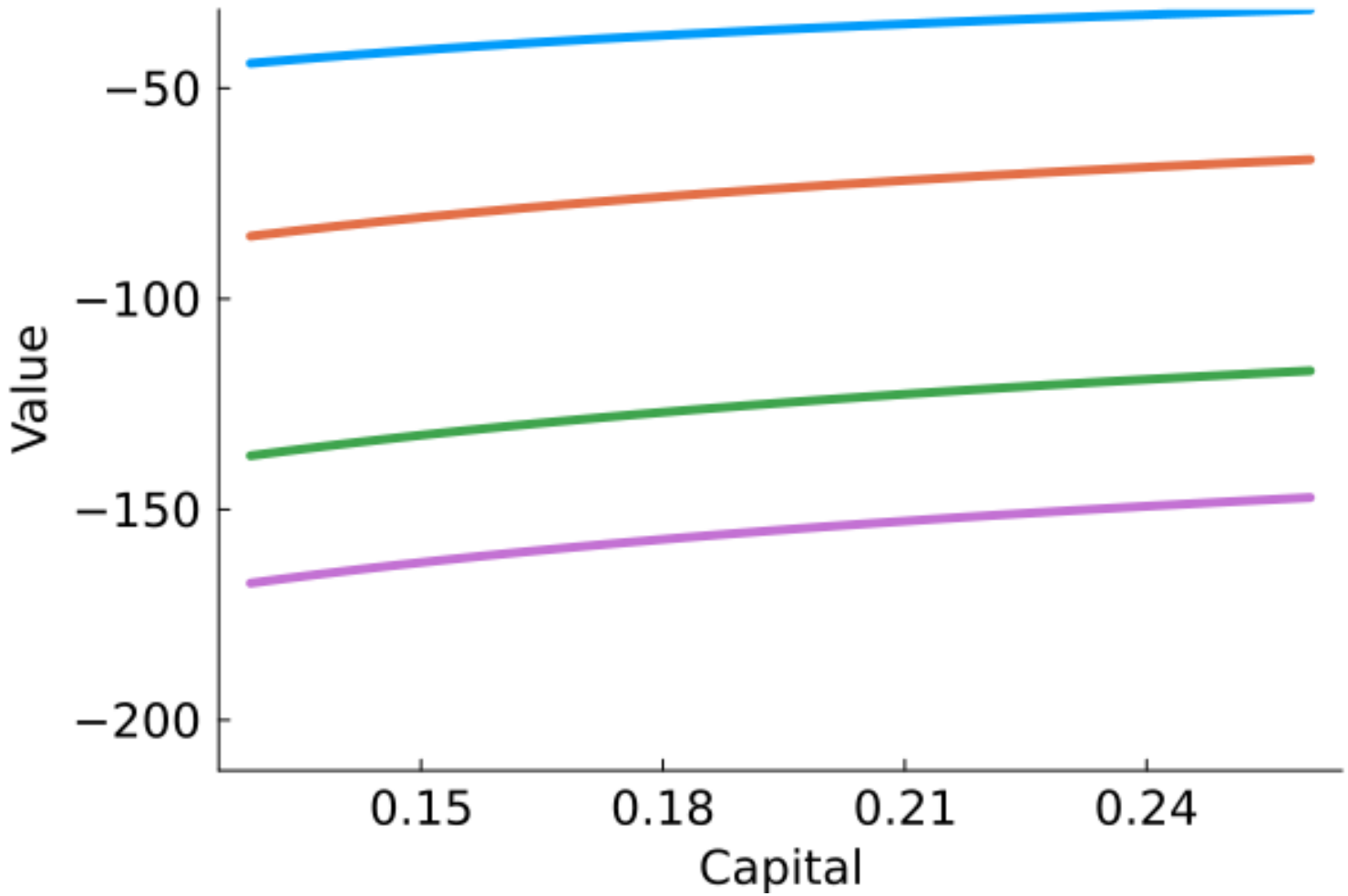
Plot the value function iterations



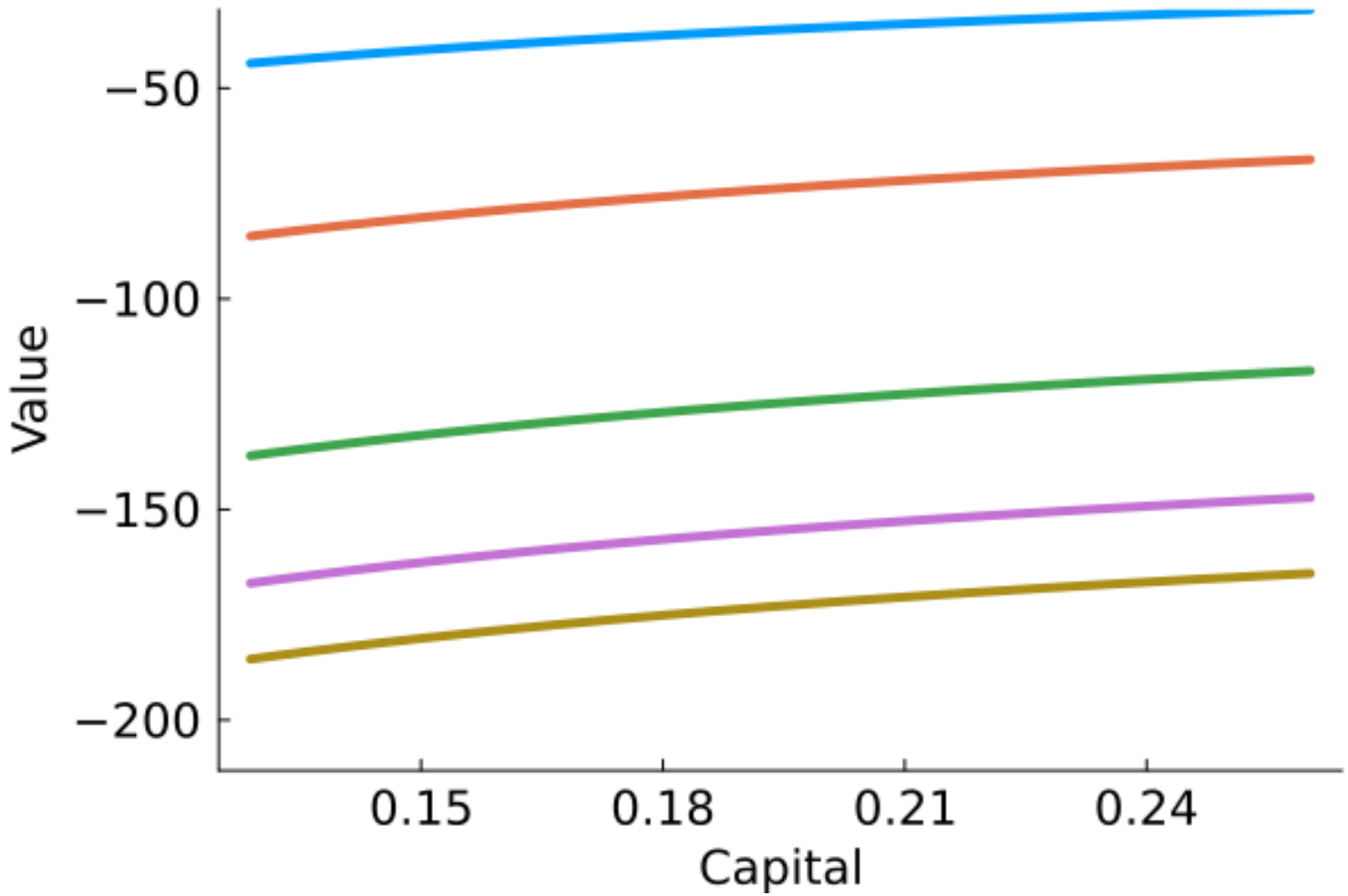
Plot the value function iterations



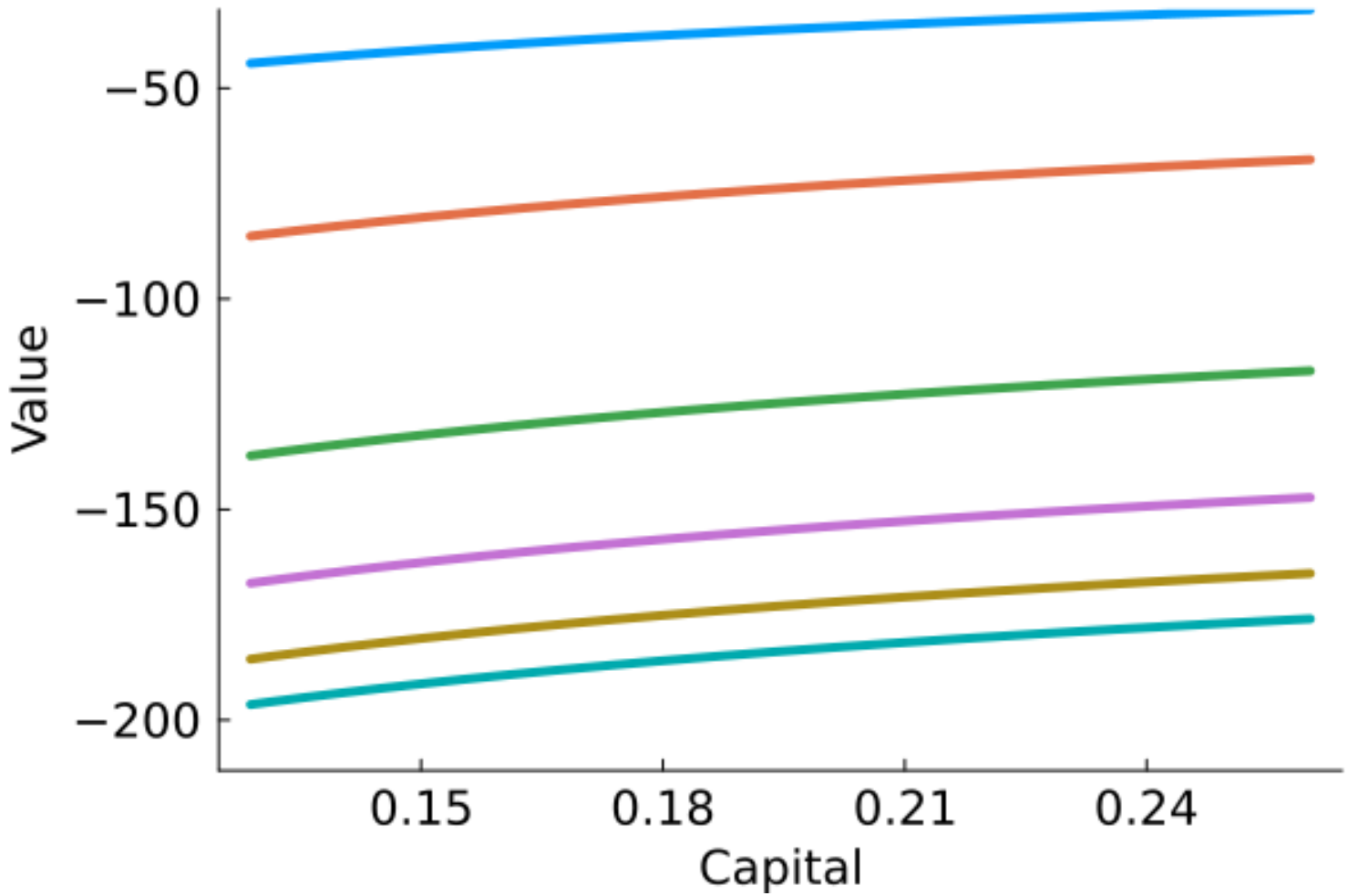
Plot the value function iterations



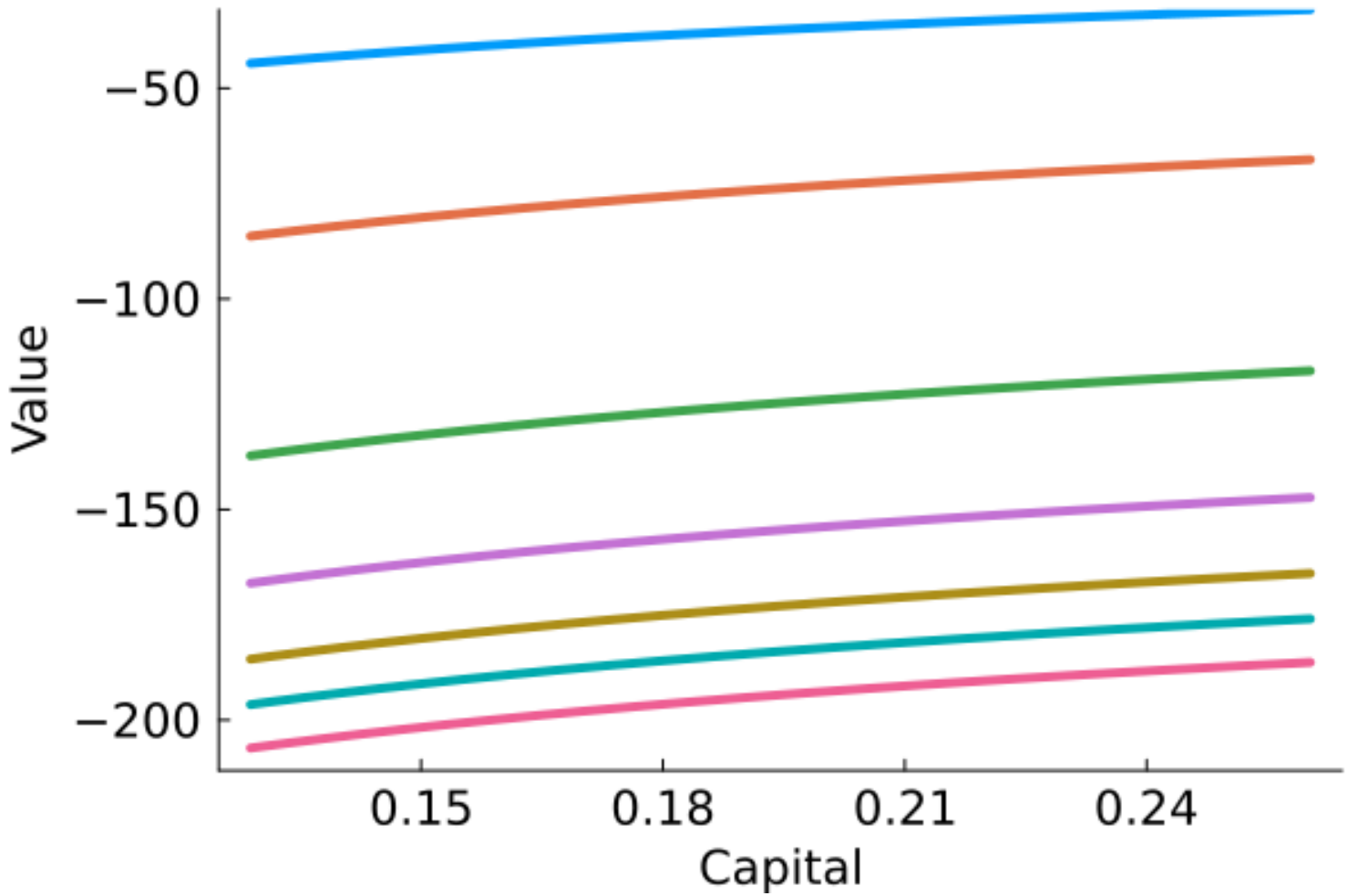
Plot the value function iterations



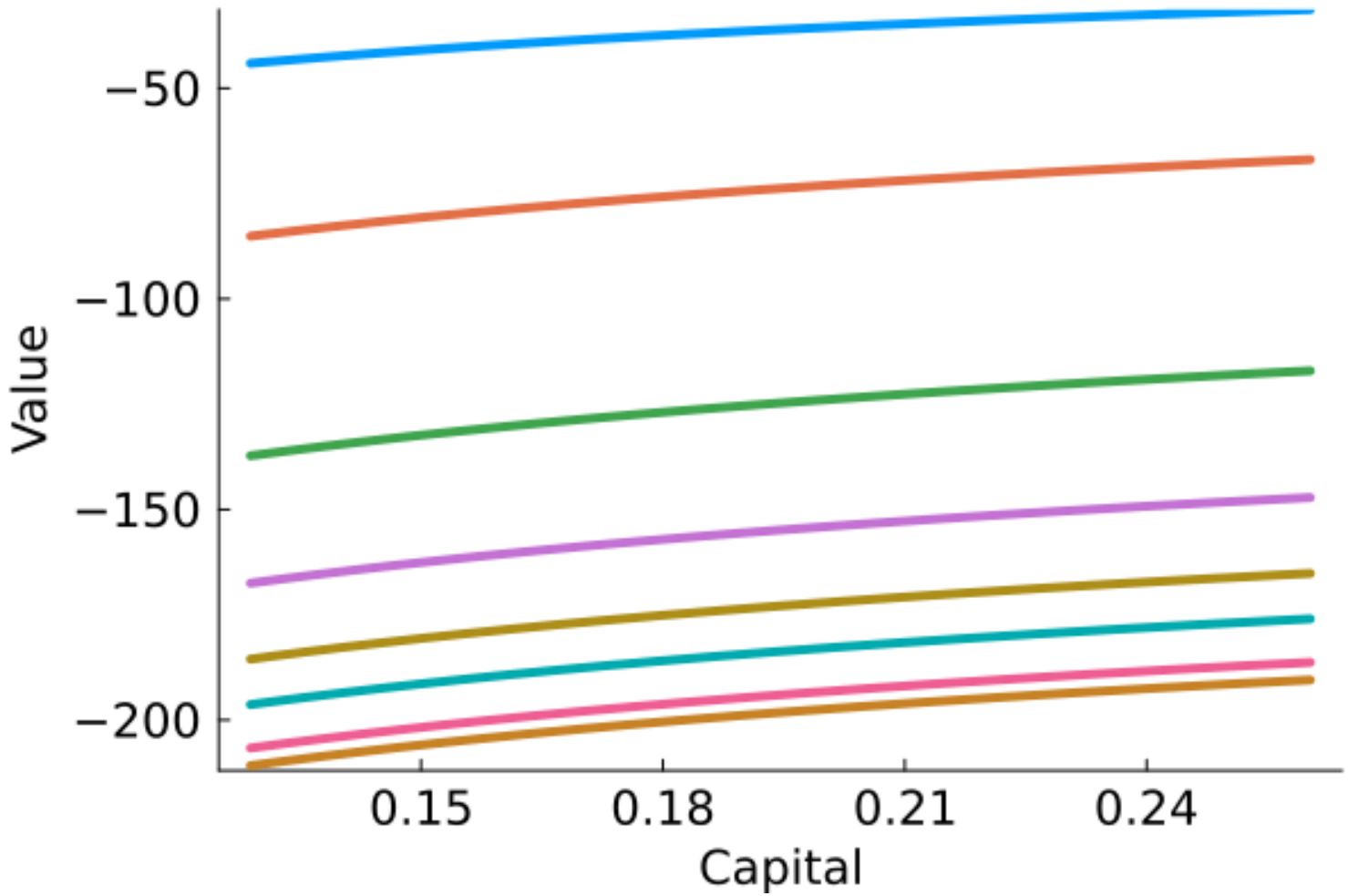
Plot the value function iterations



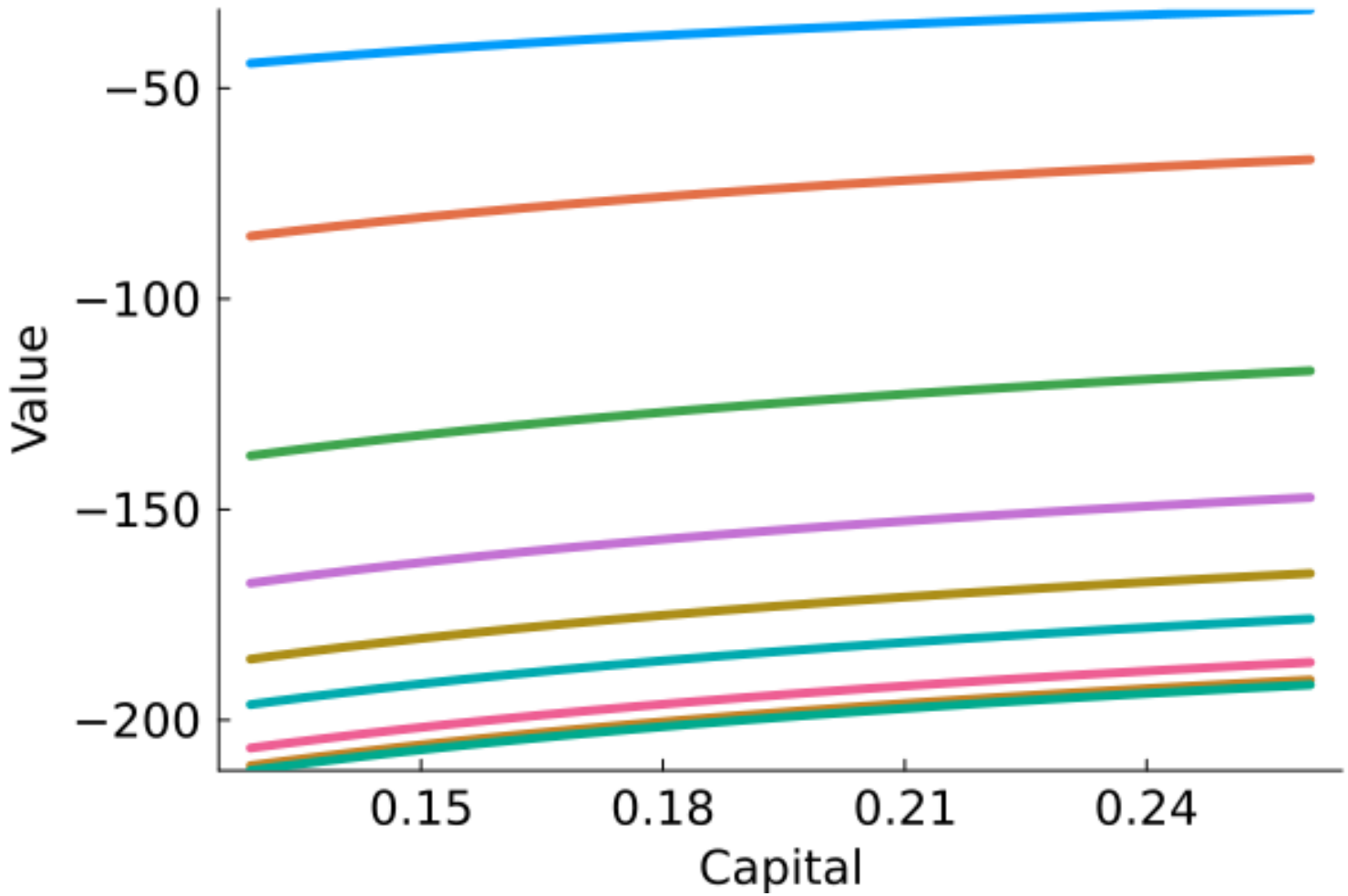
Plot the value function iterations



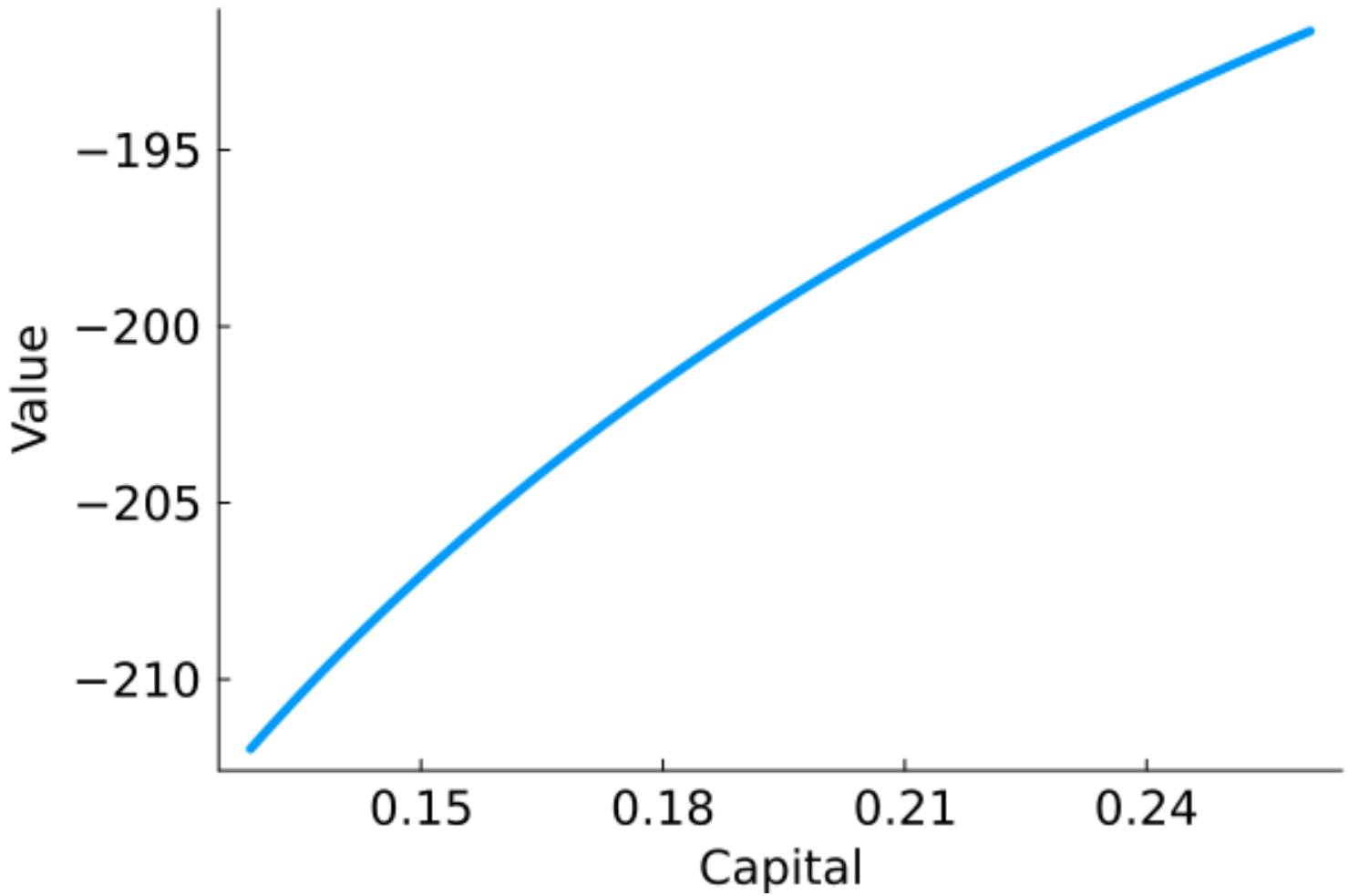
Plot the value function iterations



Plot the value function iterations



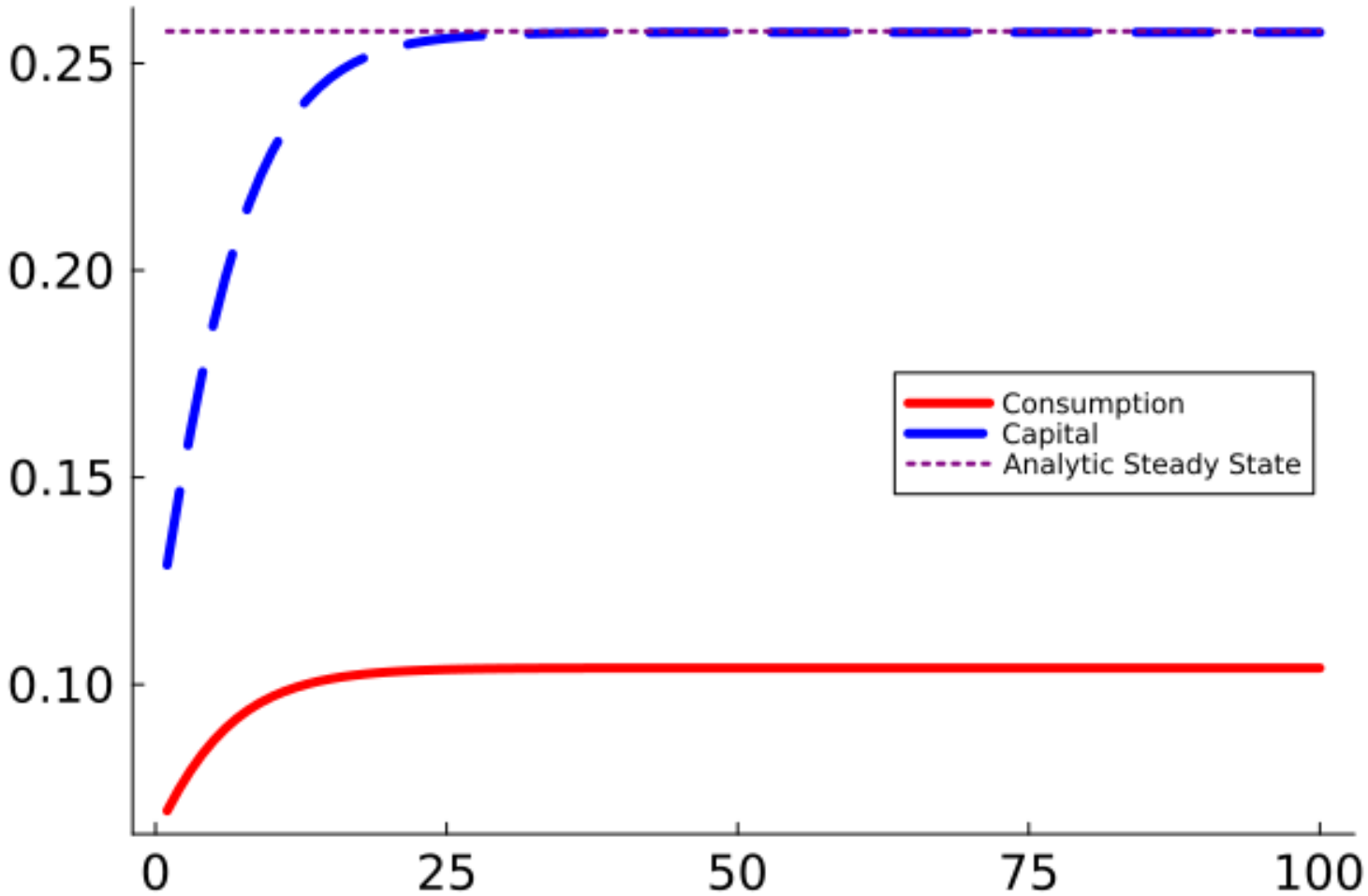
Plot the final value function



Now lets try simulating

```
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
            capital_next_scaled = shrink_grid(capital_next)
            cont_value = solution_coeffs' * [cheb_polys.(capital_next_scaled, n) for n = 0:paran
            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;
```

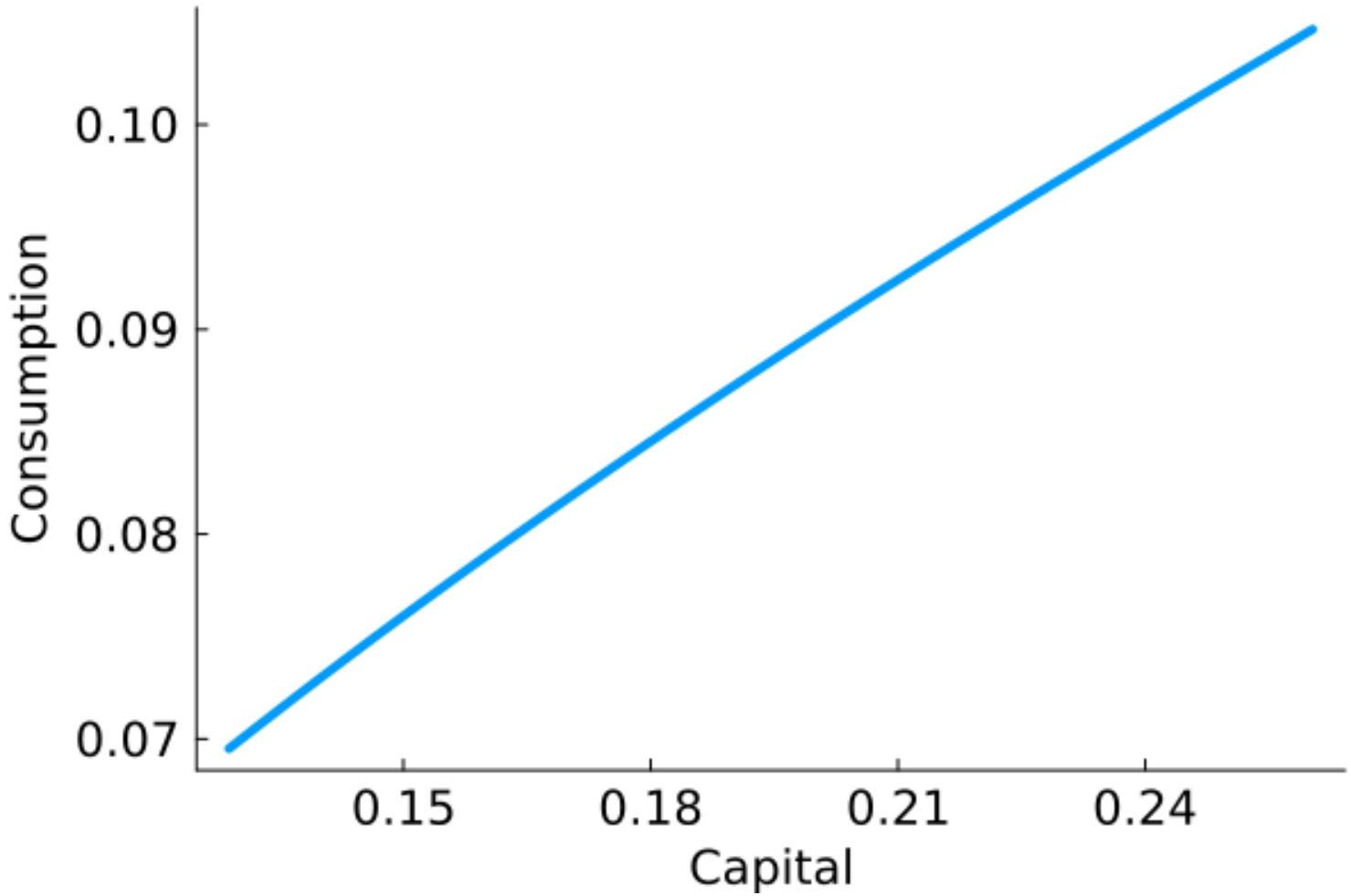
Now lets try simulating



The consumption policy function

```
capital_levels = range(params.capital_lower, params.capital_upper, length = 100);
consumption = similar(capital_levels);

# Compute optimal consumption at all capital grid points
for (iteration, capital) in enumerate(capital_levels)

    function bellman(consumption)
        capital_next = capital^params.alpha - consumption
        capital_next_scaled = shrink_grid(capital_next)
        cont_value = solution_coeffs' * [cheb_polys.(capital_next_scaled, n) for n = 0:params.nu
        value_out = (consumption)^(1-params.eta)/(1-params.eta) + params.beta*cont_value
        return -value_out
    end

    results = optimize(bellman, 0., capital^params.alpha)

    consumption[iteration] = Optim.minimizer(results)
end;
```

The consumption policy function



Fixed point iteration

Method 2: Fixed point iteration

In FPI we generally approximate a policy function with some flexible functional form $\Gamma(k_t; b)$ where b is a vector of coefficients

Method 2: Fixed point iteration

In FPI we generally approximate a policy function with some flexible functional form $\Gamma(k_t; b)$ where b is a vector of coefficients

FPI re-casts equilibrium conditions of the model as a fixed point

Method 2: Fixed point iteration

In FPI we generally approximate a policy function with some flexible functional form $\Gamma(k_t; b)$ where b is a vector of coefficients

FPI re-casts equilibrium conditions of the model as a fixed point

We then perform multi-dimensional function iteration to solve for the fixed point

Method 2: Fixed point iteration

In FPI we generally approximate a policy function with some flexible functional form $\Gamma(k_t; b)$ where b is a vector of coefficients

FPI re-casts equilibrium conditions of the model as a fixed point

We then perform multi-dimensional function iteration to solve for the fixed point

It does not bear a terrible computational cost and is derivative-free

Method 2: Fixed point iteration

In FPI we generally approximate a policy function with some flexible functional form $\Gamma(k_t; b)$ where b is a vector of coefficients

FPI re-casts equilibrium conditions of the model as a fixed point

We then perform multi-dimensional function iteration to solve for the fixed point

It does not bear a terrible computational cost and is derivative-free

The drawback is that it will not always converge and may be unstable

Method 2: Fixed point iteration

In FPI we generally approximate a policy function with some flexible functional form $\Gamma(k_t; b)$ where b is a vector of coefficients

FPI re-casts equilibrium conditions of the model as a fixed point

We then perform multi-dimensional function iteration to solve for the fixed point

It does not bear a terrible computational cost and is derivative-free

The drawback is that it will not always converge and may be unstable

This can be solved by **damping**

Eq condition: Euler equation

Standard procedure is to iterate on the Euler equation

$$C(k_t) = u'^{(-1)}\left(\beta u'(C(k_{t+1}))f'(k_{t+1}(C(k_t), k_t))\right)$$

Method 2: Fixed point iteration

The algorithm has 6 steps, very similar to VFI

Method 2: Fixed point iteration

Step 1: Select the number of collocation points in each dimension and the domain of the approximation space

Method 2: Fixed point iteration

Step 1: Select the number of collocation points in each dimension and the domain of the approximation space

Step 2: Select an initial vector of coefficients b_0 with the same number of elements as the collocation grid

Method 2: Fixed point iteration

Step 1: Select the number of collocation points in each dimension and the domain of the approximation space

Step 2: Select an initial vector of coefficients b_0 with the same number of elements as the collocation grid

Step 3: Select a rule for convergence

Method 2: Fixed point iteration

Step 1: Select the number of collocation points in each dimension and the domain of the approximation space

Step 2: Select an initial vector of coefficients b_0 with the same number of elements as the collocation grid

Step 3: Select a rule for convergence

Step 4: Construct the grid and basis matrix

Method 2: Fixed point iteration

Step 5: While convergence criterion $>$ tolerance **(outer loop)**

- Start iteration p
- For each grid point **(inner loop)**
  - Substitute $C(k_{t+1}; b^{(p)})$ into the right hand side of the Euler fixed point
  - Recover the LHS values of consumption at each grid point
- Fit the polynomial to the values and recover a new vector of coefficients $\hat{b}^{(p+1)}$.
- Compute the vector of coefficients $b^{(p+1)}$ for iteration $p+1$ by $b^{(p+1)} = (1-\gamma)b^{(p)} + \gamma\hat{b}^{(p+1)}$ where $\gamma \in (0,1)$. (damping)

Method 2: Fixed point iteration

Step 5: While convergence criterion $>$ tolerance **(outer loop)**

- Start iteration p
- For each grid point **(inner loop)**
  - Substitute $C(k_{t+1}; b^{(p)})$ into the right hand side of the Euler fixed point
  - Recover the LHS values of consumption at each grid point
- Fit the polynomial to the values and recover a new vector of coefficients $\hat{b}^{(p+1)}$.
- Compute the vector of coefficients $b^{(p+1)}$ for iteration $p+1$ by $b^{(p+1)} = (1-\gamma)b^{(p)} + \gamma\hat{b}^{(p+1)}$ where $\gamma \in (0,1)$. (damping)

Step 6: Error check your approximation

Step 1: Select the number of points and domain

Put everything in a **named tuple** to make passing things easier

```
using LinearAlgebra
using Optim
using Plots
params_fpi = (alpha = 0.75, beta = 0.95, eta = 2, damp = 0.7,
                steady_state = (0.75*0.95)^(1/(1-0.75)), k_0 = (0.75*0.95)^(1/(1-0.75))*0.5,
                capital_upper = (0.75*0.95)^(1/(1-0.75))*1.5, capital_lower = (0.75*0.95)^(1/(1-
                num_points = 5, tolerance = 0.00001)
```

```
## (alpha = 0.75, beta = 0.95, eta = 2, damp = 0.7, steady_state = 0.25771486816406236, k_0 = 0.1288574
```

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

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

Step 2: Select an initial vector of coefficients b_0

```
coefficients = zeros(params_fpi.num_points)
```

```
## 5-element Vector{Float64}:
##  0.0
##  0.0
##  0.0
##  0.0
##  0.0
```

Step 3: Select a convergence rule

Rule: maximum change in consumption on the grid < 0.001%

Step 4: Construct the grid and basis matrix

The function `cheb_nodes` from last lecture constructs the grid on $[-1, 1]$

Step 4: Construct the grid and basis matrix

The function `cheb_nodes` from last lecture constructs the grid on $[-1, 1]$

$$x_k = cos\left(\frac{2k-1}{2n}\pi\right), \ k = 1, \ldots, n$$

Step 4: Construct the grid and basis matrix

```
cheb_nodes(n) = cos.(pi * (2*(1:n) .- 1)./(2n));
grid = cheb_nodes(params_fpi.num_points) # [-1, 1] grid with n points
```

```
## 5-element Vector{Float64}:
##   0.9510565162951535
##   0.5877852522924731
##   6.123233995736766e-17
##  -0.587785252292473
##  -0.9510565162951535
```

Our actual capital domain isn't on $[-1, 1]$, we need to expand the grid to some arbitrary $[a, b]$

Step 4: Construct the grid and basis matrix

```
expand_grid(grid, params_fpi) = # function that expands [-1,1] to [a,b]
   (1 .+ grid)*(params_fpi.capital_upper - params_fpi.capital_lower)/2 .+ params_fpi.capital_lo
```

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

```
capital_grid = expand_grid(grid, params_fpi)
```

```
## 5-element Vector{Float64}:
##  0.3802655705208513
##  0.33345536756572974
##  0.2577148681640623
##  0.18197436876239492
##  0.1351641658072734
```

Step 4: Construct the grid and basis matrix

Make the inverse function to shrink from capital to Chebyshev space

```
shrink_grid(capital, params)
```

Step 4: Construct the grid and basis matrix

Make the inverse function to shrink from capital to Chebyshev space

`shrink_grid(capital, params)`

```
shrink_grid(capital) =
  2*(capital - params_fpi.capital_lower)/(params_fpi.capital_upper - params_fpi.capital_lower) -
shrink_grid.(capital_grid)
```

```
## 5-element Vector{Float64}:
##   0.9510565162951536
##   0.5877852522924731
##  -2.220446049250313e-16
##  -0.5877852522924731
##  -0.9510565162951536
```

`shrink_grid` will inherit `params_fpi` from wrapper functions

Step 4: Construct the grid and basis matrix

Step 4: Construct the grid and basis matrix

`cheb_polys(x, n)` from last lecture gives us the nth degree Chebyshev polynomial at point x

Step 4: Construct the grid and basis matrix

`cheb_polys(x, n)` from last lecture gives us the nth degree Chebyshev polynomial at point x

```
# 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;
```

Step 4: Construct the grid and basis matrix

`cheb_polys.(grid, n)` gives us the nth degree Chebyshev polynomial at all points on our grid

Step 4: Construct the grid and basis matrix

`cheb_polys.(grid, n)` gives us the nth degree Chebyshev polynomial at all points on our grid

```
cheb_polys.(grid, 2) # 2nd degree Cheb poly at each grid point
```

```
## 5-element Vector{Float64}:
##   0.8090169943749472
##  -0.30901699437494745
##  -1.0
##  -0.3090169943749477
##   0.8090169943749472
```

Step 4: Construct the grid and basis matrix

In our basis matrix, rows are grid points, columns are basis functions, make a function `construct_basis_matrix(grid, params)` that makes the basis matrix for some arbitrary grid of points

```
construct_basis_matrix(grid, params_fpi) = hcat([cheb_polys.(shrink_grid.(grid), n) for n = 0:pa
basis_matrix = construct_basis_matrix(capital_grid, params_fpi)
```

```
## 5×5 Matrix{Float64}:
##  1.0   0.951057      0.809017   0.587785      0.309017
##  1.0   0.587785     -0.309017  -0.951057     -0.809017
##  1.0  -2.22045e-16  -1.0        6.66134e-16   1.0
##  1.0  -0.587785     -0.309017   0.951057     -0.809017
##  1.0  -0.951057      0.809017  -0.587785      0.309017
```

Step 4: Pre-invert your basis matrix

Pro tip: you will be using the *exact same* basis matrix in each loop iteration to recover the coefficients: just pre-invert it to save time because inverting the same matrix every loop is costly (especially when large)

```
basis_inverse = basis_matrix \ I # pre-invert
```

```
## 5×5 Matrix{Float64}:
##  0.2        0.2        0.2            0.2        0.2
##  0.380423   0.235114  -4.69836e-17   -0.235114  -0.380423
##  0.323607  -0.123607  -0.4           -0.123607   0.323607
##  0.235114  -0.380423   2.66269e-17    0.380423  -0.235114
##  0.123607  -0.323607   0.4           -0.323607   0.123607
```

Pre-Step 5: Evaluate the policy function

We need to make a function `eval_policy_function(coefficients, capital, params_fpi)` that lets us evaluate the policy function given a vector of coefficients `coefficients`, a vector of capital nodes `capital`, and the model parameters `params_fpi`

Pre-Step 5: Evaluate the policy function

We need to make a function `eval_policy_function(coefficients, capital, params_fpi)` that lets us evaluate the policy function given a vector of coefficients `coefficients`, a vector of capital nodes `capital`, and the model parameters `params_fpi`

It needs to:

1. Scale capital back into $[-1, 1]$ (the domain of the Chebyshev polynomials)
2. Use the coefficients and Chebyshev polynomials to evaluate the value function

Pre-Step 5: Evaluate the policy function

```
# evaluates V on the [-1,1]-equivalent grid
eval_policy_function(coefficients, capital, params_fpi) =
    construct_basis_matrix(capital, params_fpi) * coefficients;
```

Step 5: Loop

Construct a function `consumption_euler(params_fpi, capital, coefficients)` that evaluates the RHS of the Euler

Step 5: Loop

Construct a function `consumption_euler(params_fpi, capital, coefficients)` that evaluates the RHS of the Euler

```
function consumption_euler(params_fpi, capital, coefficients)
    # RHS: Current consumption given current capital
    consumption = eval_policy_function(coefficients, capital, params_fpi)[1]
    # RHS: Next period's capital given current capital and consumption
    capital_next = capital^params_fpi.alpha - consumption
    # RHS: Next period's consumption given current capital and consumption
    consumption_next = eval_policy_function(coefficients, capital_next, params_fpi)[1]
    consumption_next = max(1e-10, consumption_next)
    # LHS: Next period's consumption from Euler equation
    consumption_lhs = (
      params_fpi.beta * consumption_next^(-params_fpi.eta) * params_fpi.alpha*(capital_next).^(p
        ).^(-1/params_fpi.eta)
    return consumption_lhs
end
```



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

Step 5: Loop

Construct a function `loop_grid_fpi(params_fpi, capital_grid, coefficients)` that loops over the grid points and evaluates the RHS of the Euler given $\Psi(x; b^{(p)})$

Step 5: Loop

Construct a function `loop_grid_fpi(params_fpi, capital_grid, coefficients)` that loops over the grid points and evaluates the RHS of the Euler given $\Psi(x; b^{(p)})$

```
function loop_grid_fpi(params_fpi, capital_grid, coefficients)

    consumption = similar(coefficients)

    # Compute next period's consumption from the Euler equation
    for (iteration, capital) in enumerate(capital_grid)
        consumption[iteration] = consumption_euler(params_fpi, capital, coefficients)
    end
    return consumption
end
```

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

Step 5: Loop

Construct a function `solve_fpi(params_fpi, basis_inverse, capital_grid, coefficients)` that iterates on `loop_grid_fpi` and solves for the coefficient vector b until the consumption values on the grid converge

```
function solve_fpi(params_fpi, basis_inverse, capital_grid, coefficients)
    error = 1e10
    iteration = 1
    consumption = similar(coefficients)
    consumption_prev, coefficients_prev = similar(coefficients), similar(coefficients)
    coefficients_store = Vector{Vector}(undef, 1)
    coefficients_store[1] = coefficients
    while error > params_fpi.tolerance
        consumption = loop_grid_fpi(params_fpi, capital_grid, coefficients)
        if iteration > 1
            coefficients = params_fpi.damp*(basis_inverse*consumption) + (1 - params_fpi.damp)*c
        else
            coefficients = basis_inverse*consumption
        end
```

Step 5: Loop

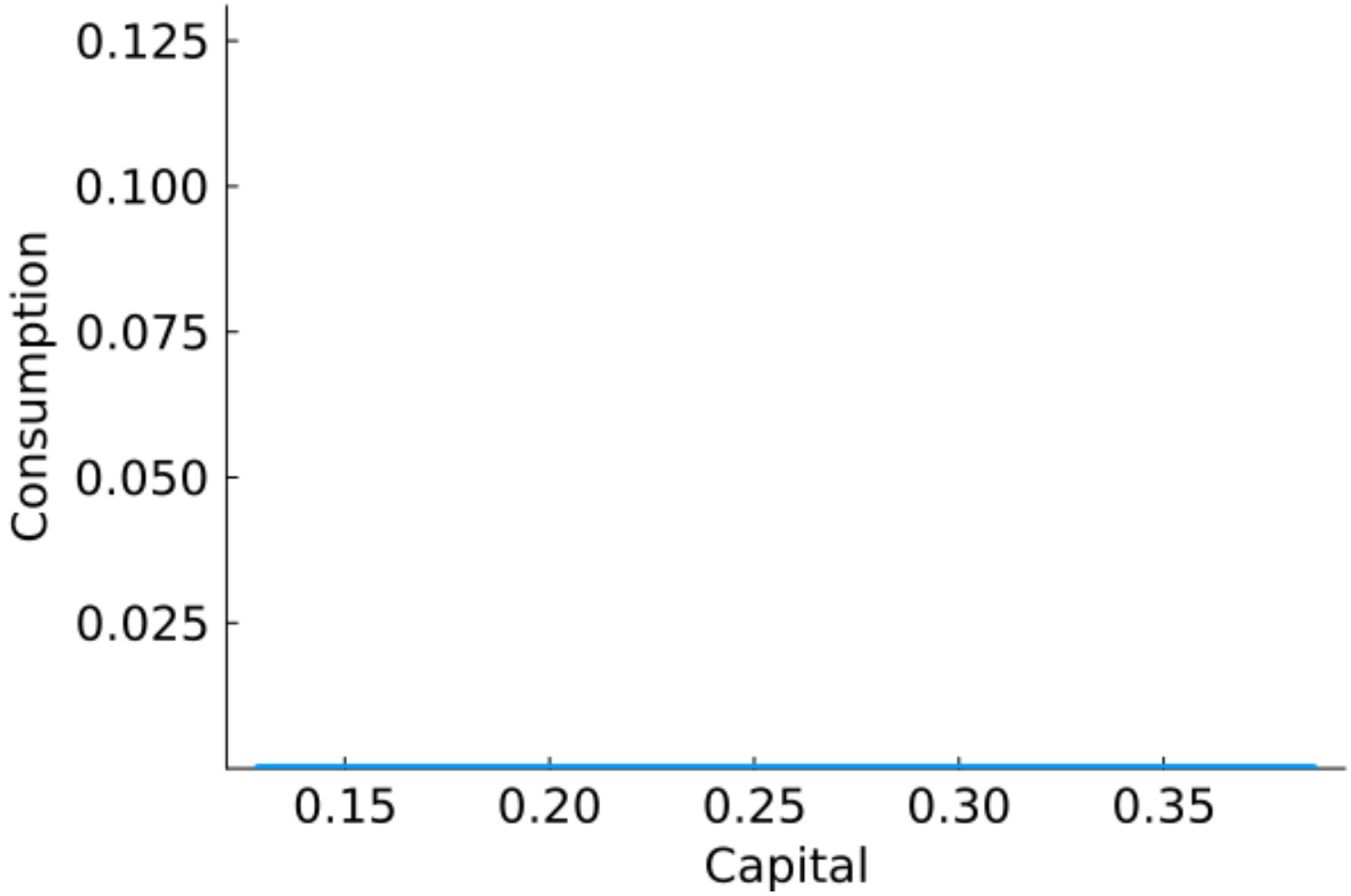
```
solution_coeffs, consumption, intermediate_coefficients =
    solve_fpi(params_fpi, basis_inverse, capital_grid, coefficients)
```

```
## Maximum Error of 0.06899533243794273 on iteration 5.
## Maximum Error of 1.4814840914335052 on iteration 10.
## Maximum Error of 0.7160886352973411 on iteration 15.
## Maximum Error of 0.30308290299478047 on iteration 20.
## Maximum Error of 0.17473142803887126 on iteration 25.
## Maximum Error of 505.4941565432374 on iteration 30.
## Maximum Error of 1.1935815404160586 on iteration 35.
## Maximum Error of 0.2847765742094213 on iteration 40.
## Maximum Error of 0.19537333980794477 on iteration 45.
## Maximum Error of 236164.34266168138 on iteration 50.
## Maximum Error of 1.4738022100603876 on iteration 55.
## Maximum Error of 0.38721294651838284 on iteration 60.
## Maximum Error of 0.21074834397654027 on iteration 65.
## Maximum Error of 0.2361368548338292 on iteration 70.
## Maximum Error of 4.722253098129778 on iteration 75.
## Maximum Error of 0.03764118064354074 on iteration 80.
## Maximum Error of 0.007970939078890096 on iteration 85.
```

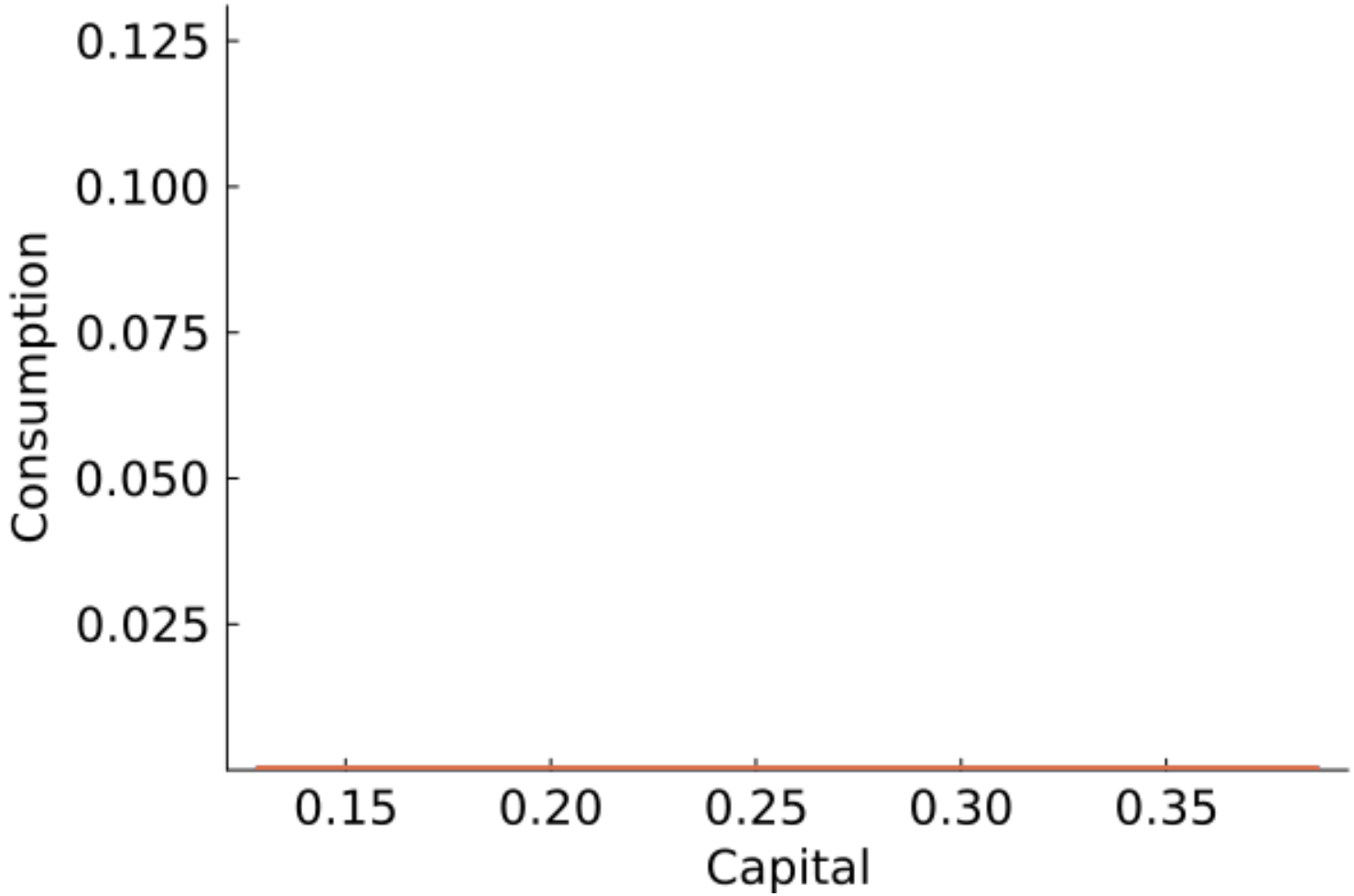
Now lets plot our solutions

```
capital_levels = range(params_fpi.capital_lower, params_fpi.capital_upper, length = 100);
eval_points = shrink_grid.(capital_levels);
solution = similar(intermediate_coefficients);

for (iteration, coeffs) in enumerate(intermediate_coefficients)
    solution[iteration] = [coeffs'*[cheb_polys.(capital, n) for n = 0:params_fpi.num_points - 1]
end
```

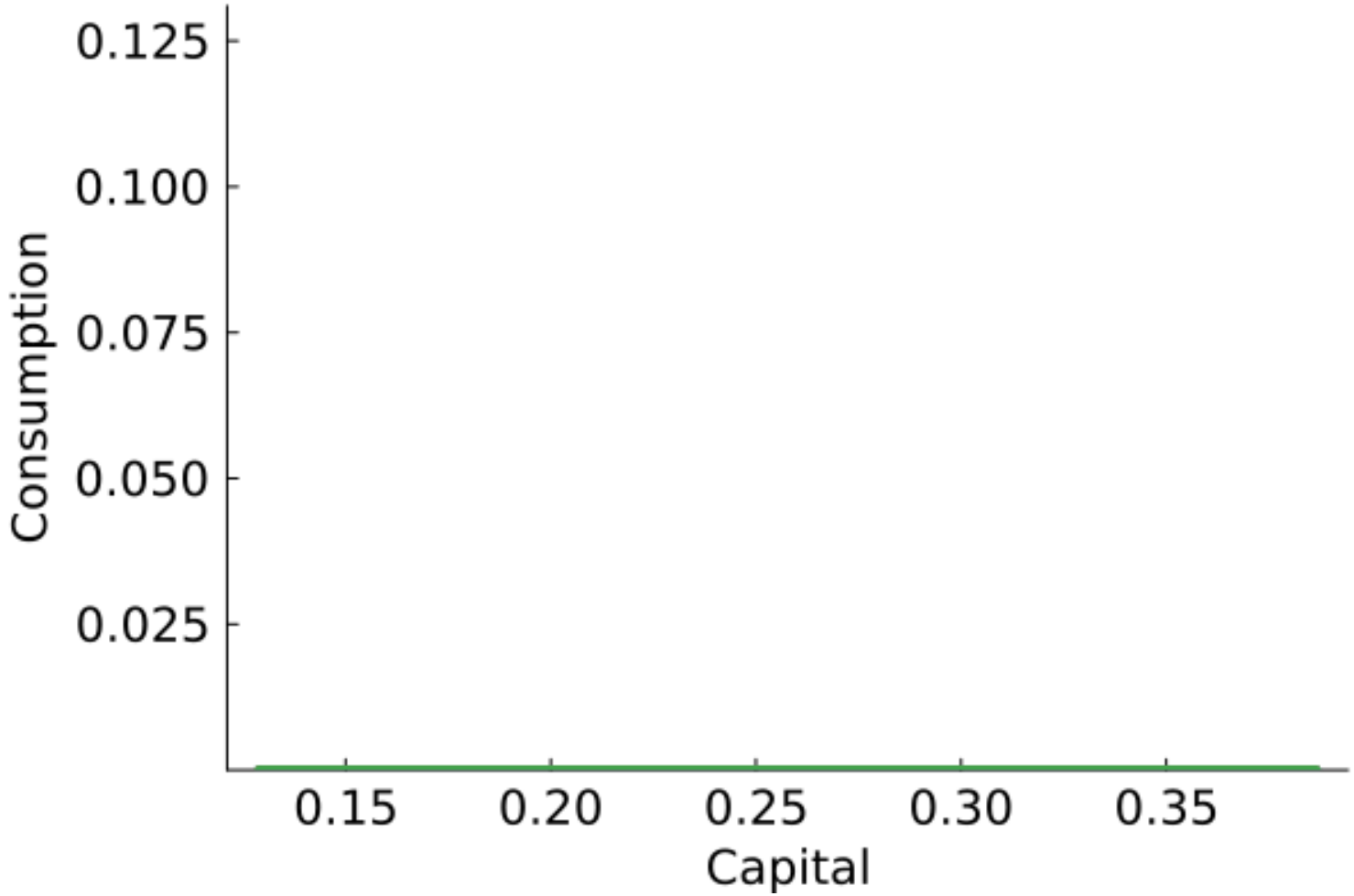
Plot the consumption policy function



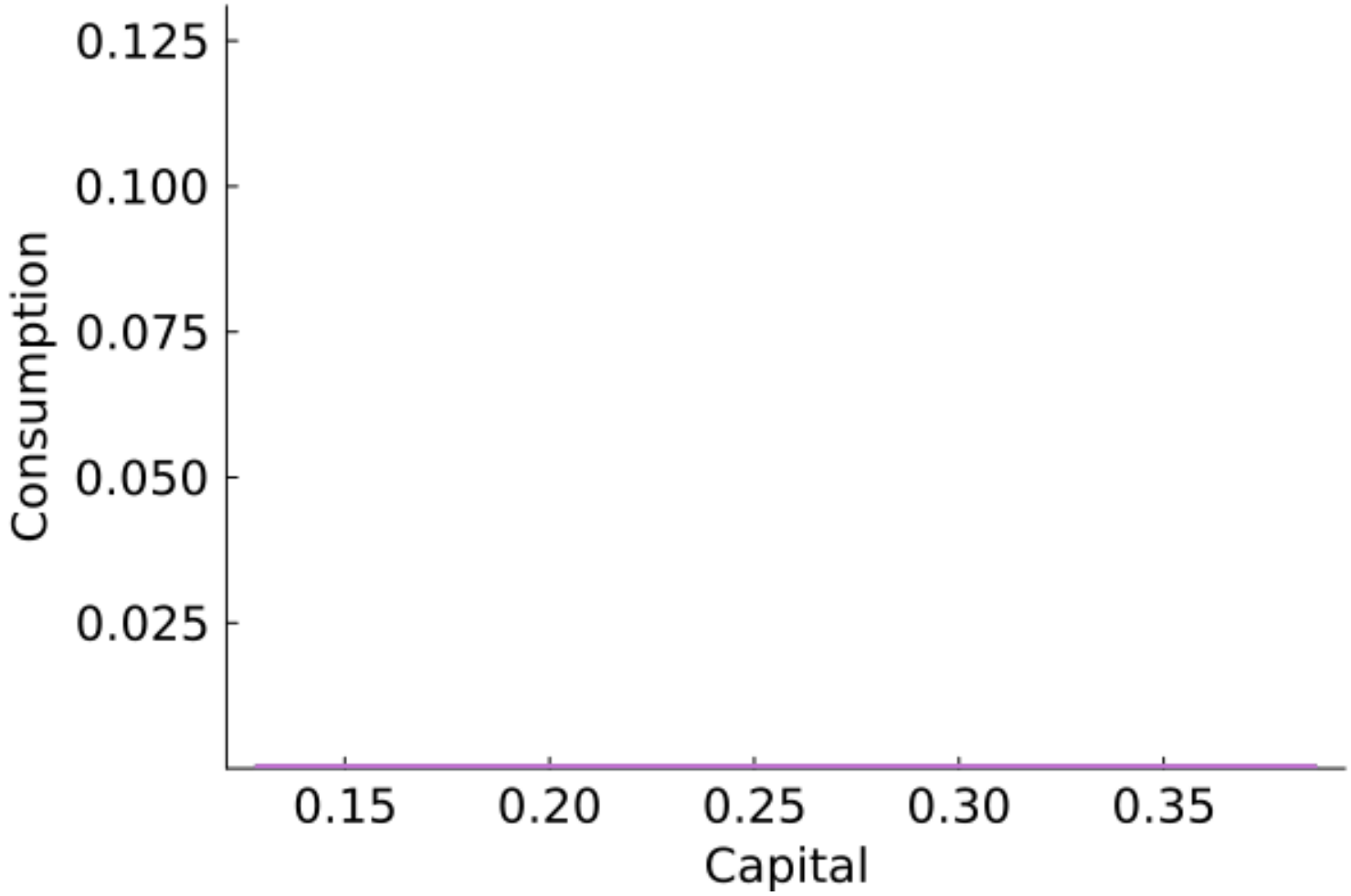
Plot the consumption policy function



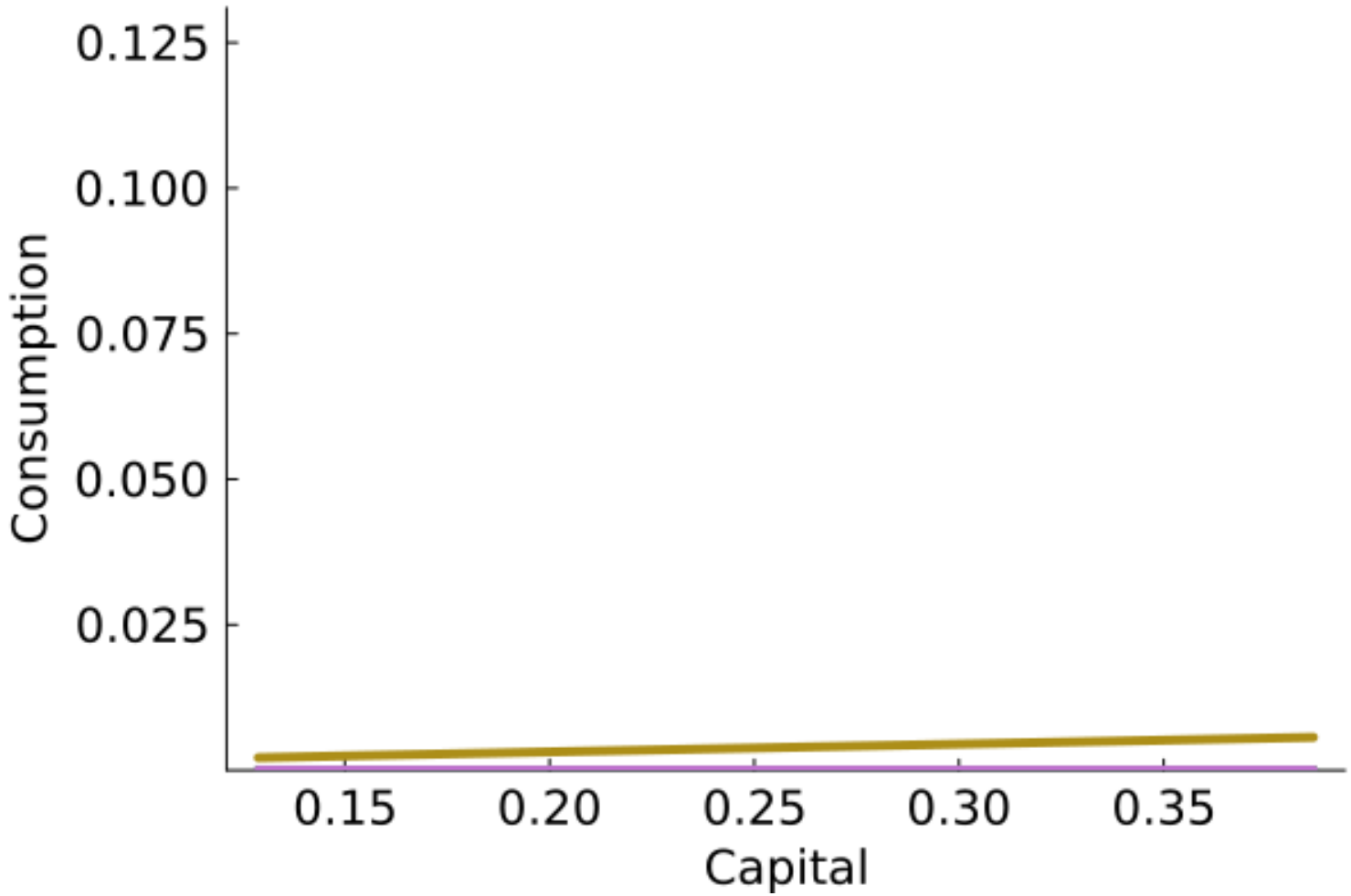
Plot the consumption policy function



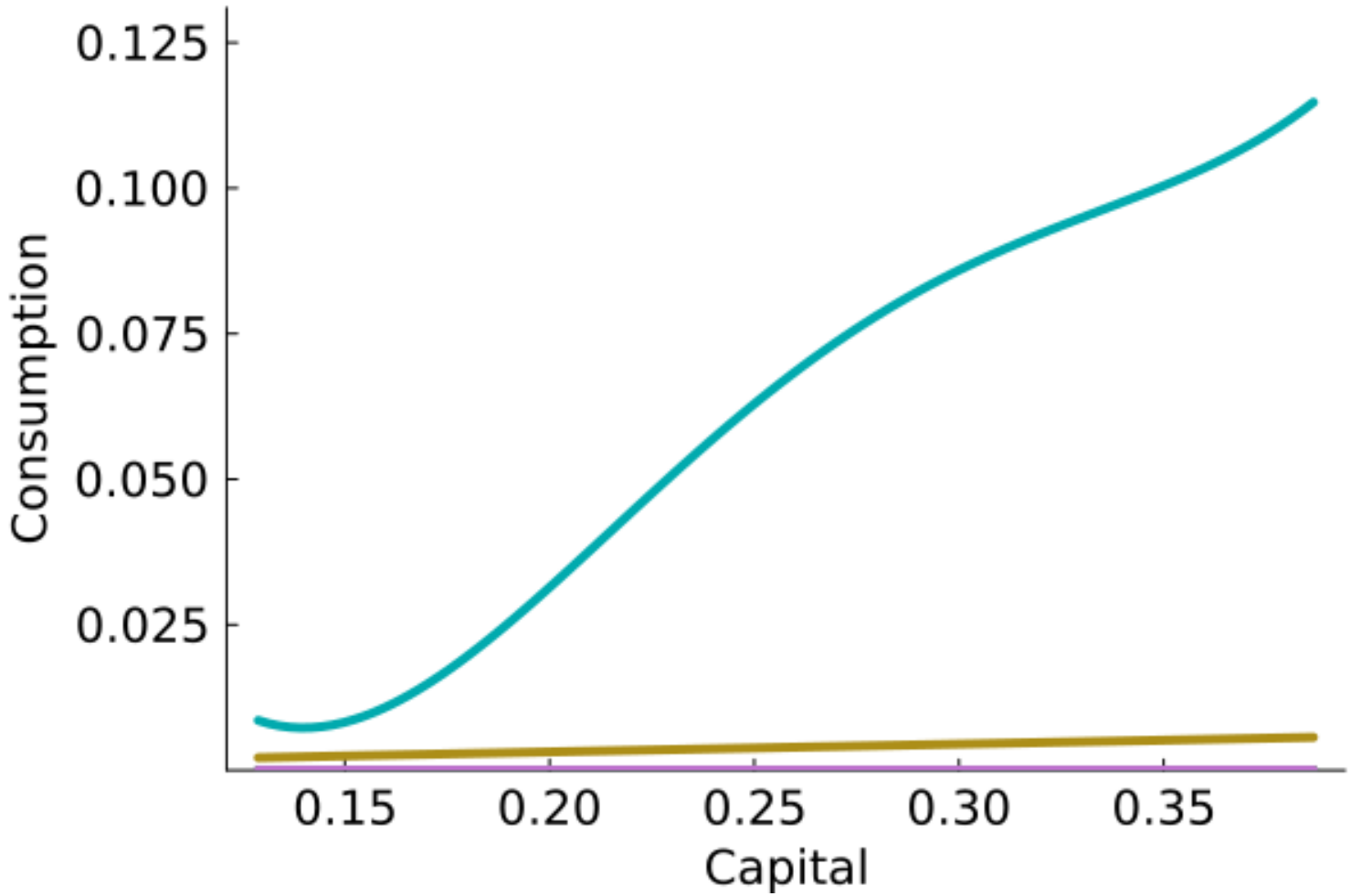
Plot the consumption policy function



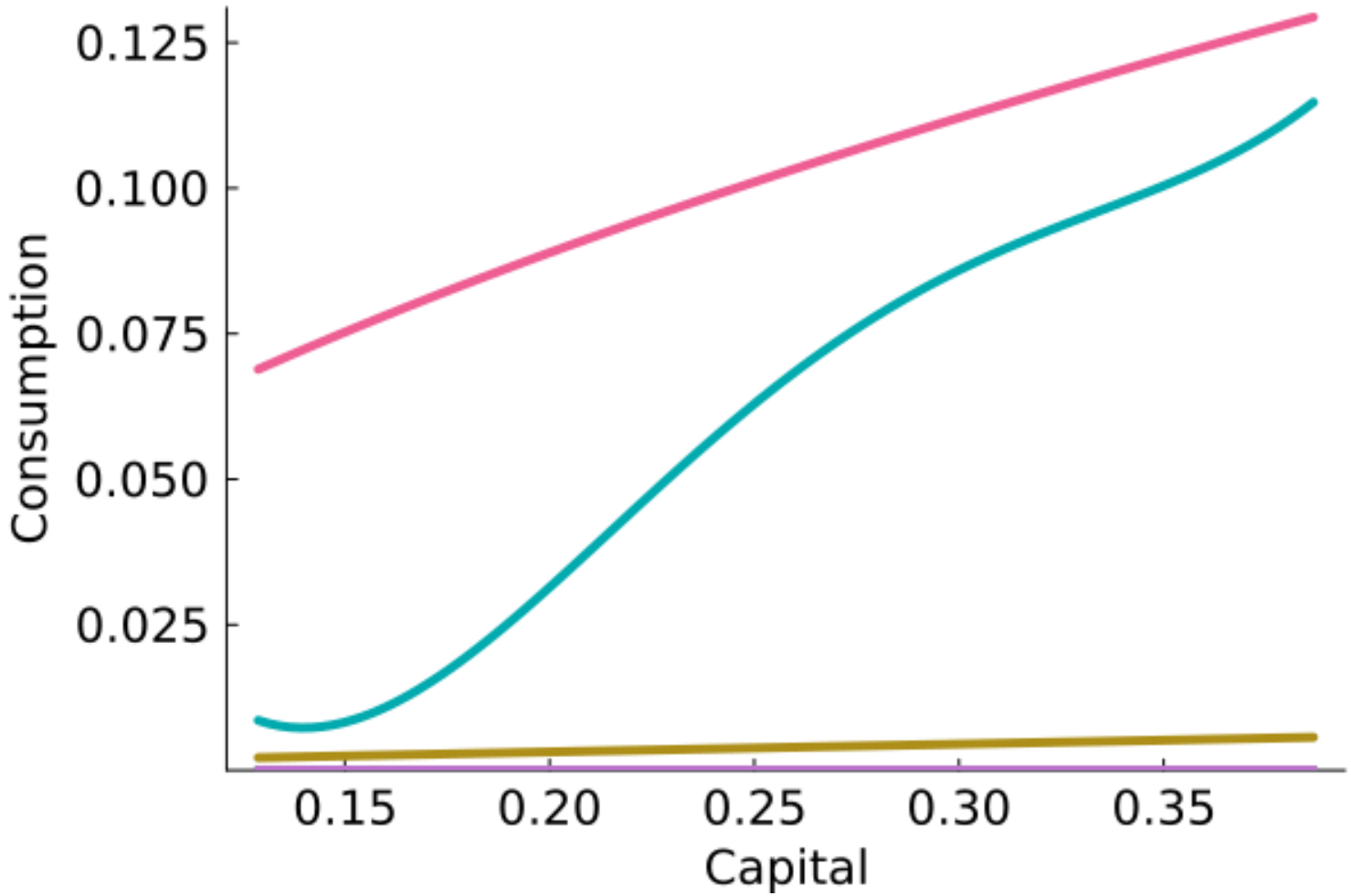
Plot the consumption policy function



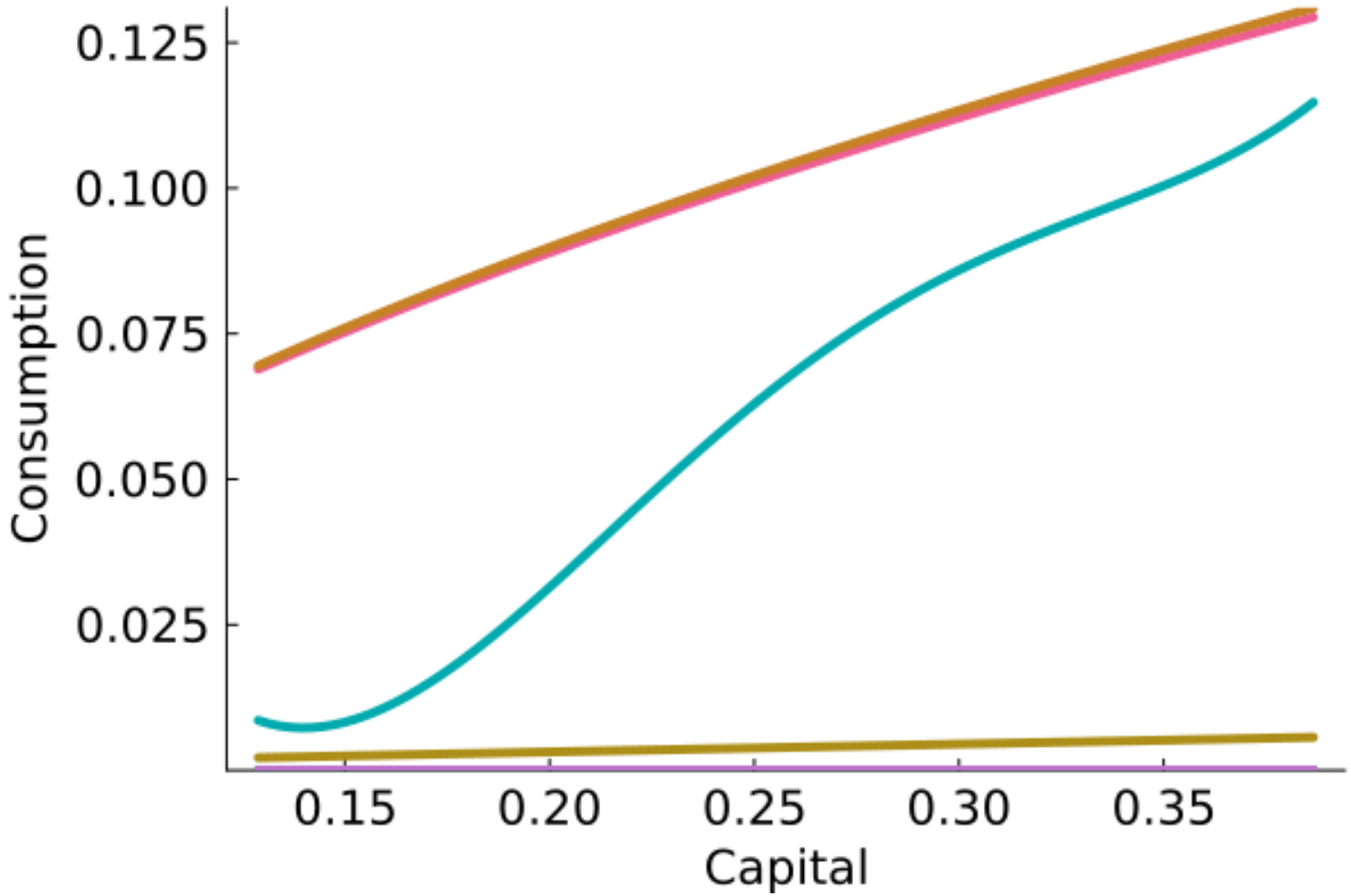
Plot the consumption policy function



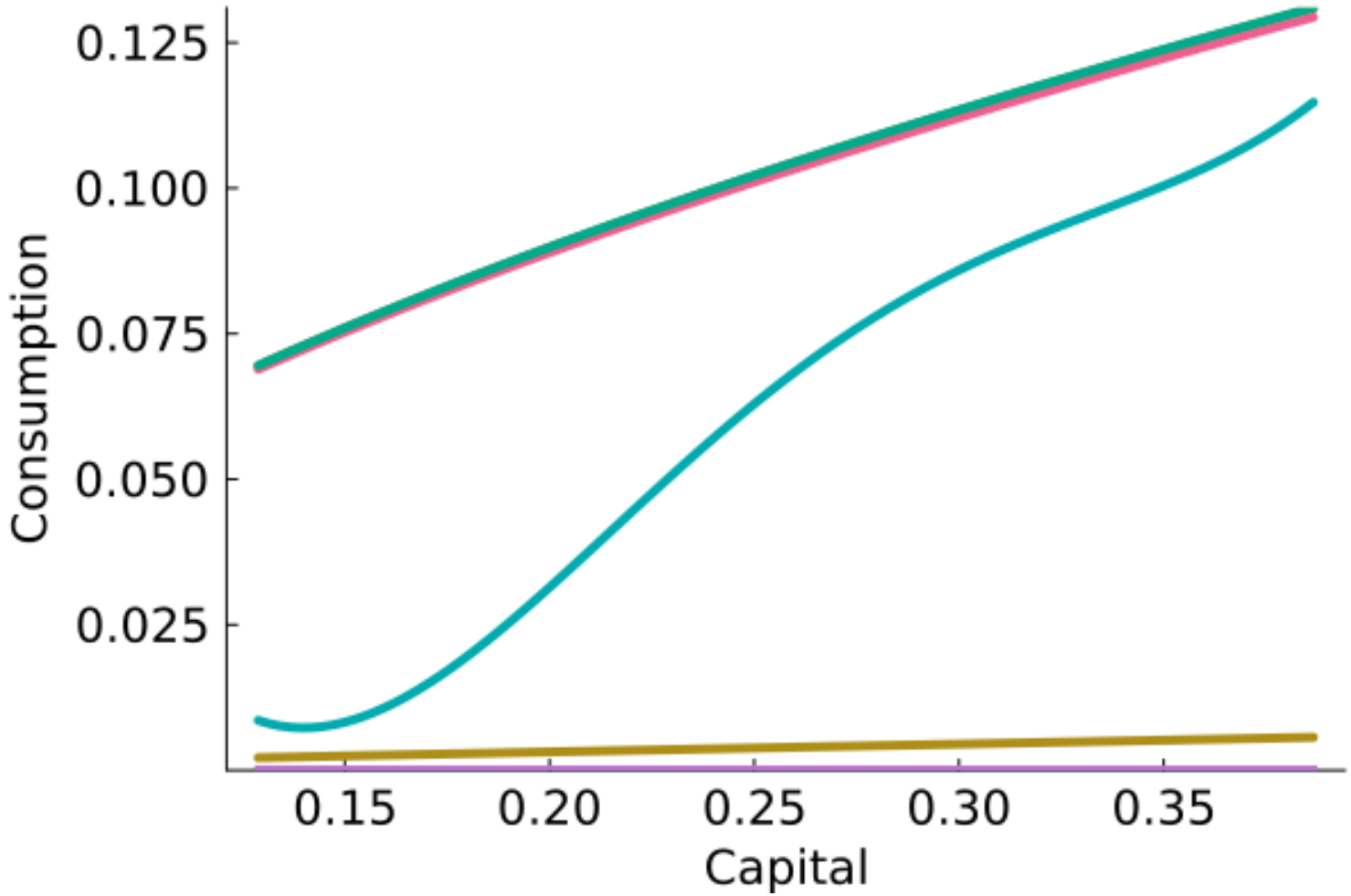
Plot the consumption policy function



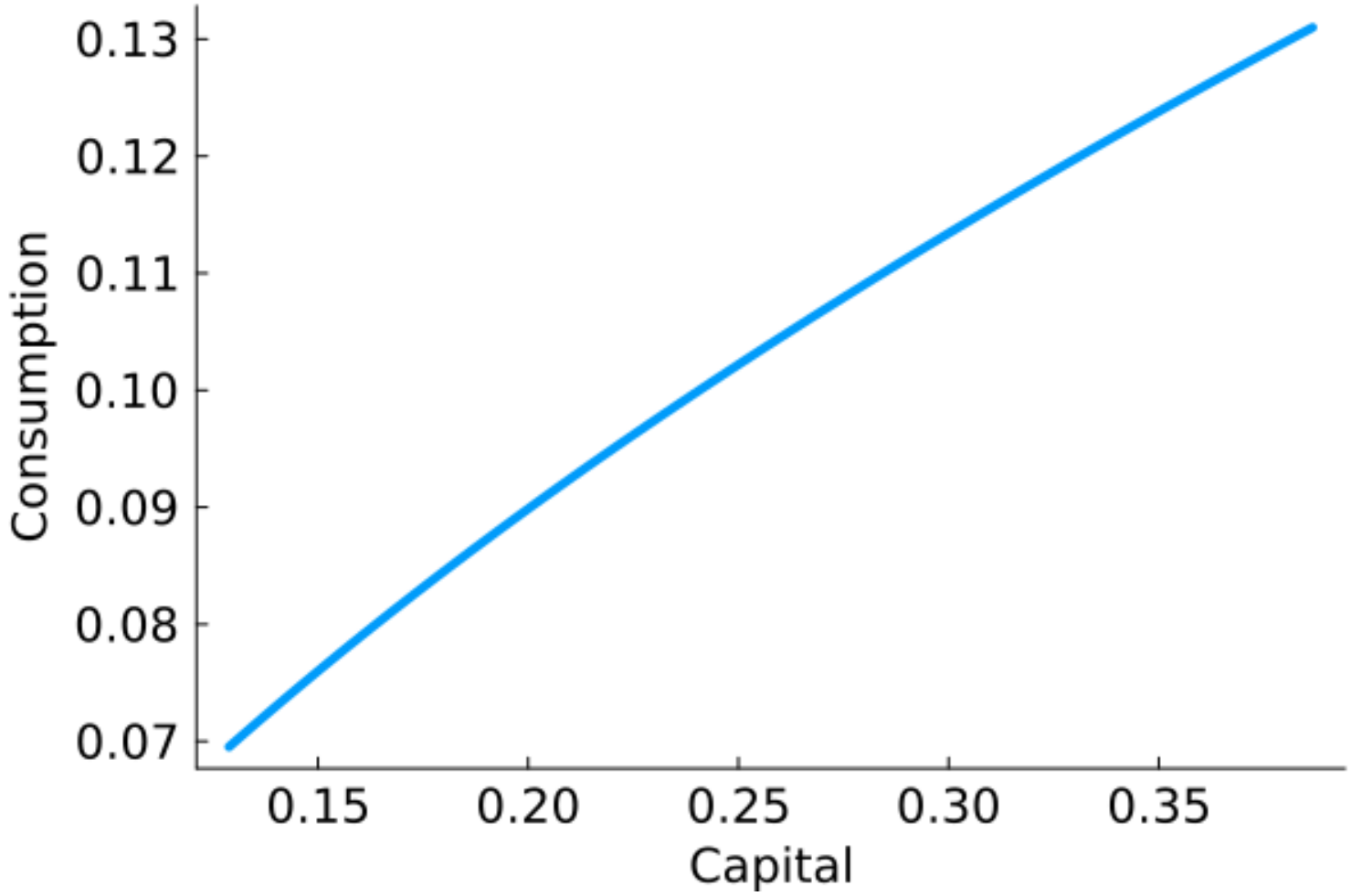
Plot the consumption policy function



Plot the consumption policy function



Plot the final consumption policy function



Now lets try simulating

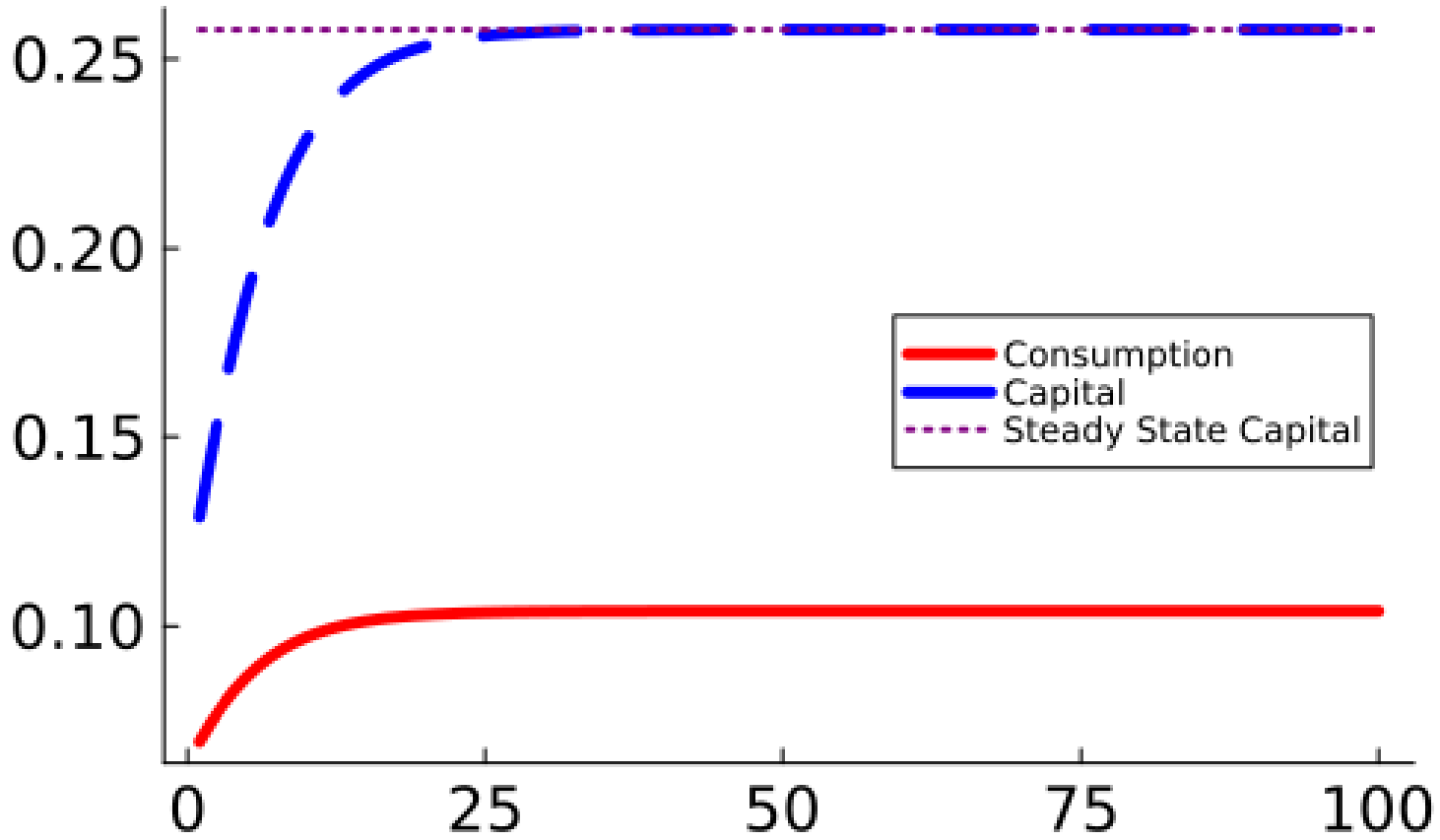
```
function simulate_model(params, solution_coeffs, time_horizon = 100)
    capital_store = zeros(time_horizon + 1)
    consumption_store = zeros(time_horizon)
    capital_store[1] = params.k_0

    for t = 1:time_horizon
        capital = capital_store[t]
        consumption_store[t] = consumption_euler(params, capital, solution_coeffs)
        capital_store[t+1] = capital^params.alpha - consumption_store[t]
    end

    return consumption_store, capital_store

end
```

```
## simulate_model (generic function with 2 methods)
```

Now lets try simulating

```
time_horizon = 100;
consumption, capital = simulate_model(params_fpi, solution_coeffs, time_horizon);
plot(1:time_horizon, consumption, color = :red, linewidth = 4.0, tickfontsize = 14, guidefontsi
plot!(1:time_horizon, capital[1:end-1], color = :blue, linewidth = 4.0, linestyle = :dash, labe
plot!(1:time_horizon, params_fpi.steady_state*ones(time_horizon), color = :purple, linewidth = 
```

Now lets try simulating



Time iteration

Method 3: Time iteration

In time iteration we approximate the *policy function* with some flexible functional form $\Psi(k_t; b)$ where b is a vector of coefficients

Method 3: Time iteration

In time iteration we approximate the *policy function* with some flexible functional form $\Psi(k_t; b)$ where b is a vector of coefficients

The difference vs FPI is we use root-finding techniques on our n node collocation grid where we search for the **scalar** $c^{(p+1)}(k_t)$ that solves

$$u'(c^{(p+1)}(k_t^j)) = \beta u'(C^{(p)}(f(k_t^j) - c^{(p+1)}(k_t^j)))f'(f(k_t^i) - c^{(p+1)}(k_t^j))$$

for $j = 1, \ldots, N$

Method 3: Time iteration

$$u'(c^{(p+1)}(k_t^j)) = \beta u'(C^{(p)}(f(k_t^j) - c^{(p+1)}(k_t^j)))f'(f(k_t^i) - c^{(p+1)}(k_t^j))$$

$C^{(p)}()$ is our current approximation to the policy function, and we are searching for a **scalar** $c^{(p+1)}(k_t^j)$, given our collocation node k_t^j, that solves the Euler equation root-finding problem

Method 3: Time iteration

$$u'(c^{(p+1)}(k_t^j)) = \beta u'(C^{(p)}(f(k_t^j) - c^{(p+1)}(k_t^j)))f'(f(k_t^i) - c^{(p+1)}(k_t^j))$$

$C^{(p)}()$ is our current approximation to the policy function, and we are searching for a **scalar** $c^{(p+1)}(k_t^j)$, given our collocation node k_t^j, that solves the Euler equation root-finding problem

In the Euler equation $c^{(p+1)}$ corresponds to today's policy function while $C^{(p)}$ corresponds to tomorrow's policy function: we are searching for today's policy that satisfies the Euler equation

Method 3: Time iteration

Step 1: Select the number of collocation points in each dimension and the domain of the approximation space

Method 3: Time iteration

Step 1: Select the number of collocation points in each dimension and the domain of the approximation space

Step 2: Select an initial vector of coefficients b_0 with the same number of elements as the collocation grid, and initial guesses for consumption for the root-finder

Method 3: Time iteration

Step 1: Select the number of collocation points in each dimension and the domain of the approximation space

Step 2: Select an initial vector of coefficients b_0 with the same number of elements as the collocation grid, and initial guesses for consumption for the root-finder

Step 3: Select a rule for convergence

Method 3: Time iteration

Step 1: Select the number of collocation points in each dimension and the domain of the approximation space

Step 2: Select an initial vector of coefficients b_0 with the same number of elements as the collocation grid, and initial guesses for consumption for the root-finder

Step 3: Select a rule for convergence

Step 4: Construct the grid and basis matrix

Method 3: Time iteration

Step 5: While convergence criterion $>$ tolerance **(outer loop [fixed point])**

- Start iteration p
- For each grid point **(inner loop [rootfinding])**
  - Substitute $C(k_{t+1}^j; b^{(p)})$ in for $t+1$ consumption
  - Recover the $c^{(p+1)}(k_t^j) \in \mathbb{R}$ scalar values that satisfy the equation
- Fit the polynomial to the values and recover a new vector of coefficients $\hat{b}^{(p+1)}$
- Compute the vector of coefficients $b^{(p+1)}$ for iteration $p+1$ by $b^{(p+1)} = (1-\gamma)b^{(p)} + \gamma\hat{b}^{(p+1)}$ where $\gamma \in (0,1)$ (damping)

Step 6: Error check your approximation

Step 1: Select the number of points and domain

Put everything in a **named tuple** to make passing things easier

```
using LinearAlgebra, Optim, Plots, Roots
params_ti = (alpha = 0.75, beta = 0.95, eta = 2, damp = 0.7,
                steady_state = (0.75*0.95)^(1/(1-0.75)), k_0 = (0.75*0.95)^(1/(1-0.75))*0.5,
                capital_upper = (0.75*0.95)^(1/(1-0.75))*1.5, capital_lower = (0.75*0.95)^(1/(1-
                num_points = 6, tolerance = 0.00001)
```

```
## (alpha = 0.75, beta = 0.95, eta = 2, damp = 0.7, steady_state = 0.25771486816406236, k_0 = 0.1288574
```

Step 2: Select an initial vector of coefficients b_0

In some cases you might have a good guess (e.g. increasing and concave so you know the second value is positive, third value is negative, rest maybe set to zero)

Step 2: Select an initial vector of coefficients b_0

In some cases you might have a good guess (e.g. increasing and concave so you know the second value is positive, third value is negative, rest maybe set to zero)

Other cases you might not, guessing zeros effectively turns the initial iteration into a static problem, the second iteration into a 2 period problem, and so on

Step 2: Select an initial vector of coefficients b_0

In some cases you might have a good guess (e.g. increasing and concave so you know the second value is positive, third value is negative, rest maybe set to zero)

Other cases you might not, guessing zeros effectively turns the initial iteration into a static problem, the second iteration into a 2 period problem, and so on

```
coefficients = zeros(params_ti.num_points)
```

```
## 6-element Vector{Float64}:
##  0.0
##  0.0
##  0.0
##  0.0
##  0.0
##  0.0
```

Step 3: Select a convergence rule

There's a lot of potential options here to determine convergence of the function

Step 3: Select a convergence rule

There's a lot of potential options here to determine convergence of the function

Relative or absolute change? Or both?

Step 3: Select a convergence rule

There's a lot of potential options here to determine convergence of the function

Relative or absolute change? Or both?

Change in the value function? Change in the policy function?

Step 3: Select a convergence rule

There's a lot of potential options here to determine convergence of the function

Relative or absolute change? Or both?

Change in the value function? Change in the policy function?

Which norm?

Step 3: Select a convergence rule

There's a lot of potential options here to determine convergence of the function

Relative or absolute change? Or both?

Change in the value function? Change in the policy function?

Which norm?

Our rule for class: convergence is when the maximum relative change in value on the grid is < 0.001%

Step 4: Construct the grid and basis matrix

The function `cheb_nodes` constructs the grid on $[-1, 1]$

Step 4: Construct the grid and basis matrix

The function `cheb_nodes` constructs the grid on $[-1, 1]$

$$x_k = cos\left(\frac{2k-1}{2n}\pi\right),\ k = 1, \ldots, n$$

Step 4: Construct the grid and basis matrix

The function `cheb_nodes` constructs the grid on $[-1, 1]$

$$x_k = cos\left(\frac{2k-1}{2n}\pi\right),\ k = 1, \ldots, n$$

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

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

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

```
## 6-element Vector{Float64}:
##   0.9659258262890683
##   0.7071067811865476
##   0.25881904510252096
##  -0.25881904510252063
##  -0.7071067811865475
```

Step 4: Construct the grid and basis matrix

But we need to expand the grid from $[-1, 1]$ to our actual capital domain

Step 4: Construct the grid and basis matrix

But we need to expand the grid from $[-1, 1]$ to our actual capital domain

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

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

```
capital_grid = expand_grid(grid, params_ti)
```

```
## 6-element Vector{Float64}:
##  0.3821815916532374
##  0.3488308336097651
##  0.2910656262075347
##  0.22436411012059004
##  0.16659890271835956
##  0.13324814467488724
```

Step 4: Construct the grid and basis matrix

Make the inverse function to shrink from capital to Chebyshev space

```
shrink_grid(capital)
```

Step 4: Construct the grid and basis matrix

Make the inverse function to shrink from capital to Chebyshev space

`shrink_grid(capital)`

```
shrink_grid(capital) =
  2*(capital - params_ti.capital_lower)/(params_ti.capital_upper - params_ti.capital_lower) - 1
shrink_grid.(capital_grid)
```

```
## 6-element Vector{Float64}:
##   0.9659258262890678
##   0.7071067811865472
##   0.25881904510252096
##  -0.2588190451025205
##  -0.7071067811865475
##  -0.9659258262890683
```

`shrink_grid` will inherit `params_ti` from wrapper functions

Step 4: Construct the grid and basis matrix

Use `cheb_polys` to construct the basis matrix

```
# 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;
```

Step 4: Construct the grid and basis matrix

In our basis matrix, rows are grid points, columns are basis functions, make a function `construct_basis_matrix(grid, params)` that makes the basis matrix for some arbitrary grid of points

```
construct_basis_matrix(grid, params_ti) = hcat([cheb_polys.(shrink_grid.(grid), n) for n = 0:par
basis_matrix = construct_basis_matrix(capital_grid, params_ti)
```

```
## 6×6 Matrix{Float64}:
##  1.0   0.965926   0.866025      0.707107   0.5   0.258819
##  1.0   0.707107  -7.77156e-16  -0.707107  -1.0  -0.707107
##  1.0   0.258819  -0.866025     -0.707107   0.5   0.965926
##  1.0  -0.258819  -0.866025      0.707107   0.5  -0.965926
##  1.0  -0.707107  -2.22045e-16   0.707107  -1.0   0.707107
##  1.0  -0.965926   0.866025     -0.707107   0.5  -0.258819
```

Step 4: Pre-invert your basis matrix

Pro tip: you will be using the *exact same* basis matrix in each loop iteration to recover the coefficients: just pre-invert it to save time because inverting the same matrix every loop is costly (especially when large)

```
basis_inverse = basis_matrix \ I # pre-invert
```

```
## 6×6 Matrix{Float64}:
##  0.166667   0.166667      0.166667   0.166667   0.166667      0.166667
##  0.321975   0.235702      0.086273  -0.086273  -0.235702     -0.321975
##  0.288675  -1.52278e-15  -0.288675  -0.288675   2.62176e-16   0.288675
##  0.235702  -0.235702     -0.235702   0.235702   0.235702     -0.235702
##  0.166667  -0.333333      0.166667   0.166667  -0.333333      0.166667
##  0.086273  -0.235702      0.321975  -0.321975   0.235702     -0.086273
```

Pre-Step 5: Evaluate the policy function

We need to make a function `eval_policy_function(coefficients, capital, params_ti)` that lets us evaluate the policy function given a vector of coefficients `coefficients`, a vector of capital nodes `capital`, and the model parameters `params_ti`

Pre-Step 5: Evaluate the policy function

We need to make a function `eval_policy_function(coefficients, capital, params_ti)` that lets us evaluate the policy function given a vector of coefficients `coefficients`, a vector of capital nodes `capital`, and the model parameters `params_ti`

It needs to:

1. Scale capital back into $[-1, 1]$ (the domain of the Chebyshev polynomials)
2. Use the coefficients and Chebyshev polynomials to evaluate the value function

Pre-Step 5: Evaluate the policy function

```
# evaluates V on the [-1,1]-equivalent grid
eval_policy_function(coefficients, capital, params_ti) =
    construct_basis_matrix(capital, params_ti) * coefficients;
```

Step 5: Loop

Construct a function `loop_grid_ti(params_ti, capital_grid, coefficients)` that loops over the grid points and solves the Euler given $\Psi(x; b^{(p)})$

Step 5: Loop

```
function loop_grid_ti(params_ti, capital_grid, coefficients)
    consumption = similar(coefficients)
    for (iteration, capital) in enumerate(capital_grid)
        function consumption_euler(consumption_guess)
            capital_next = capital^params_ti.alpha - consumption_guess
            # Next period consumption based on approximating policy function
            consumption_next = eval_policy_function(coefficients, capital_next, params_ti)[1]
            consumption_next = max(1e-10, consumption_next)
            # Organize Euler so it's g(c,k) = 0
            euler_error = consumption_guess^(-params_ti.eta) /
                (params_ti.beta*consumption_next^(-params_ti.eta)*params_ti.alpha*(capital_next)
            return euler_error
        end
        # Search over consumption such that Euler = 0
        consumption[iteration] = fzero(consumption_euler, 0., capital)
    end
    return consumption
end
```

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

Step 5: Loop

Construct a function `solve_ti(params_fpi, basis_inverse, capital_grid, coefficients)` that iterates on `loop_grid_ti` and solves for the coefficient vector b until the scalar c values on the grid converge

Step 5: Loop

```
function solve_ti(params_ti, basis_inverse, capital_grid, coefficients)
    error = 1e10
    iteration = 1
    consumption = similar(coefficients)
    consumption_prev, coefficients_prev = similar(coefficients), similar(coefficients)
    coefficients_store = Vector{Vector}(undef, 1)
    coefficients_store[1] = coefficients
    while error > params_ti.tolerance
        consumption = loop_grid_ti(params_ti, capital_grid, coefficients)
        if iteration > 1
            coefficients = params_ti.damp*(basis_inverse*consumption) + (1 - params_ti.damp)*coe
        else
            coefficients = basis_inverse*consumption
        end
        error = maximum(abs.((consumption - consumption_prev)./(consumption_prev)))
        consumption_prev, coefficients_prev = deepcopy(consumption), deepcopy(coefficients)
        if mod(iteration, 5) == 0
            println("Maximum Error of $(error) on iteration $(iteration).")
            append!(coefficients_store, [coefficients])
        end
        iteration += 1
```

Step 5: Loop

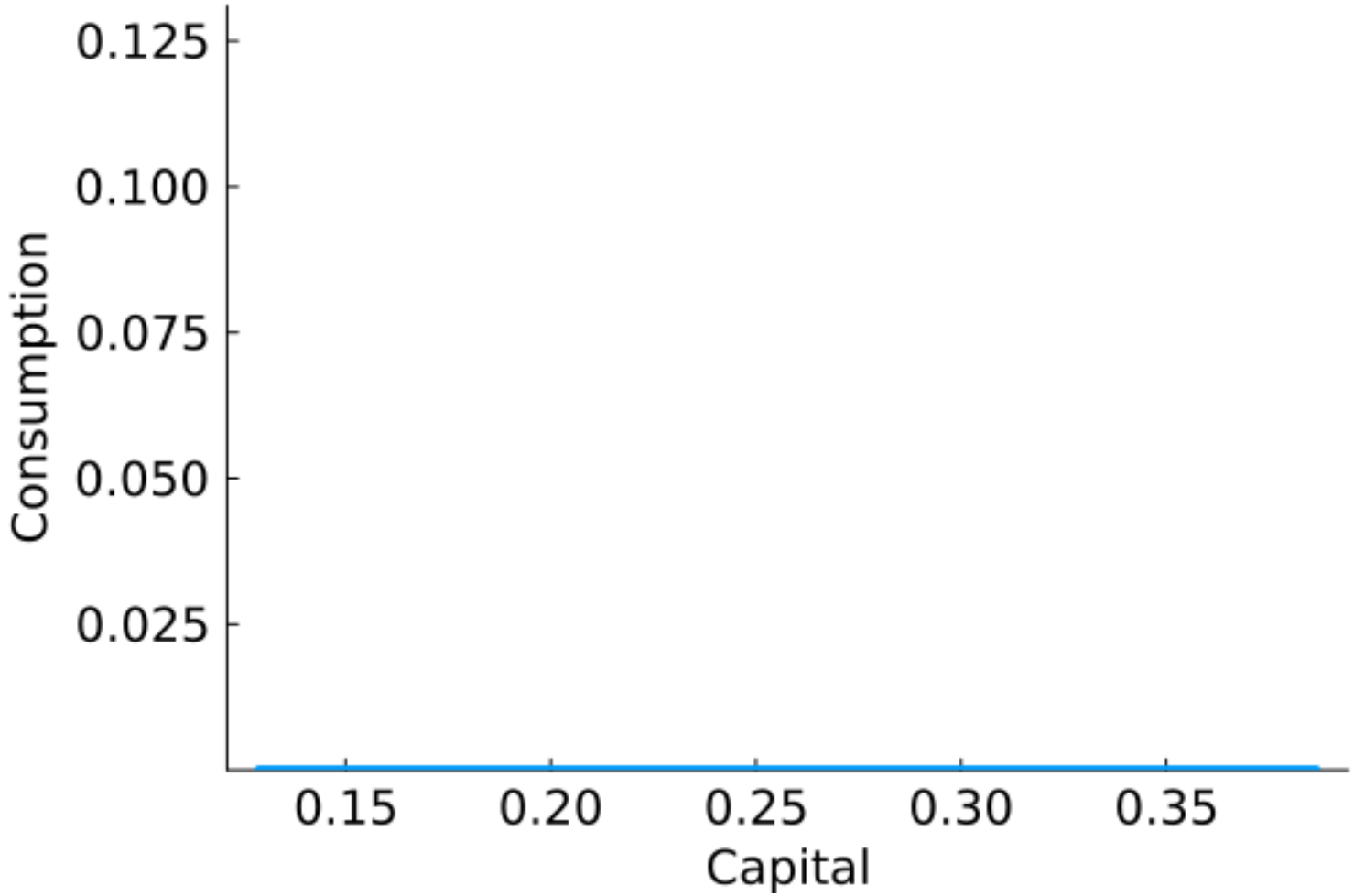
```
solution_coeffs, consumption, intermediate_coefficients =
    solve_ti(params_ti, basis_inverse, capital_grid, coefficients)
```

```
## Maximum Error of 0.25321486014602274 on iteration 5.
## Maximum Error of 0.5080049627820759 on iteration 10.
## Maximum Error of 0.437979808520151 on iteration 15.
## Maximum Error of 0.3028938891677158 on iteration 20.
## Maximum Error of 0.37071932036765914 on iteration 25.
## Maximum Error of 0.38905423220148744 on iteration 30.
## Maximum Error of 0.3603910536945966 on iteration 35.
## Maximum Error of 0.3351549146984757 on iteration 40.
## Maximum Error of 0.3591346757111148 on iteration 45.
## Maximum Error of 0.3585490158260728 on iteration 50.
## Maximum Error of 0.34330698497556356 on iteration 55.
## Maximum Error of 0.3276713699071389 on iteration 60.
## Maximum Error of 0.29460121133680833 on iteration 65.
## Maximum Error of 0.20990288276886732 on iteration 70.
## Maximum Error of 0.11884185187600733 on iteration 75.
## Maximum Error of 0.060469194796897 on iteration 80.
## Maximum Error of 0.02995709287567328 on iteration 85.
```

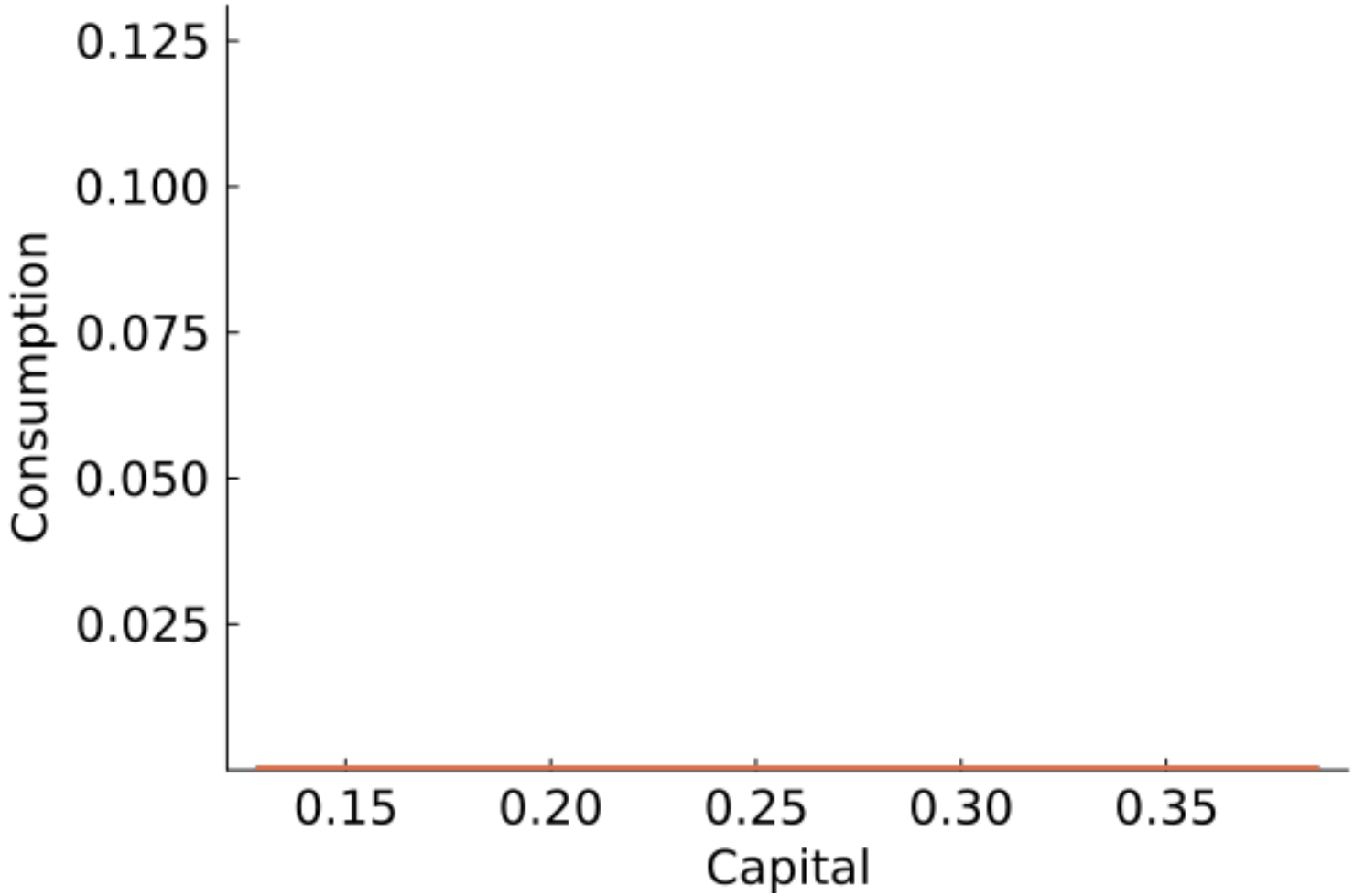
Now lets plot our solutions

```
capital_levels = range(params_ti.capital_lower, params_ti.capital_upper, length = 100);
eval_points = shrink_grid.(capital_levels);
solution = similar(intermediate_coefficients);

for (iteration, coeffs) in enumerate(intermediate_coefficients)
    solution[iteration] = [coeffs' * [cheb_polys.(capital, n) for n = 0:params_ti.num_points - 1
end
```

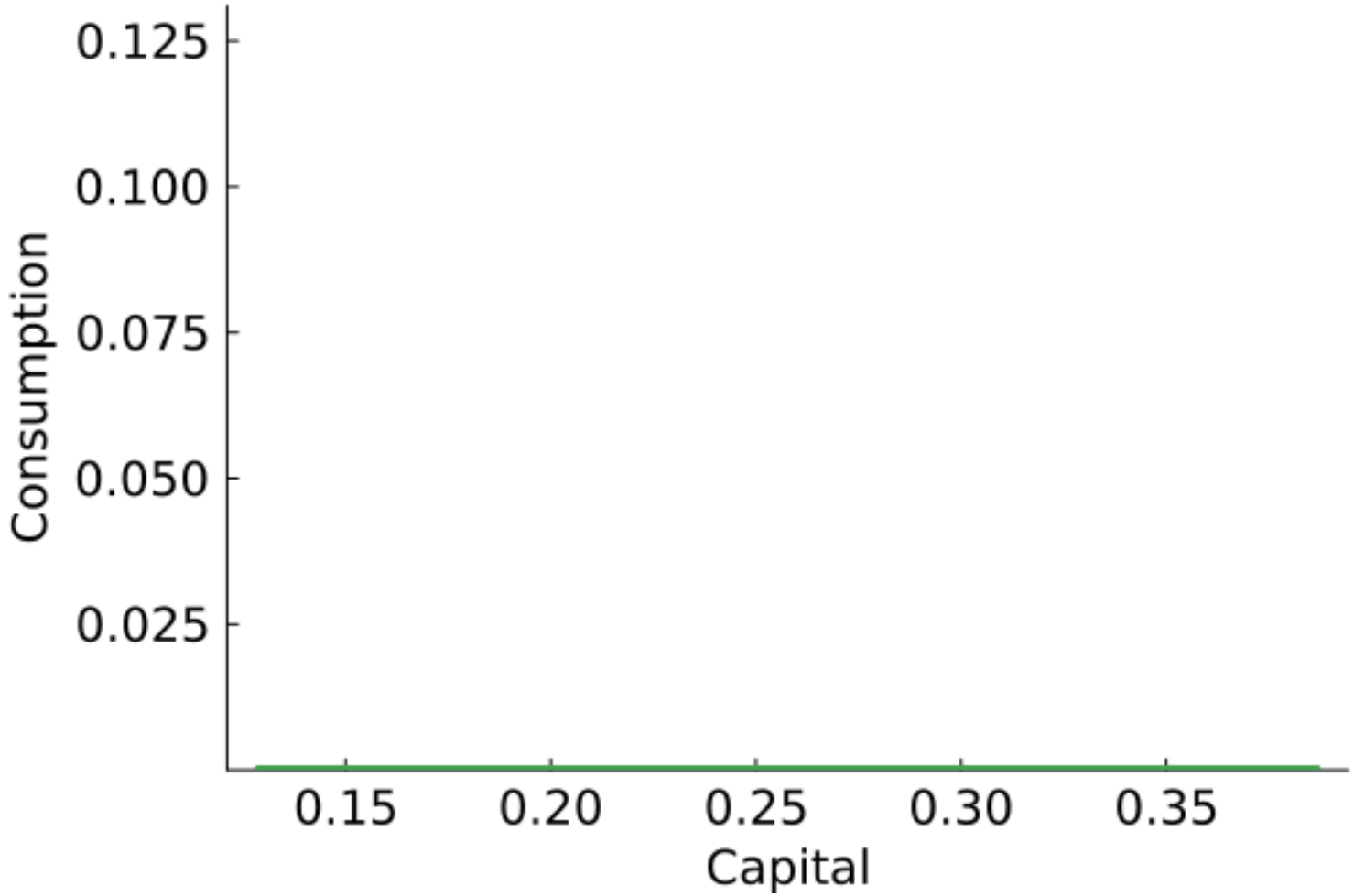
Plot the consumption policy function



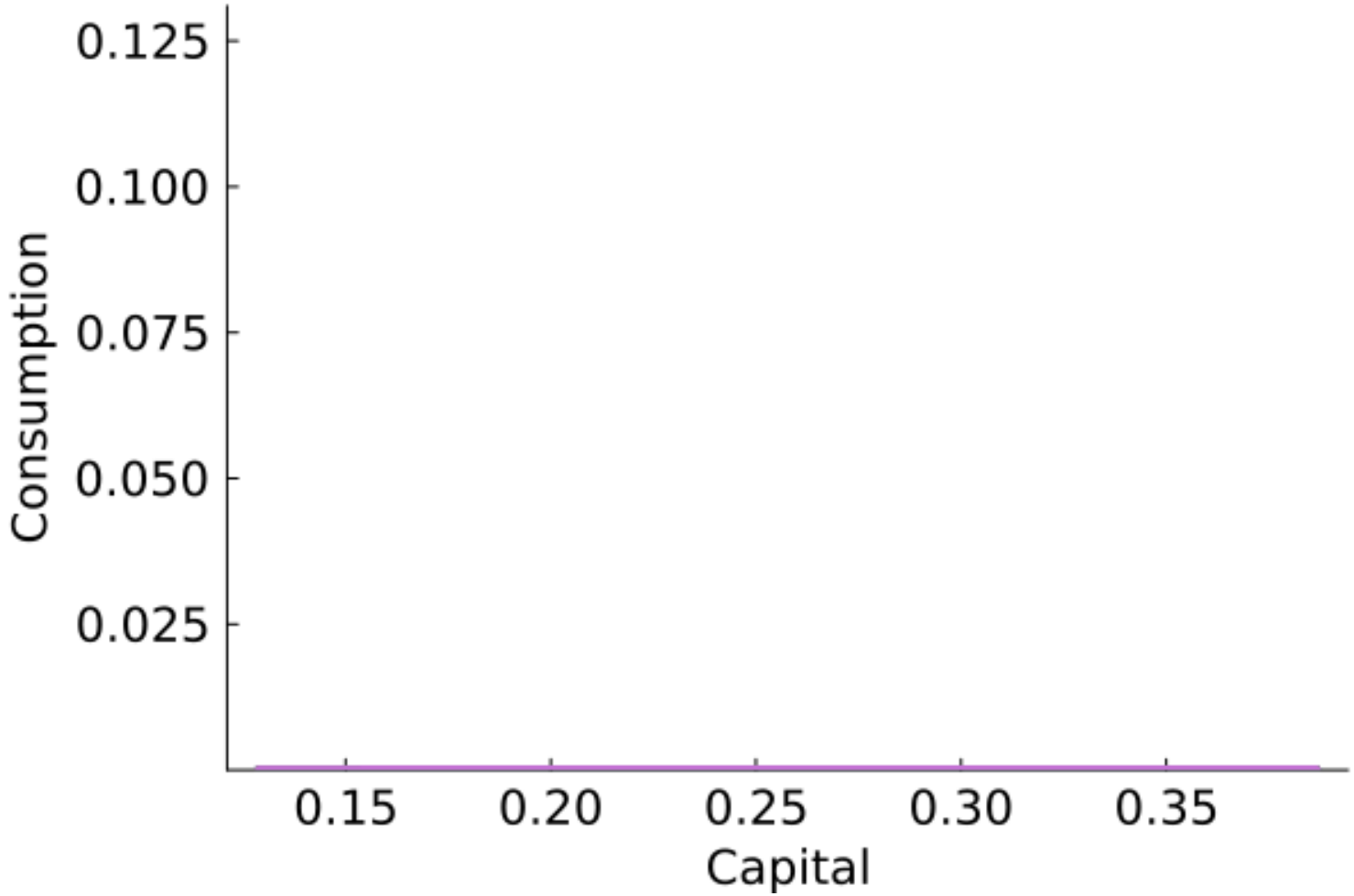
Plot the consumption policy function



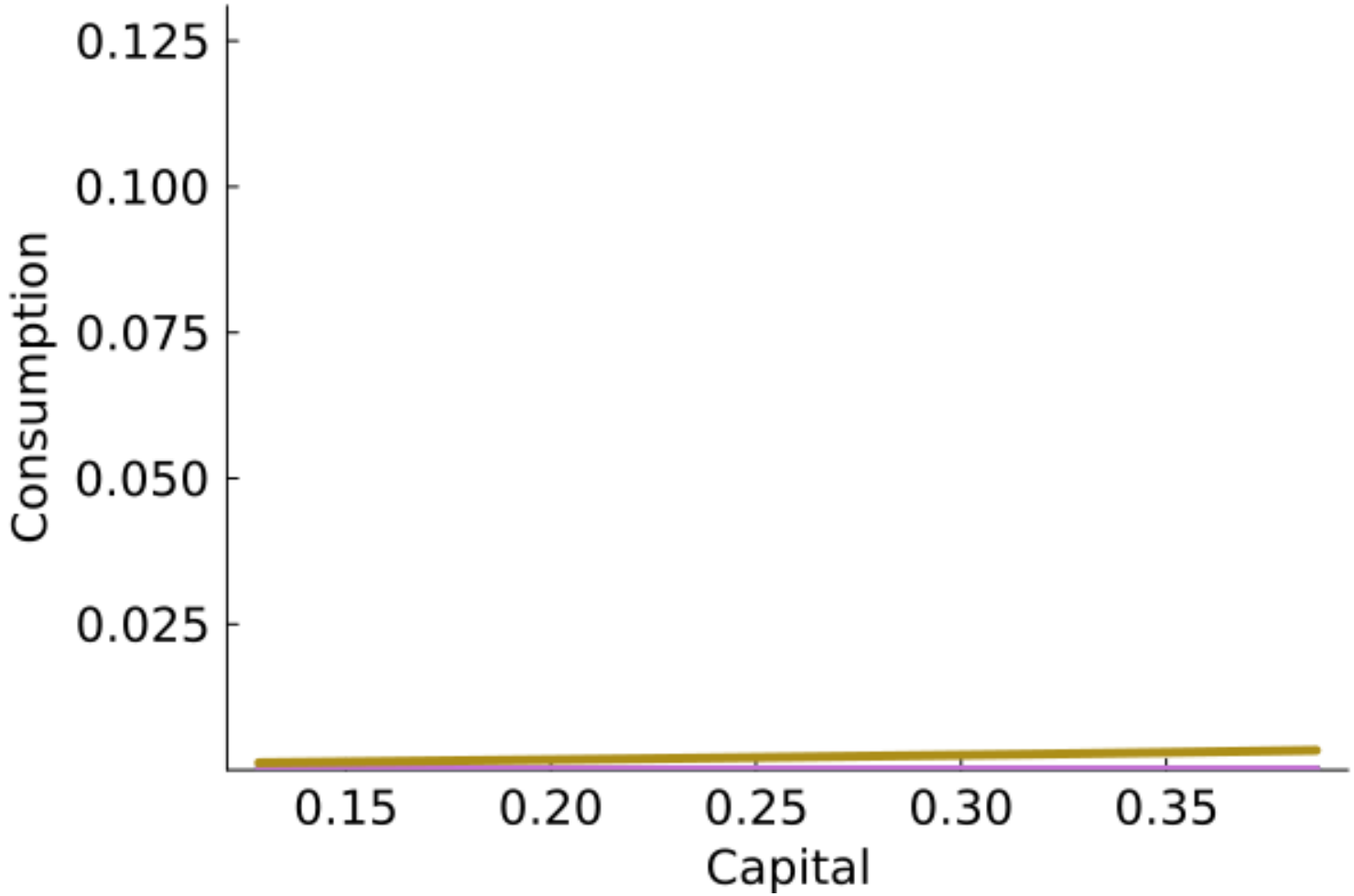
Plot the consumption policy function



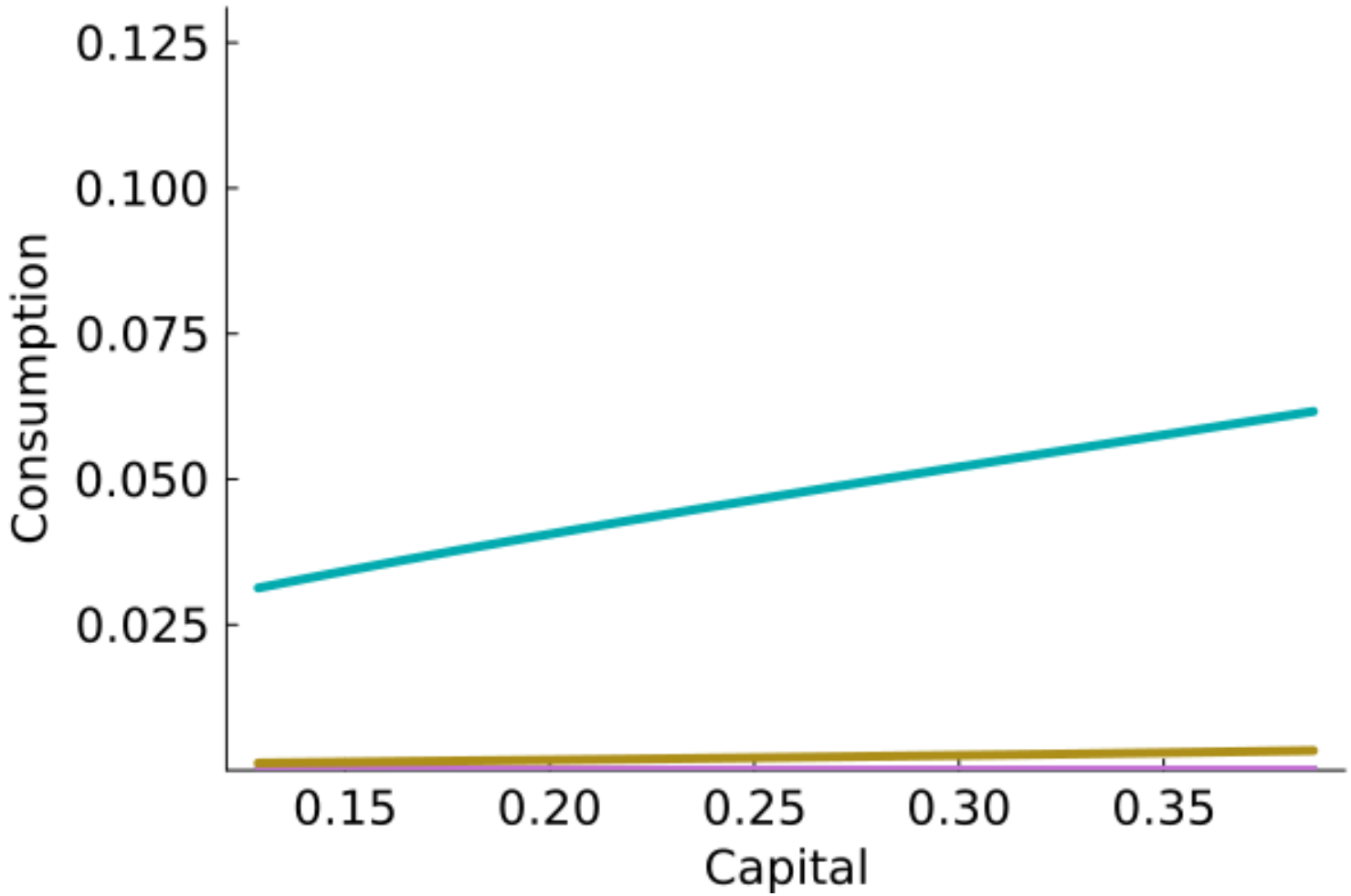
Plot the consumption policy function



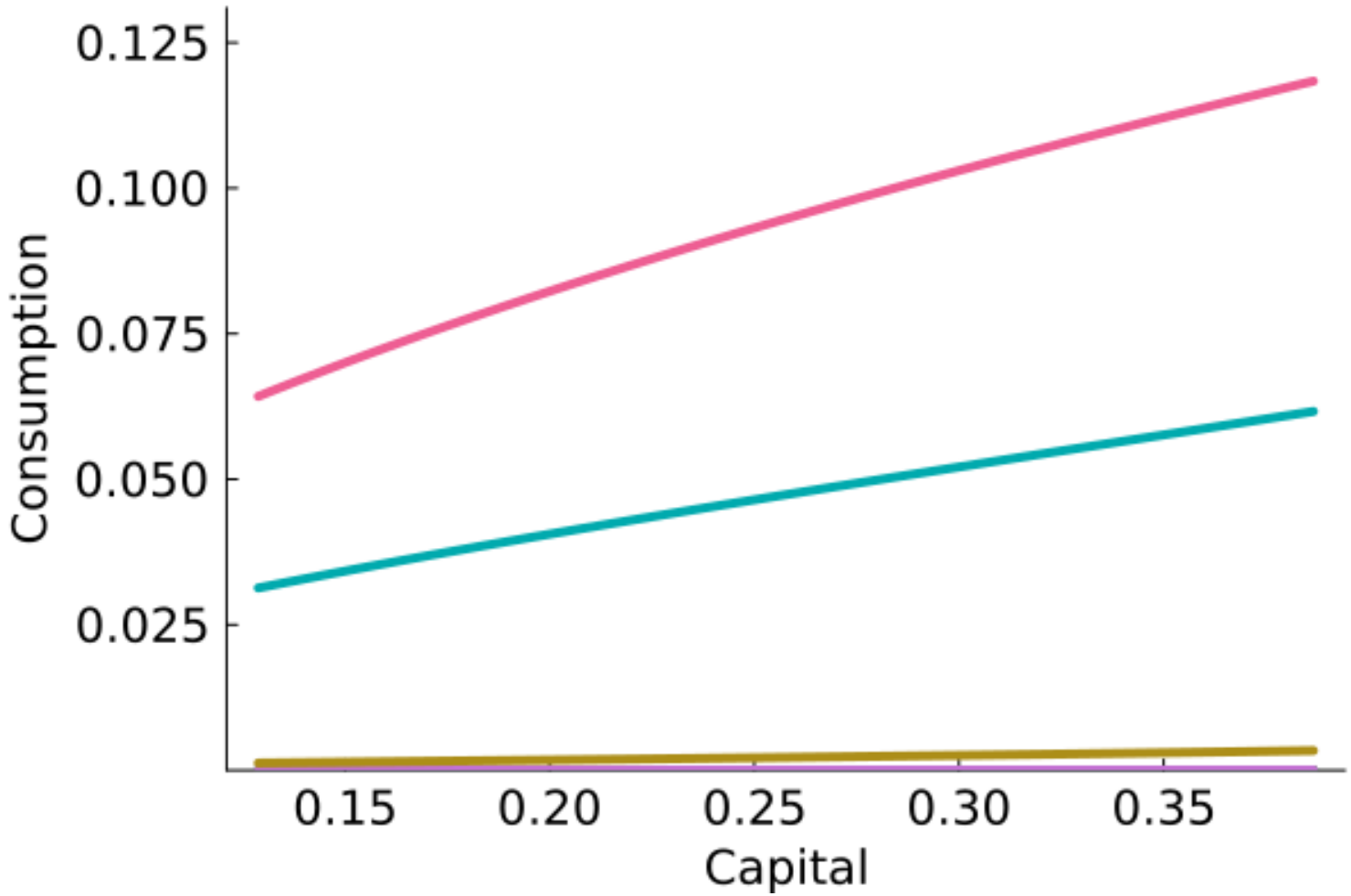
Plot the consumption policy function



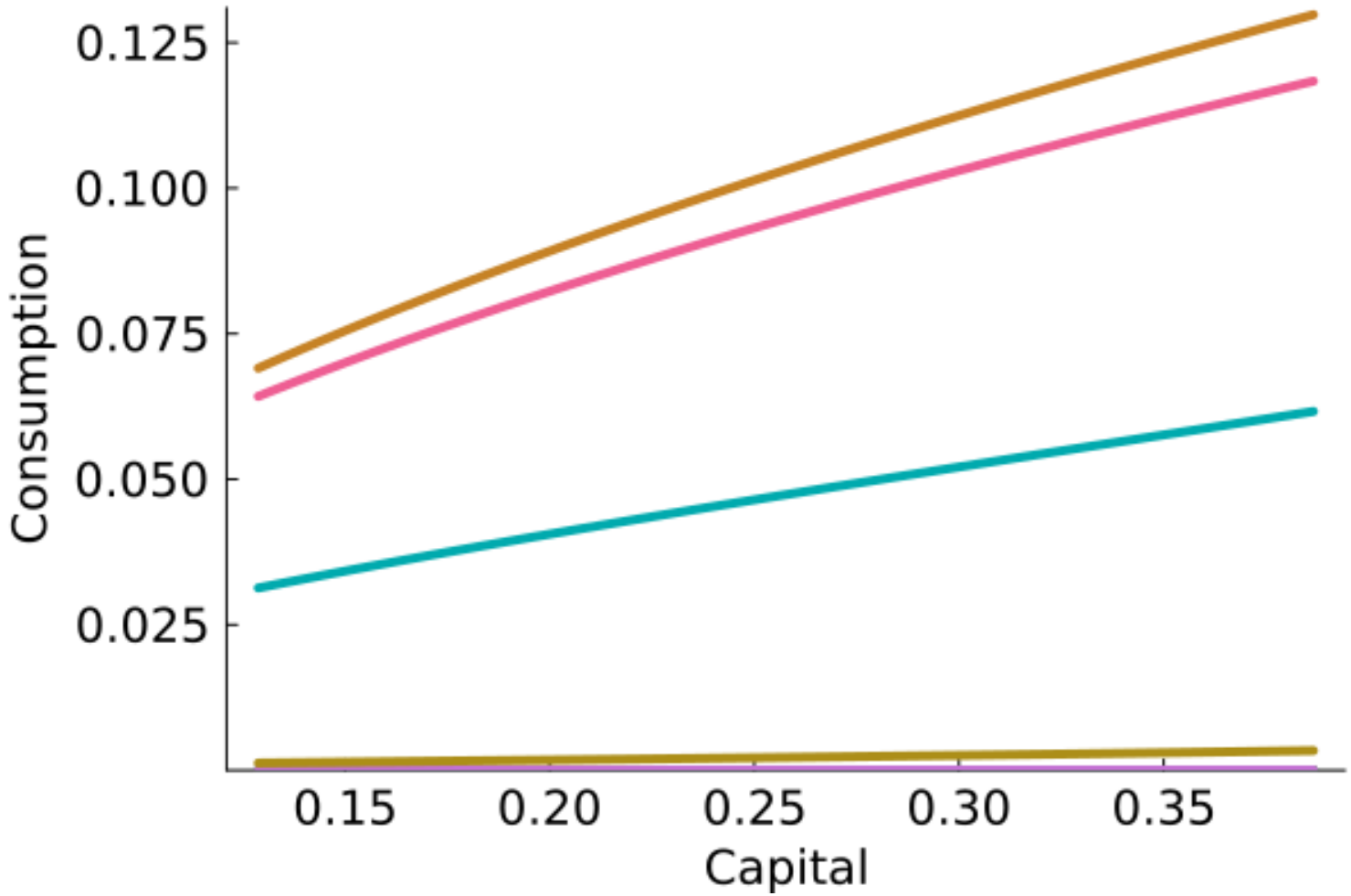
Plot the consumption policy function



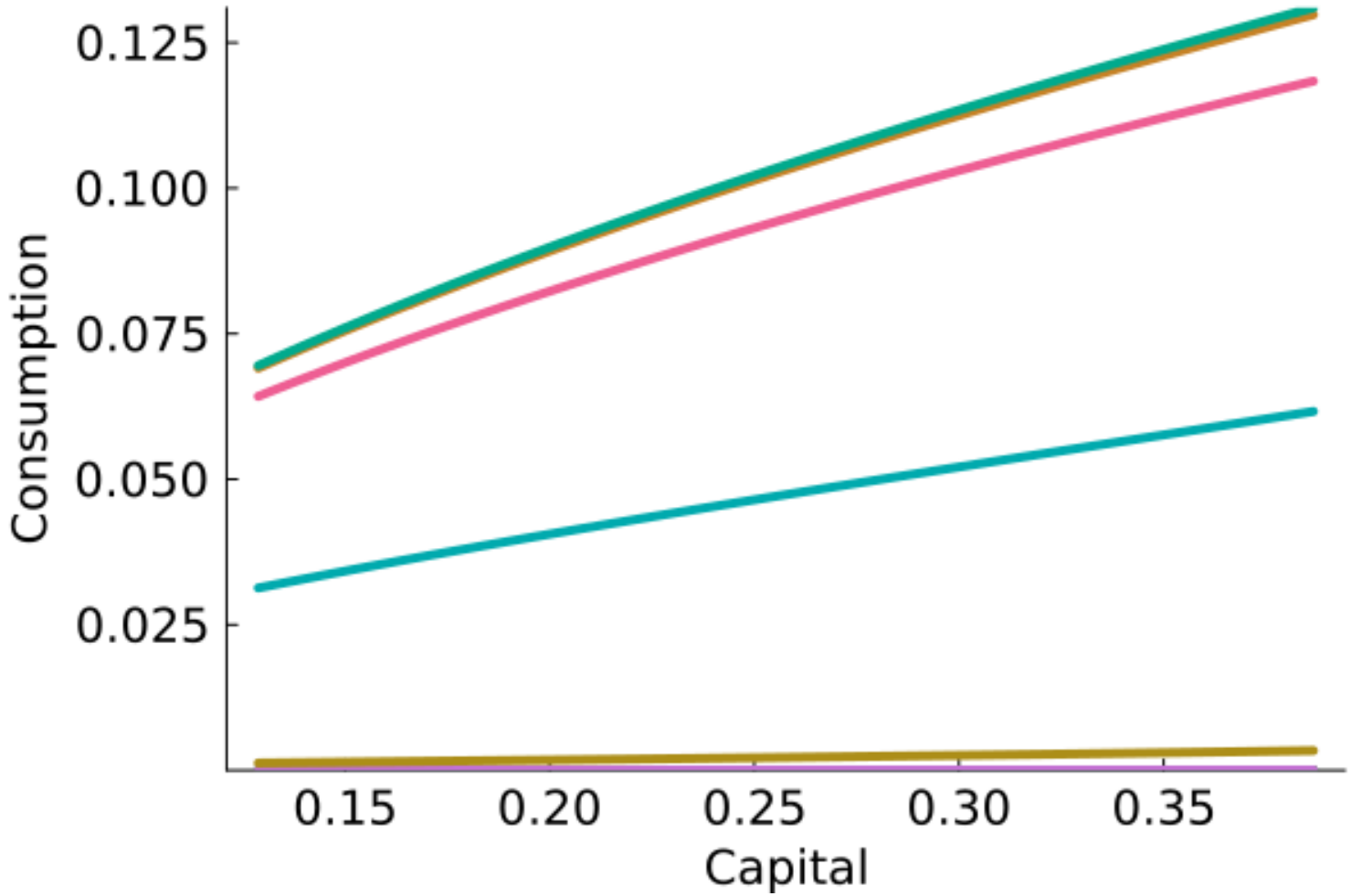
Plot the consumption policy function



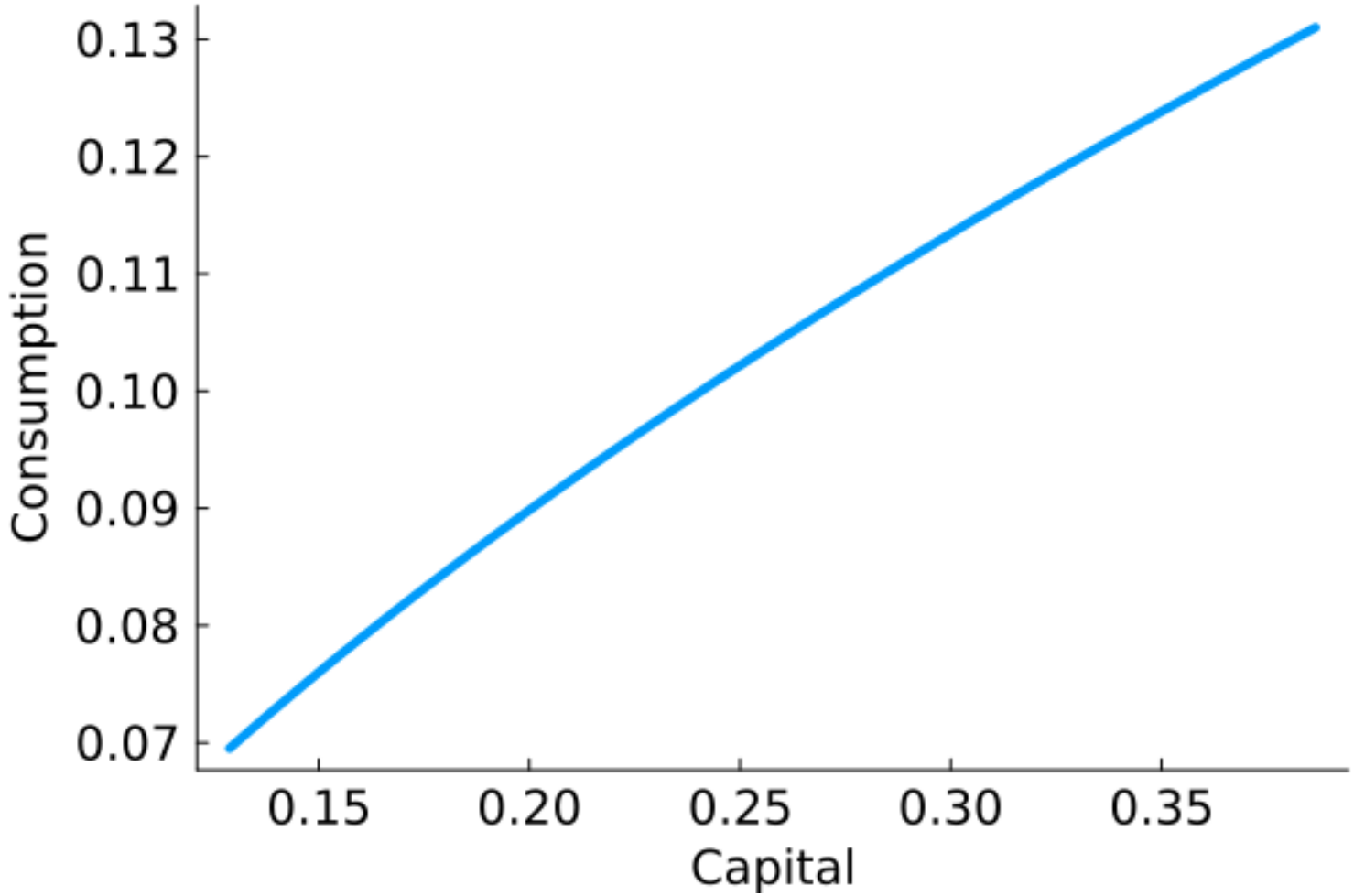
Plot the consumption policy function



Plot the consumption policy function



Plot the final consumption policy function



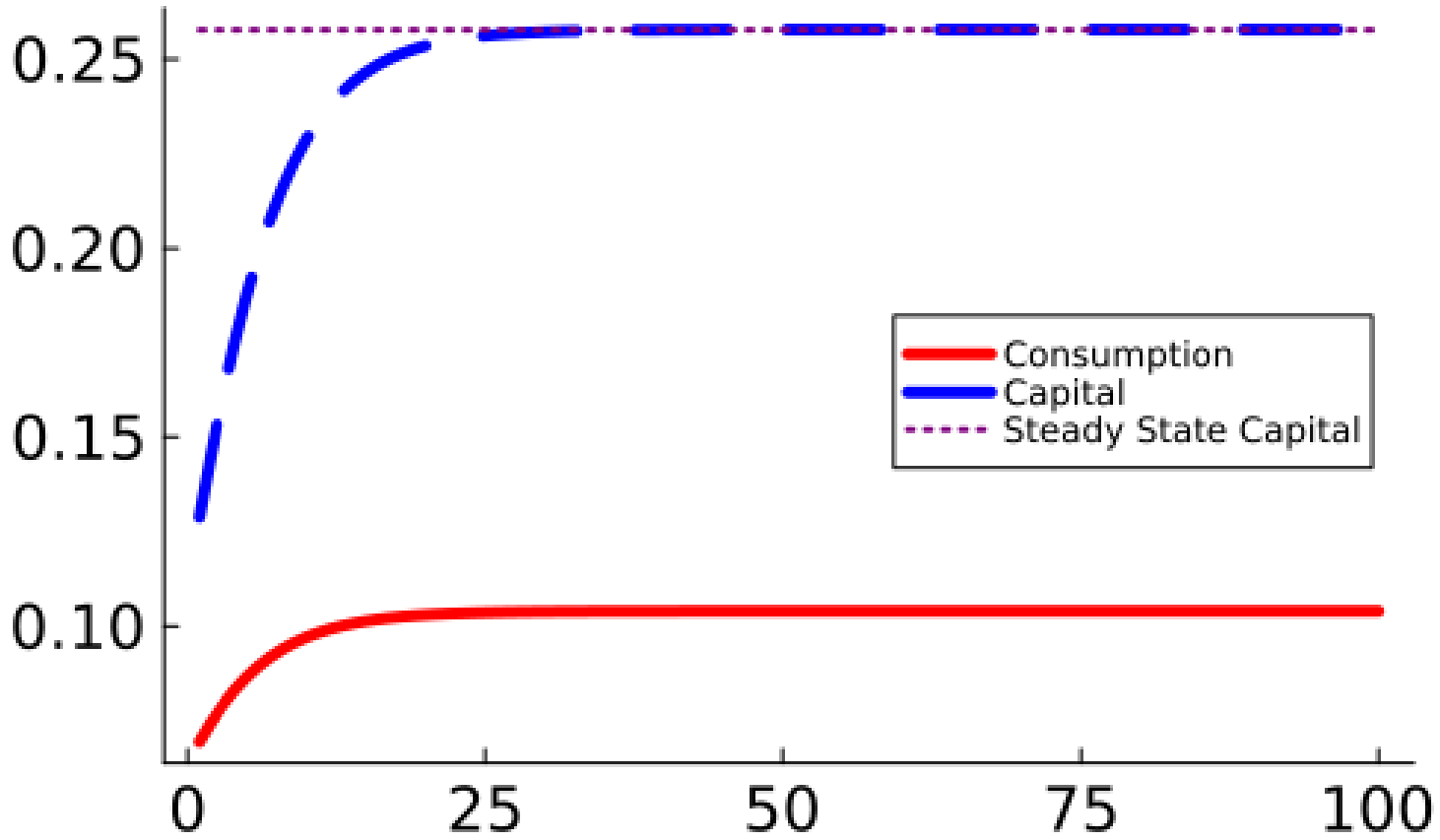
Now lets try simulating

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

    for t = 1:time_horizon
        capital = capital_store[t]
        consumption_store[t] = eval_policy_function(solution_coeffs, capital, params_ti)[1]
        capital_store[t+1] = capital^params_ti.alpha - consumption_store[t]
    end

    return consumption_store, capital_store
end;
```

Now lets try simulating

```
time_horizon = 100;
consumption, capital = simulate_model(params_ti, solution_coeffs, time_horizon);
plot(1:time_horizon, consumption, color = :red, linewidth = 4.0, tickfontsize = 14, guidefontsi
plot!(1:time_horizon, capital[1:end-1], color = :blue, linewidth = 4.0, linestyle = :dash, labe
plot!(1:time_horizon, params_ti.steady_state*ones(time_horizon), color = :purple, linewidth = 2
```

Now lets try simulating



Discretization

A short overview of discretization + VFI

When we use discretization methods we create a grid on our state space, typically evenly spaced

A short overview of discretization + VFI

When we use discretization methods we create a grid on our state space, typically evenly spaced

This becomes our **actual** state space, not just collocation points

A short overview of discretization + VFI

When we use discretization methods we create a grid on our state space, typically evenly spaced

This becomes our **actual** state space, not just collocation points

How does it work?

A short overview of discretization + VFI

The discretized state space implies a discretized control space

A short overview of discretization + VFI

The discretized state space implies a discretized control space

If there are only a finite number of states tomorrow conditional on the current state, then there is only a finite number of valid controls

A short overview of discretization + VFI

The discretized state space implies a discretized control space

If there are only a finite number of states tomorrow conditional on the current state, then there is only a finite number of valid controls

This makes solving easy!

A short overview of discretization + VFI

Search over all possible controls today until you find the one that yields the highest value of the RHS of the Bellman: just requires looping and a max operator

A short overview of discretization + VFI

Search over all possible controls today until you find the one that yields the highest value of the RHS of the Bellman: just requires looping and a max operator

The maximized value is the new value of this discretized state

A short overview of discretization + VFI

Search over all possible controls today until you find the one that yields the highest value of the RHS of the Bellman: just requires looping and a max operator

The maximized value is the new value of this discretized state

3 loops now: outer VFI loop, middle capital grid loop, inner consumption loop

Discretizing the state space

```
using LinearAlgebra
using Optim
using Plots
params_dis = (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/
                tolerance = 0.0001, max_iterations = 1000)
```

```
## (alpha = 0.75, beta = 0.95, eta = 2, steady_state = 0.25771486816406236, k_0 = 0.19328615112304676,
```

Discretizing the state space

```
function iterate_value(grid, params)
    grid_size = size(grid, 1)
    V, V_prev = zeros(grid_size, 1), zeros(grid_size, 1)
    V_store = Array{Float64}(undef, grid_size, params.max_iterations)
    max_diff = 1e10
    it = 1
    while max_diff > params.tolerance && it <= params.max_iterations # iterate on the value func
        for (iteration, grid_point) in enumerate(grid) # iterate across the capital grid
            # possible consumption values (output + remaining capital - capital next period)
            c_vec = grid_point.^params.alpha .- grid
            value_max = -Inf
            # find consumption that maximizes the right hand side of the Bellman, search over or
            for (it_inner, consumption) in enumerate(c_vec[c_vec .> 0]) # iterate across possibl
                value_temp = consumption^(1 - params.eta)/(1 - params.eta) + params.beta*V[it_in
                value_max = max(value_temp, value_max)
            end
            V[iteration] = value_max
        end
        max_diff = maximum(abs.(V .- V_prev))
        if mod(it,10) == 0
            println("Current maximum value difference at iteration $it is $max_diff.")
```

Discretizing the state space

```
        max_diff = maximum(abs.((V .- V_prev)./V_prev))
        if mod(it,10) == 0
            println("Current maximum value difference at iteration $it is $max_diff.")
        end
        V_prev = copy(V)
        V_store[:,it] = V
        if it == params.max_iterations
            println("Hit maximum iterations")
            break
        end
        it += 1
    end
    V_store = V_store[:, 1:it-1]
    return V, V_store
end
```

Discretizing the state space

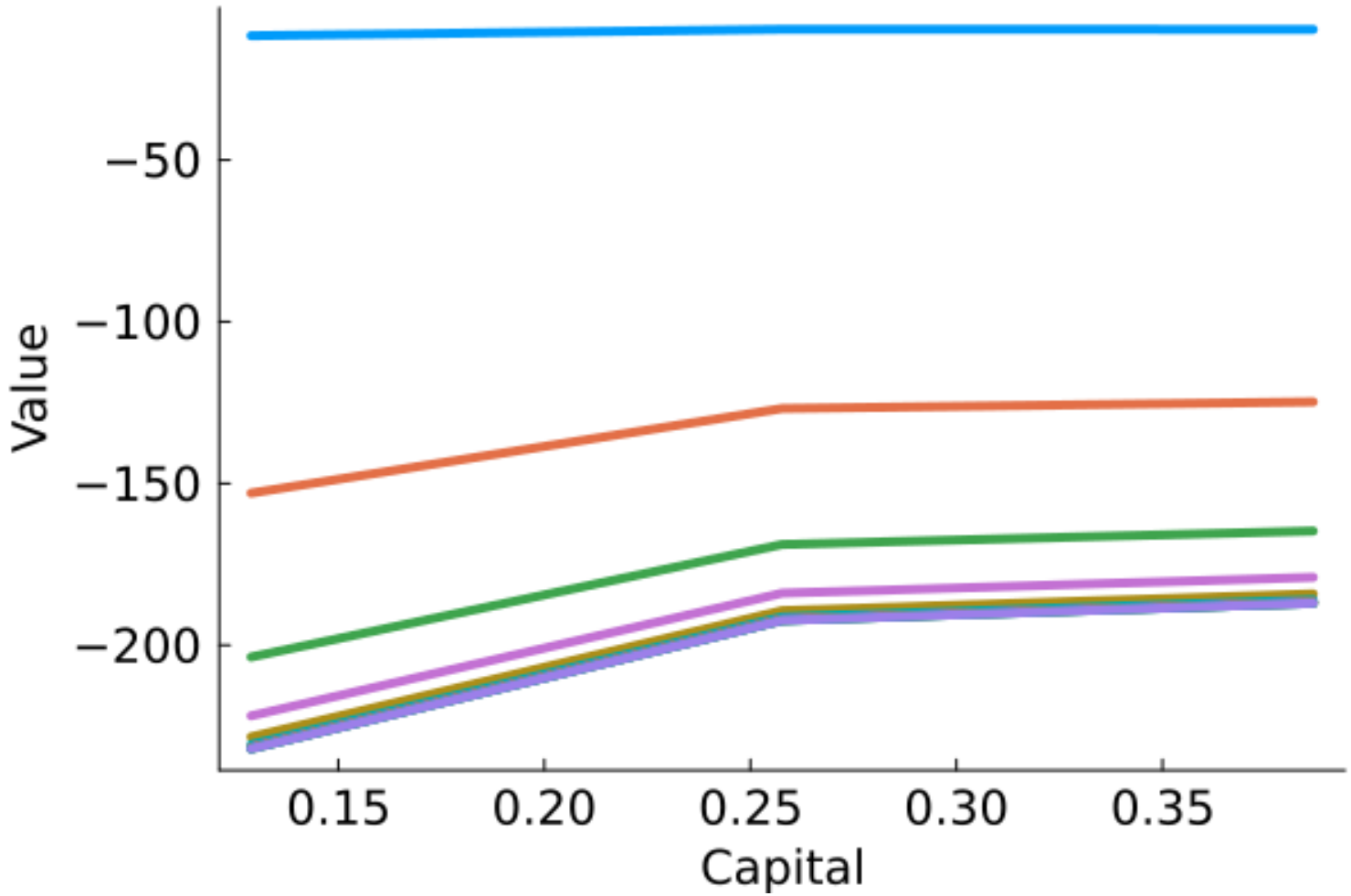
```
grid_size = 3;
grid = collect(range(params_dis.capital_lower,
    stop = params_dis.capital_upper,
    length = grid_size))
```

```
## 3-element Vector{Float64}:
##  0.12885743408203118
##  0.25771486816406236
##  0.3865723022460935
```

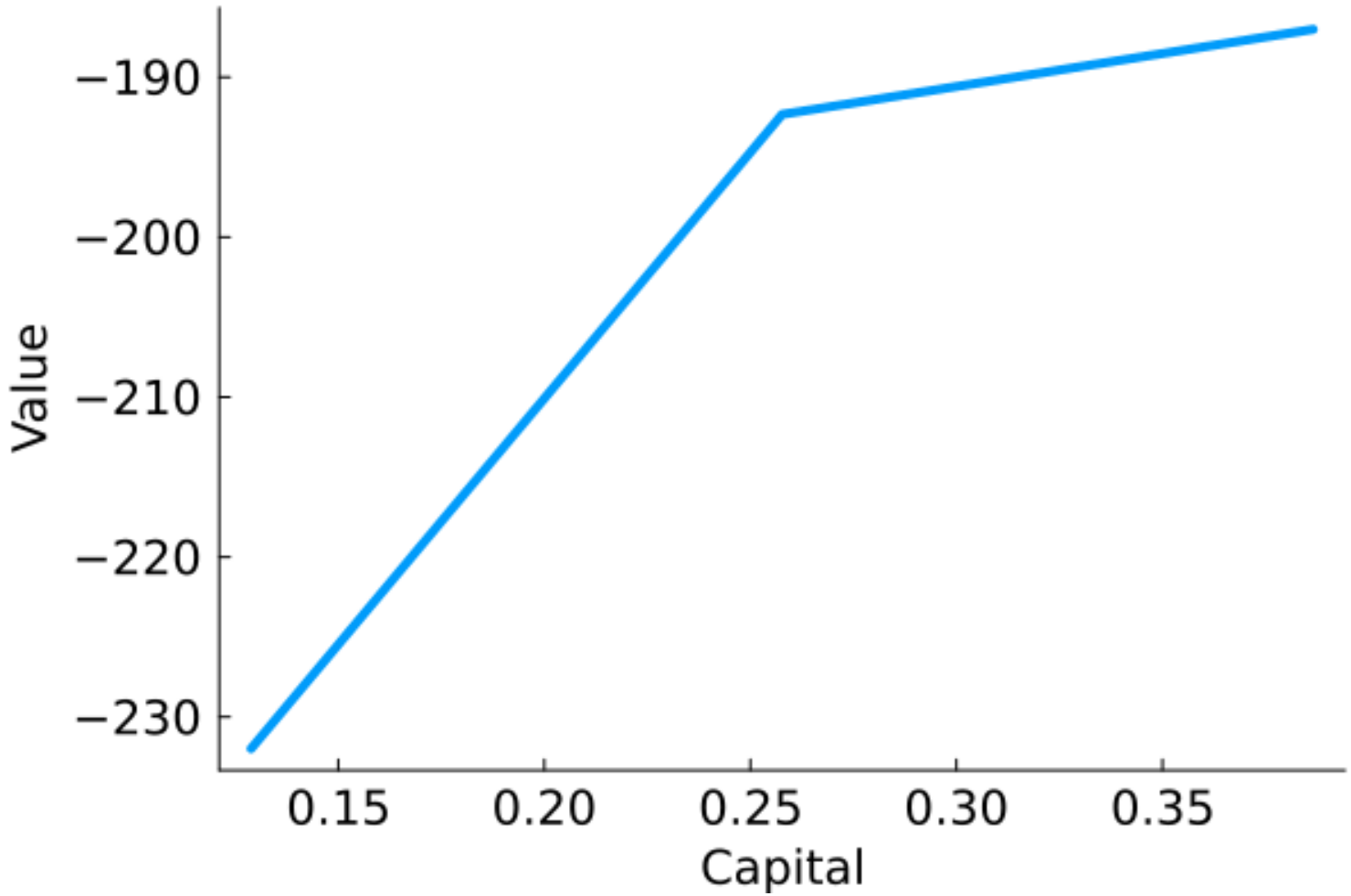
```
value, v_store = @time iterate_value(grid, params_dis)
```

```
## Current maximum value difference at iteration 10 is 7.310316889342374.
## Current maximum value difference at iteration 20 is 4.376956759187493.
## Current maximum value difference at iteration 30 is 2.6206456931746516.
## Current maximum value difference at iteration 40 is 1.5690773811596443.
## Current maximum value difference at iteration 50 is 0.9394645886236788.
## Current maximum value difference at iteration 60 is 0.5624921523154001.
## Current maximum value difference at iteration 70 is 0.33678482962292833.
```

The value function: every 20 iterations



The value function: final

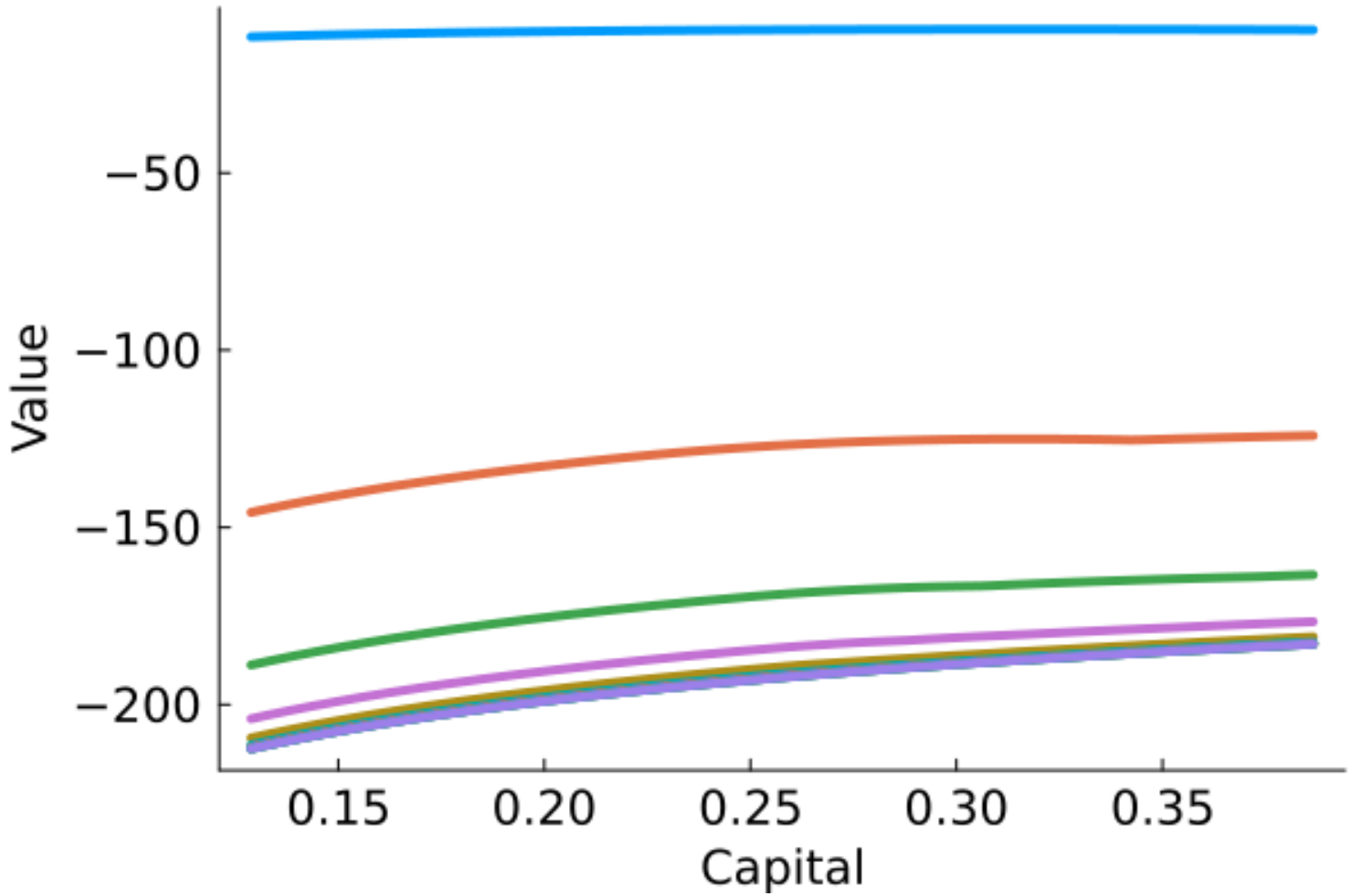


Discretizing the state space

```
grid_size = 100;
grid = collect(range(params_dis.capital_lower,
    stop = params_dis.capital_upper,
    length = grid_size));

value, v_store = @time iterate_value(grid, params_dis)
```

```
## Current maximum value difference at iteration 10 is 6.914720355073825.
## Current maximum value difference at iteration 20 is 3.8092197025250982.
## Current maximum value difference at iteration 30 is 2.221007019891772.
## Current maximum value difference at iteration 40 is 1.316405627475831.
## Current maximum value difference at iteration 50 is 0.7840556792955624.
## Current maximum value difference at iteration 60 is 0.4679885486935973.
## Current maximum value difference at iteration 70 is 0.27994507576624983.
## Current maximum value difference at iteration 80 is 0.16751908240780722.
## Current maximum value difference at iteration 90 is 0.1002998626648548.
## Current maximum value difference at iteration 100 is 0.059970032149010 42.
## Current maximum value difference at iteration 110 is 0.03590627349490205.
## Current maximum value difference at iteration 120 is 0.0214984122918338.
## Current maximum value difference at iteration 130 is 0.01287189357412899.
## Current maximum value difference at iteration 140 is 0.007706878160803399.
```

The value function: every 20 iterations



The value function: final