Econ 712: Problem set 4 - Solutions

Q1&Q2

See attached codes and inlined comments.

Q3

(a)

Household problem:

$$V(k, K, z) = \max_{k', c} \{u(c) + \beta E[V(k', K', z')|z]\}$$

s.t. $c + k' = (1 - \tau)(w(K, z)N + r(K, z)k) + (1 - \delta)k + T(K, z) + \pi(K, z)$

Firms:

$$\max_{\hat{K}, \hat{N}} \pi(K, z) = zF(\hat{K}, \hat{N}) - w(K, N)\hat{N} - r(K, N)\hat{K}$$

Gov:

$$\tau(w(K, z)N + r(K, z)K) = T(K, z)$$

Defining RCE: A value function $V(k, K, z)$, policy functions $c(k, K, z), k'(k, K, z)$; firm demands $\hat{K}(K, z), \hat{N}(K, z)$; pricing functions $w(K, z), r(K, z)$; transfer function $T(K, z)$ such that

1. Given prices, V solves the Bellman eq, and the policy functions give the maximum of the RHS of the Bellman eq.
2. Given prices, $\hat{K}$and $\hat{N}$ solves the firms problem.
3. $T(K, z)$ satisfies the Gov balanced budget.
4. Markets clear: $\hat{N} = N$, $\hat{K} = K = k$

(b)

Firms problem give $w(K, z) = zF_2(K, N), r(K, z) = zF_1(K, N), \pi(K, z) = zF(K, N) - zF_1(K, N)K - zF_2(K, N)N$.

Household FOCs:

$$\lambda = u'(c)$$
$$\lambda = \beta E[V_1(k', K', z')|z]$$

Envelope condition:

$$V_1(k, K, z) = ((1 - \tau)r(K, z) + 1 - \delta)\lambda$$

Euler eq:

$$u'(c) = \beta E[u'(c')((1 - \tau)r' + 1 - \delta)|z]$$

(c)

Equil condition: The consumer's BC basically becomes the resource constraint after imposing equil conditions

$$C + K' = zF(K, N) + (1 - \delta)K$$

Then

$$u'(zF(K,N)+(1-\delta)K-K') = \beta E[u'(z'F(K',N)+(1-\delta)K'-K'')((1-\tau)F_1(K',N)+1-\delta)|z]$$

(d)
At $\delta = 1$, the above becomes

$$u'(zF(K,N)-K') = \beta(1-\tau)E[u'(z'F(K',N)-K'')F_1(K',N)|z]$$

which is the same as the planner's problem but with a lower discount $\hat{\beta} = 1-\tau$. This is because the tax lowers the returns to investment, making future consumption more costly, which is the same as having less weight.

Q4

(a)
With $a = \rho$, we can see that

$$V_t^{1-\rho} = (1-\beta)c_t^{1-\rho} + \beta\mathbb{E}_t(V_{t+1}^{1-\rho})$$

By relabeling $W_t = V_t^{1-\rho}$, we can see this is the same as the standard formulation. It can be seen as the agent is maximizing the objective $(1-\beta)(1-\rho)\mathbb{E}_t \sum_{j=0}^{\infty} \beta^{t+j} \frac{c_{t+j}^{1-\rho}}{1-\rho}$.

(b)
In Epstein-Zin, risk aversion is characterized by α whereas the inter-temporal elasticity of substitution is governed by parameter ρ.

(c)
It is easier to derive the derivatives using the following formulation of Epstein-Zin:

$$V_t^{1-\rho} = (1-\beta)c_t^{1-\rho} + \beta(\mathbb{E}_t V_{t+1}^{1-\alpha})^{\frac{1-\rho}{1-\alpha}} \tag{1}$$

Take derivatives with respect to c_t on both sides, we have

$$(1-\rho)V_t^{-\rho} \cdot \frac{\partial V_t}{\partial c_t} = (1-\beta)(1-\rho)c_t^{-\rho} \Longrightarrow \frac{\partial V_t}{\partial c_t} = V_t^{\rho}(1-\beta)c_t^{-\rho}$$

Roll this one period forward, we have

$$\frac{\partial V_{t+1}}{\partial c_{t+1}} = V_{t+1}^{\rho}(1-\beta)c_{t+1}^{-\rho}$$

Similarly, take derivative with respect to V_{t+1} on both sides:

$$(1-\rho)V_t^{-\rho} \cdot \frac{\partial V_t}{\partial V_{t+1}} = \beta \cdot \frac{1-\rho}{1-\alpha}(\mathbb{E}_t V_{t+1}^{1-\alpha})^{\frac{\alpha-\rho}{1-\alpha}} \cdot (1-\alpha)V_{t+1}^{-\alpha}$$
$$\Longrightarrow \frac{\partial V_t}{\partial V_{t+1}} = \beta V_t^{\rho}(\mathbb{E}_t V_{t+1}^{1-\alpha})^{\frac{\alpha-\rho}{1-\alpha}} V_{t+1}^{-\alpha}$$

Plug these results into the definition of stochastic discount factor, we have

$$S_t = \frac{\frac{\partial V_t}{\partial V_{t+1}}\frac{\partial V_{t+1}}{\partial c_{t+1}}}{\frac{\partial V_t}{\partial c_t}} = \frac{\beta V_t^{\rho}(\mathbb{E}_t V_{t+1}^{1-\alpha})^{\frac{\alpha-\rho}{1-\alpha}} V_{t+1}^{-\alpha} \cdot V_{t+1}^{\rho}(1-\beta)c_{t+1}^{-\rho}}{V_t^{\rho}(1-\beta)c_t^{-\rho}}$$
$$= \beta\left(\frac{c_{t+1}}{c_t}\right)^{-\rho}\left(\frac{V_{t+1}}{(\mathbb{E}V_{t+1}^{1-\alpha})^{\frac{1}{1-\alpha}}}\right)^{\rho-\alpha}$$

Note that the first terms are the usual CRRA terms, and the second term measures next period's value relative to the certainty equivalence. If $\rho = \alpha$, we get the standard stochastic discount factor under CRRA preferences.

(d)

Conjecture a Markov pricing function $p(\cdot)$ with today's state, m_t, as the argument (since S_t is already used to denote stochastic discount factor). The Bellman equation is given by

$$\begin{aligned} &V(a,m) = \max_{c,a'}\left((1-\beta)c^{1-\rho} + \beta(\mathbb{E}V^{1-\alpha}(a',m'))^{\frac{1-\rho}{1-\alpha}}\right)^{\frac{1}{1-\rho}} \\ &\text{subject to} \quad c + p(m)a' = (p(m)+m)a \end{aligned} \tag{2}$$

Substitute in the budget constraint, the optimization problem is given by

$$V(a,m) = \max_{a'}\left((1-\beta)[(p(m)+m)a - p(m)a']^{1-\rho} + \beta(\mathbb{E}V^{1-\alpha}(a',m'))^{\frac{1-\rho}{1-\alpha}}\right)^{\frac{1}{1-\rho}}$$

The first order condition is given by

$$(1-\beta)(1-\rho)p(m)c^{-\rho} = \beta\frac{1-\rho}{1-\alpha}(\mathbb{E}V^{1-\alpha}(a',m'))^{\frac{\alpha-\rho}{1-\alpha}} \cdot (1-\alpha)V^{-\alpha}(a',m') \cdot V_1(a',m')$$
$$\implies [FOC] \qquad p(m)c^{-\rho} = \frac{\beta}{1-\beta}(\mathbb{E}V^{1-\alpha}(a',m'))^{\frac{\alpha-\rho}{1-\alpha}} \cdot V^{-\alpha}(a',m') \cdot V_1(a',m')$$

By the envelope theorem (similar to the approach in equation (1)),

$$(1-\rho)V^{-\rho}(a,m) \cdot V_1(a,m) = (1-\beta)(1-\rho)c^{-\rho} \cdot (p(m)+m)$$

Roll one period forward, we get

$$[ENV] \qquad V_1(a',m') = (1-\beta)(c')^{-\rho}V^{\rho}(a',m')(p(m')+m')$$

Combine $[FOC]$ and $[ENV]$, we have

$$p(m)c^{-\rho} = \frac{\beta}{1-\beta}(\mathbb{E}V^{1-\alpha}(a',m'))^{\frac{\alpha-\rho}{1-\alpha}} \cdot V^{-\alpha}(a',m') \cdot (1-\beta)(c')^{-\rho}V^{\rho}(a',m')(p(m')+m')$$
$$\implies [EE] \qquad p(m) = \beta\mathbb{E}[S \cdot (p(m')+m')]$$

where $S_t = \frac{\frac{\partial V_t}{\partial V_{t+1}}\frac{\partial V_{t+1}}{\partial c_{t+1}}}{\frac{\partial V_t}{\partial c_t}}$ is the stochastic discount factor defined previously.

(e)

A competitive recursive equilibrium is a value function $V(a,m)$ with corresponding policy function $c(a,m), a'(a,m)$ and a bounded price function $p(m)$ such that

Markets clear: $(c = d, a' = 1)$

The value function solves the household problem

(f)

We can just plug in the guess $V_t = \nu c_t$ into the value function and see the two sides actually match. Use $c_{t+1} = c_t(g + \sigma_c \epsilon_{t+1})$:

$$\begin{aligned} V_t = \nu c_t &= \max_{c_t, a_{t+1}} \left((1-\beta) c_t^{1-\rho} + \beta (\mathbb{E}_t (\nu c_{t+1})^{1-\alpha})^{\frac{1-\rho}{1-\alpha}} \right)^{\frac{1}{1-\rho}} \\ &= \max_{c_t, a_{t+1}} \left((1-\beta) c_t^{1-\rho} + \beta (\mathbb{E}_t (\nu c_t (g + \sigma_c \epsilon_{t+1}))^{1-\alpha})^{\frac{1-\rho}{1-\alpha}} \right)^{\frac{1}{1-\rho}} \end{aligned}$$

Note that c_t can be pulled out of the expectation operator, and it will cancel on both sides. So the $V_t = \nu c_t$ can be a solution to the value function (and is the only solution by contraction mapping theorem) where ν is defined as the solution to the equation

$$\nu = \left((1-\beta) + \beta \nu^{1-\rho} (\mathbb{E}_t (g + \sigma_c \epsilon_{t+1})^{1-\alpha})^{\frac{1-\rho}{1-\alpha}} \right)^{\frac{1}{1-\rho}}$$

We can insert the guess of value function into the expression of S_t:

$$\begin{aligned} S_t &= \beta \left(\frac{c_{t+1}}{c_t} \right)^{-\rho} \left(\frac{V_{t+1}}{(\mathbb{E}_t V_{t+1}^{1-\alpha})^{\frac{1}{1-\alpha}}} \right)^{\rho-\alpha} \\ &= \beta [(g + \sigma_c \epsilon_{t+1})]^{-\rho} \left(\frac{(g + \sigma_c \epsilon_{t+1})}{(\mathbb{E}_t (g + \sigma_c \epsilon_{t+1})^{1-\alpha})^{\frac{1}{1-\alpha}}} \right)^{\rho-\alpha} \\ &= \beta [(g + \sigma_c \epsilon_{t+1})]^{-\alpha} (\mathbb{E}_t (g + \sigma_c \epsilon_{t+1})^{1-\alpha})^{\frac{\alpha-\rho}{1-\alpha}} \end{aligned}$$

Where $\mathbb{E}_t (g + \sigma_c \epsilon_{t+1})^{1-\alpha}$ is some constant. Therefore,

$$\log S_t = \log \beta - \alpha \log (g + \sigma_c \epsilon_{t+1}) + \frac{\alpha - \rho}{1-\alpha} \log (\mathbb{E}_t (g + \sigma_c \epsilon_{t+1})^{1-\alpha}) \tag{3}$$

(g)

The price of the bond is given by

$$p^B = \mathbb{E}(S_t) = \beta \mathbb{E}(g + \sigma_c \epsilon_{t+1})^{-\alpha} (\mathbb{E}_t (g + \sigma_c \epsilon_{t+1})^{1-\alpha})^{\frac{\alpha-\rho}{1-\alpha}}$$

The risk free rate is then simply $1/p^B$. This differs from the standard CRRA case is just that S_t is different.

(h)

By recursively substitute the Euler equation and assume there is no bubble, we have

$$p_t = \mathbb{E}_t \sum_{j=1}^{\infty} \beta^j S_{t+j} m_{t+j}$$

Thus the expected return on the Lucas tree is

$$\mathbb{E}_t R_{t+1} = \mathbb{E}_t[\frac{p_{t+1} + m_{t+1}}{p_t}] = \mathbb{E}_t[\frac{\left(\mathbb{E}_{t+1} \sum_{j=2}^{\infty} \beta^j S_{t+j} m_{t+j}\right) + m_{t+1}}{\mathbb{E}_t \sum_{j=1}^{\infty} \beta^j S_{t+j} m_{t+j}}]$$

In [1]:
```
using Plots, QuantEcon, LinearAlgebra, Dierckx, Statistics
```

Q1

In [5]:
```
### Params
const L = [0.7, 1.1]
const Q = [0.85 0.15
           0.05 0.95]
const β, γ, δ = 0.95, 3.0, 0.08
const α = 0.36
;
```

In [45]:
```
### Discretize asset grid
amin = 0.0
amax = 20.0
na = 1000
agrid = collect(range(amin, amax, length = na))
;
```

In [7]:
```
### Coding up my own function for max and argmax
function mymax(x)
    opt = -1e10
    arg = 0.0
    for i in eachindex(x)
        if x[i] >= opt
            opt = x[i]
            arg = i
        end
    end
    return opt, arg
end

### Function to find ergodic distro of a transition matrix
function ergodic(P)
    A = P - I + ones(size(P))
    B = ones(size(P)[1])
    X = A' \ B
    return X
end
;
```

In [6]:
```
### Mean of labour
const N = L' * ergodic(Q)
;
```

In [7]:

```
### Some functions
function u(c)
    return c^(1-γ) / (1-γ)
end

### Returns net of depreciation
function R(k)
    return α * k^(α-1) * N^(1-α) - δ
end

function W(k)
    return (1-α) * k^α * N^(-α)
end
;
```

In [8]:

```
### Small modification to Ps3 functions
function T(v, grid, K)
    r = R(K)
    w = W(K)
    vnext = zero(v) ### Placeholders. v is a na X 2 matrix, since we have 2 discrete sta
tes for shocks
    pol = zero(v)
    pol_arg = zero(v) ### This will be useful for the large transition matrix for (d)
    for (il, l) in enumerate(L) ### Loop for each shock state
        prob = Q[il, :]
        for (ia, a) in enumerate(grid) ### Loop for each asset state
            ynow = w * l + (1+r) * a
            val = zero(grid[ynow .- grid .> 0]) ### Only considering feasible a_next (wh
ich i call a_p)
            for (ia_p, a_p) in enumerate(grid[ynow .- grid .> 0])
                val[ia_p] = u(ynow - a_p) + β * v[ia_p, :]' * prob ### Note the expectat
ion of future values here
            end
            opt, arg = mymax(val)
            vnext[ia, il] = opt
            pol[ia, il] = grid[arg]
            pol_arg[ia, il] = arg
        end
    end
    return vnext, pol, pol_arg
end

function VFI(vguess, grid, K; tol = 1e-4, maxiter = 1000)
    err = 1.0
    i = 0
    vnow = vguess
    pol, pol_arg = zero(vguess), zero(vguess)
    while err > tol && i < maxiter
        vnext, pol, pol_arg = T(vnow, grid, K)
        err = maximum(abs.(vnext - vnow))
        i += 1
        vnow = vnext
        if i % 80 == 1
            println("iter: ", i, " error: ", err) ### Print some stuff so we dont get im
patient
        end
    end
    return vnow, pol, pol_arg
end

function get_trans(pol_arg, grid)
    P = zeros((2 * na, 2 * na)) ### A square matrix for total states (which is 2*na)
    for il in eachindex(L)
        prob = Q[il, :]
        for ia in eachindex(grid)
            index = Int(pol_arg[ia, il]) ### This gives us the index of the optimal anex
t, given our state (a, l)
            P[na * (il-1) + ia, index] += prob[1] ### A fraction goes to (anext, l = l_l
ow)
            P[na * (il-1) + ia, na + index] += prob[2] ### A fraction goes to (anext, l
 = l_high)
        end
    end
    return P
end
;
```

In [46]:
```
### Wrap the above into a single function
function get_K(vguess, grid, Know)
    v, pol, pol_arg = VFI(vguess, grid, Know, tol = 1e-5)
    P = get_trans(pol_arg, grid)
    ivd = ergodic(P)
    Knew = ivd' * vcat(grid, grid)
    return Knew, v
end
;
```

In [47]:
```
### Coding up my own root finder to take advantage of the guess of V for each value
### function iteration. Here I am using bisection.
function solve(vguess, Kguess, grid; tol = 5e-4)
    i = 0
    err = 1
    Know = Kguess
    vnow = vguess
    Klow = 4.0
    Khigh = 8.0
    while err > tol && Klow != Khigh
        Knext, vnow = get_K(vnow, grid, Know)
        diff = Knext - Know
        if diff > 0
            Klow = Know
        else
            Khigh = Know
        end
        Know = (Khigh + Klow) / 2
        err = abs(diff)
        i += 1

        println("Diff: $err , Iter: $i , K: $Know")

    end
    return Know, vnow
end
;
```

In [48]:
```
Kstar, vstar = solve(hcat(u.(agrid.+0.01), u.(agrid.+0.01)), 6.0, agrid)
;
```

```
iter: 1 error: 4997.866429979431
iter: 81 error: 0.005083012939982368
iter: 161 error: 8.394700871505734e-5
Diff: 4.554938931561045 , Iter: 1 , K: 5.0
iter: 1 error: 0.09619984741214616
iter: 81 error: 0.0004172997494205077 4
Diff: 0.8212680575272113 , Iter: 2 , K: 5.5
iter: 1 error: 0.05184816950948168
iter: 81 error: 0.0002113829737204753 2
Diff: 3.282288392698769 , Iter: 3 , K: 5.25
iter: 1 error: 0.02485769297785545
iter: 81 error: 0.0001077150463704157 3
Diff: 2.1084238202455206 , Iter: 4 , K: 5.125
iter: 1 error: 0.013221390325632854
iter: 81 error: 4.7898504369925377e-5
Diff: 1.085383748965409 , Iter: 5 , K: 5.0625
iter: 1 error: 0.006825524701831398
iter: 81 error: 2.257576427266983e-5
Diff: 0.29631731840151154 , Iter: 6 , K: 5.03125
iter: 1 error: 0.003470792882676932
iter: 81 error: 1.1729666991477927e-5
Diff: 0.20628474584987622 , Iter: 7 , K: 5.046875
iter: 1 error: 0.0017276564755785984
Diff: 0.046707274077526506 , Iter: 8 , K: 5.0390625
iter: 1 error: 0.0008580574534864382
Diff: 0.07110350211969685 , Iter: 9 , K: 5.04296875
iter: 1 error: 0.0004246778593355316
Diff: 0.016207731083299315 , Iter: 10 , K: 5.044921875
iter: 1 error: 0.00022598080038171986
Diff: 0.014547188726295346 , Iter: 11 , K: 5.0439453125
iter: 1 error: 9.910698492099357e-5
Diff: 0.004678166856335153 , Iter: 12 , K: 5.04345703125
iter: 1 error: 6.331301606543605e-5
Diff: 0.003107697719393343 , Iter: 13 , K: 5.043701171875
iter: 1 error: 1.8203053830490035e-5
Diff: 0.002004284391603733 , Iter: 14 , K: 5.0438232421875
iter: 1 error: 2.1802774266888036e-5
Diff: 0.0018783581266603733 , Iter: 15 , K: 5.04388427734375
iter: 1 error: 1.5738372631801667e-5
Diff: 0.0036762475859992705 , Iter: 16 , K: 5.043853759765625
iter: 1 error: 6.681794499030502e-6
Diff: 0.002956511007797147 , Iter: 17 , K: 5.0438385009765625
iter: 1 error: 5.184565997851109e-6
Diff: 0.002941252218734647 , Iter: 18 , K: 5.043830871582031
iter: 1 error: 4.205570443005513e-6
Diff: 0.002933622824203397 , Iter: 19 , K: 5.043827056884766
iter: 1 error: 3.5177045987211386e-6
Diff: 0.002929808126937772 , Iter: 20 , K: 5.043825149536133
iter: 1 error: 2.9942768140500675e-6
Diff: 0.002927900778304959 3 , Iter: 21 , K: 5.043824195861816
iter: 1 error: 2.5827726339855417e-6
Diff: 0.002926947103988553 , Iter: 22 , K: 5.043823719024658
iter: 1 error: 2.2439361702097926e-6
Diff: 0.002926470266830352 , Iter: 23 , K: 5.043823480606079
iter: 1 error: 1.9572615013174754e-6
Diff: 0.002926231848251248 4 , Iter: 24 , K: 5.043823361396789 6
iter: 1 error: 1.7082366037257657e-6
Diff: 0.002926112638961697 6 , Iter: 25 , K: 5.043823301792145
iter: 1 error: 1.498043578962438e-6
Diff: 0.002926053034316922 , Iter: 26 , K: 5.043823271989822
iter: 1 error: 1.3234526994310158e-6
```

```
Diff: 0.0029260232319945345 , Iter: 27 , K: 5.043823257088661
iter: 1 error: 1.17081638251193e-6
Diff: 0.0036985720403652778 , Iter: 28 , K: 5.043823249638081
iter: 1 error: 1.0436003554659123e-6
Diff: 0.003698564589784681 , Iter: 29 , K: 5.04382324591279
iter: 1 error: 9.352139676011006e-7
Diff: 0.0036985608644943824 , Iter: 30 , K: 5.043823244050145
iter: 1 error: 8.548763430482609e-7
Diff: 0.003698559001849233 , Iter: 31 , K: 5.043823243118823
iter: 1 error: 7.993374380177443e-7
Diff: 0.0036985580705266585 , Iter: 32 , K: 5.043823242653161
iter: 1 error: 7.499881764516658e-7
Diff: 0.0036985576048653712 , Iter: 33 , K: 5.043823242420331
iter: 1 error: 7.118241782677615e-7
Diff: 0.0036985573720347276 , Iter: 34 , K: 5.043823242303915
iter: 1 error: 6.750465733063038e-7
Diff: 0.0036985572556194057 , Iter: 35 , K: 5.043823242245708
iter: 1 error: 6.389983635557428e-7
Diff: 0.003698557197411745 , Iter: 36 , K: 5.043823242216604
iter: 1 error: 6.041480080654082e-7
Diff: 0.0036985571683079144 , Iter: 37 , K: 5.043823242202052
iter: 1 error: 5.700125544905177e-7
Diff: 0.003698557153755999 , Iter: 38 , K: 5.043823242194776
iter: 1 error: 5.375460556678036e-7
Diff: 0.0036985571464800415 , Iter: 39 , K: 5.043823242191138
iter: 1 error: 5.067063604258237e-7
Diff: 0.0036985571428420627 , Iter: 40 , K: 5.043823242189319
iter: 1 error: 4.77339593629722e-7
Diff: 0.0036985571410230733 , Iter: 41 , K: 5.0438232421884095
iter: 1 error: 4.495863681341916e-7
Diff: 0.0036985571401135786 , Iter: 42 , K: 5.043823242187955
iter: 1 error: 4.233557575616942e-7
Diff: 0.0036985571396588313 , Iter: 43 , K: 5.043823242187727
iter: 1 error: 3.98783020827409e-7
Diff: 0.0036985571394314576 , Iter: 44 , K: 5.043823242187614
iter: 1 error: 3.7566849364623067e-7
Diff: 0.0036985571393177707 , Iter: 45 , K: 5.043823242187557
iter: 1 error: 3.537758210825359e-7
Diff: 0.0036985571392609273 , Iter: 46 , K: 5.043823242187528
iter: 1 error: 3.330847171412188e-7
Diff: 0.0036985571392325056 , Iter: 47 , K: 5.043823242187514
iter: 1 error: 3.1351813944979767e-7
Diff: 0.0036985571392182948 , Iter: 48 , K: 5.043823242187507
iter: 1 error: 2.950095172593592e-7
Diff: 0.0036985571392111893 , Iter: 49 , K: 5.0438232421875036
iter: 1 error: 2.7753491771420613e-7
Diff: 0.0036985571392076366 , Iter: 50 , K: 5.043823242187502
iter: 1 error: 2.6103601413751676e-7
Diff: 0.0036985571392058603 , Iter: 51 , K: 5.043823242187501
iter: 1 error: 2.455055341243906e-7
Diff: 0.003698557139204972 , Iter: 52 , K: 5.0438232421875
iter: 1 error: 2.30889042995841e-7
Diff: 0.003698557139204084 , Iter: 53 , K: 5.0438232421875
```

(a)

In [49]:
```
Wstar = W(Kstar)
Rstar = R(Kstar)
println("Equil kapital: $Kstar, Equil wage: $Wstar, Equil interest: $Rstar")
```

```
Equil kapital: 5.0438232421875, Equil wage: 1.1459707244622583, Equil interest: 0.04780
157066536185
```

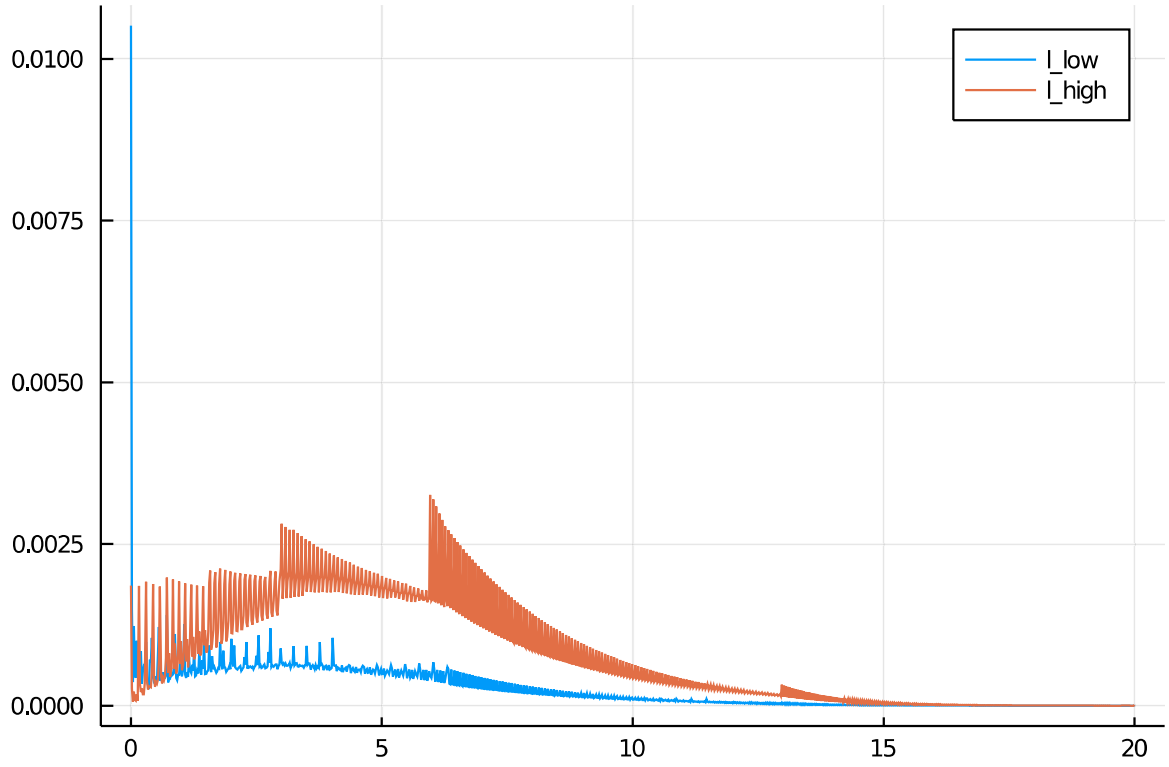
In [50]:
```
v, pol, pol_arg = VFI(vstar, agrid, Kstar, tol = 5e-5)
ivd = ergodic(get_trans(pol_arg, agrid))
;
```

```
iter: 1 error: 2.1711658249046195e-7
```

(b)

In [52]:
```
plot(agrid, ivd[1:na], label = "l_low")
plot!(agrid, ivd[na+1:end], label = "l_high")
```

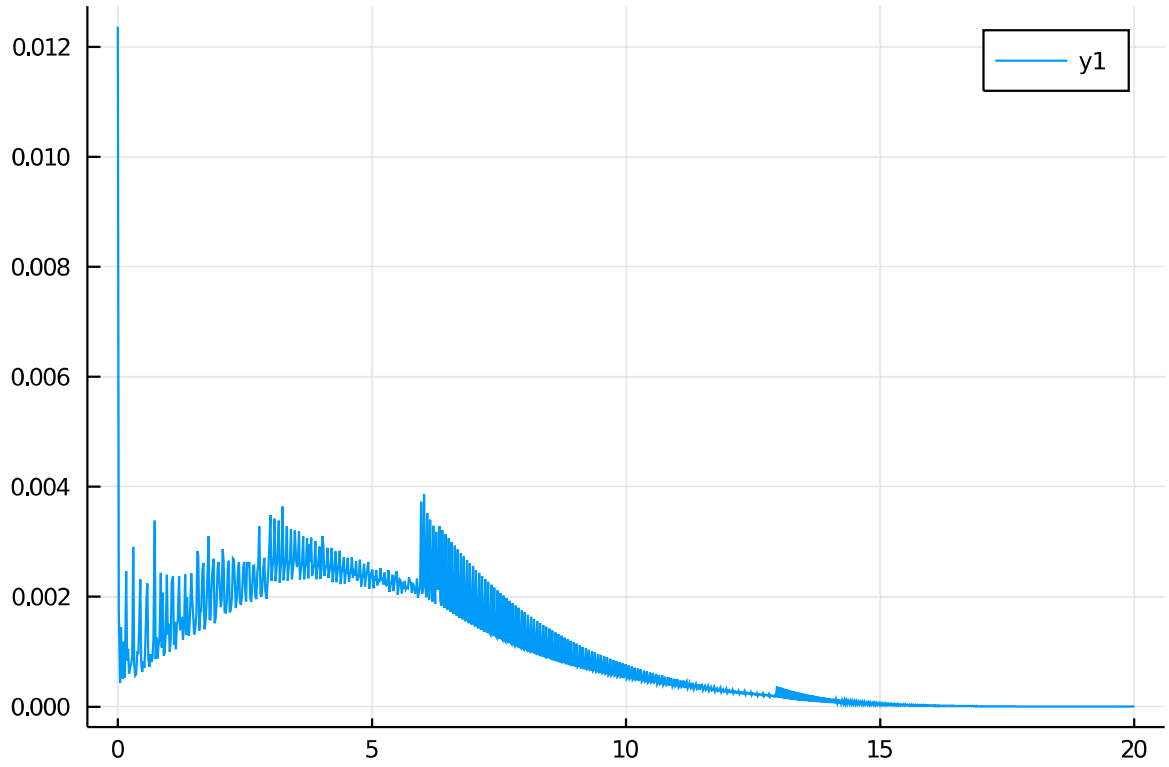
Out[52]:



Asset distro by labour shock

In [73]:
```
plot(agrid, ivd[1:na] + ivd[na+1:end])
```

Out[73]:



Asset distro added up

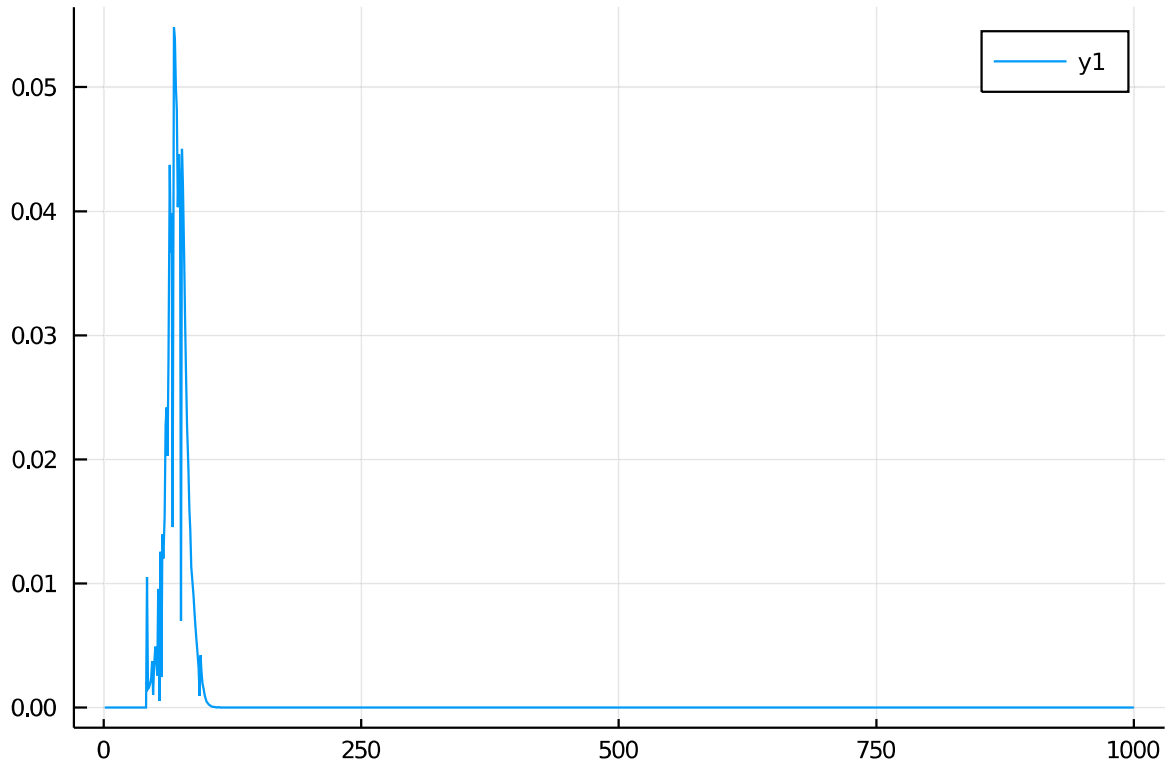
In [90]:
```
function get_cpol(pol, grid, K)
    r = R(K)
    w = W(K)
    cpol = zero(pol)
    cpol_arg = zero(pol)
    for (il, l) in enumerate(L) ### Loop for each shock state
        prob = Q[il, :]
        for (ia, a) in enumerate(grid) ### Loop for each asset state
            ynow = w * l + (1+r) * a
            cnow = ynow - pol[ia, il]
            cpol[ia, il] = cnow
            cpol_arg[ia, il] = searchsortedfirst(grid, cnow) ### Getting our index for c
onsumption policy
        end
    end
    return cpol, cpol_arg
end
;
```

In [91]:
```
cpol, cpol_arg = get_cpol(pol, agrid, Kstar)
;
```

In [111]:
```
function get_distro_c(ivd, cpol_arg, grid)
    na = length(grid)
    D = zero(grid)
    for il in eachindex(L)
        for ia in eachindex(grid)
            index = Int(cpol_arg[ia, il])
            D[index] += ivd[na * (il-1) + ia]
        end
    end
    return D
end
;
```

In [112]:
```
plot(get_distro_c(ivd, cpol_arg, agrid))
```

Out[112]:



Less inequality in consumption

In [113]:
```
### Discretize asset grid
amin = -2.0
amax = 20.0
na = 1000
agrid = collect(range(amin, amax, length = na))
;
```

In [114]:
```
Kstar, vstar = solve(hcat(u.(agrid.+0.01), u.(agrid.+0.01)), 6.0, agrid)
;
```

```
iter: 1 error: 7777.261871833908
iter: 81 error: 0.006576268623184944
iter: 161 error: 0.00010839451622146612
Diff: 6.424420323393997 , Iter: 1 , K: 5.0
iter: 1 error: 0.20178602192693695
iter: 81 error: 0.0008247081588912408
iter: 161 error: 1.0589646491254712e-5
Diff: 0.5217208632136439 , Iter: 2 , K: 4.5
iter: 1 error: 0.1528830814967037
iter: 81 error: 0.0002609479701831674
Diff: 14.099183210318621 , Iter: 3 , K: 4.75
iter: 1 error: 0.08249819574308503
iter: 81 error: 0.00013024575491549228
Diff: 10.891155410439993 , Iter: 4 , K: 4.875
iter: 1 error: 0.03653323674841147
iter: 81 error: 8.068507924363644e-5
Diff: 4.803216221819508 , Iter: 5 , K: 4.9375
iter: 1 error: 0.01724825793666973
iter: 81 error: 4.398113209092003e-5
Diff: 1.397323375498571 , Iter: 6 , K: 4.96875
iter: 1 error: 0.008389125521235385
iter: 81 error: 2.284764763516023e-5
Diff: 0.3299102354336032 , Iter: 7 , K: 4.984375
iter: 1 error: 0.004141091145573483
iter: 81 error: 1.1631590192351382e-5
Diff: 0.10866391619941229 , Iter: 8 , K: 4.9765625
iter: 1 error: 0.002052103154202456
Diff: 0.09174195018691655 , Iter: 9 , K: 4.98046875
iter: 1 error: 0.0010227340944535257
Diff: 0.005770505288337446 , Iter: 10 , K: 4.978515625
iter: 1 error: 0.0005063825172850756
Diff: 0.04122335345470063 , Iter: 11 , K: 4.9794921875
iter: 1 error: 0.00024900836309171837
Diff: 0.026661015205167615 , Iter: 12 , K: 4.97998046875
iter: 1 error: 0.00013808731721098866
Diff: 0.006263330676588907 , Iter: 13 , K: 4.980224609375
iter: 1 error: 7.334197079877924e-5
Diff: 0.0032174058236948966 , Iter: 14 , K: 4.9803466796875
iter: 1 error: 4.100219471325772e-5
Diff: 0.0024852157694921218 , Iter: 15 , K: 4.98040771484375
iter: 1 error: 2.49408491530545e-5
Diff: 0.005709470132087446 , Iter: 16 , K: 4.980377197265625
iter: 1 error: 5.280345726887958e-6
Diff: 0.005678952553962446 , Iter: 17 , K: 4.9803619384765625
iter: 1 error: 3.663045708535151e-6
Diff: 0.005663693764899946 , Iter: 18 , K: 4.980354309082031
iter: 1 error: 4.4052123193694115e-6
Diff: 0.005656064370368696 , Iter: 19 , K: 4.980350494384766
iter: 1 error: 4.560039565504326e-6
Diff: 0.005652249673103071 , Iter: 20 , K: 4.980348587036133
iter: 1 error: 4.24142085364565e-6
Diff: 0.005650342324470259 , Iter: 21 , K: 4.980347633361816
iter: 1 error: 3.7408190927123997e-6
Diff: 0.005649388650153853 , Iter: 22 , K: 4.980347156524658
iter: 1 error: 3.210959622634846e-6
Diff: 0.005648911818129956495 , Iter: 23 , K: 4.980346918106079
iter: 1 error: 2.718295551318306e-6
Diff: 0.005648673394416548 , Iter: 24 , K: 4.9803467988967896
iter: 1 error: 2.2835727104819625e-6
Diff: 0.005648554185126997 , Iter: 25 , K: 4.980346739292145
iter: 1 error: 1.912501378598108e-6
```

Diff: 0.005648494580482222 , Iter: 26 , K: 4.980346709489822
iter: 1 error: 1.6000181410902314e-6
Diff: 0.00564846477815983 4 , Iter: 27 , K: 4.980346694588661
iter: 1 error: 1.3291990903496753e-6
Diff: 0.00564844987699864 , Iter: 28 , K: 4.980346687138081
iter: 1 error: 1.1021248500497904e-6
Diff: 0.005648442426418043 , Iter: 29 , K: 4.98034668341279
iter: 1 error: 9.120875237300652e-7
Diff: 0.005648438701127745 , Iter: 30 , K: 4.980346681550145
iter: 1 error: 7.540977140507721e-7
Diff: 0.005648436838482596 , Iter: 31 , K: 4.980346680618823
iter: 1 error: 6.224739603766238e-7
Diff: 0.005648435907160021 , Iter: 32 , K: 4.980346680153161
iter: 1 error: 5.148539674593167e-7
Diff: 0.005648435441498734 , Iter: 33 , K: 4.980346679920331
iter: 1 error: 4.26352769089533e-7
Diff: 0.00564843520866809 , Iter: 34 , K: 4.980346679803915
iter: 1 error: 3.906093697381152e-7
Diff: 0.005648435092252768 , Iter: 35 , K: 4.980346679745708
iter: 1 error: 3.588100838669561e-7
Diff: 0.005648435034045107 , Iter: 36 , K: 4.980346679716604
iter: 1 error: 3.299826758507151 7e-7
Diff: 0.005648435004941277 , Iter: 37 , K: 4.980346679702052
iter: 1 error: 3.037981812781254e-7
Diff: 0.005648434990389362 , Iter: 38 , K: 4.980346679694776
iter: 1 error: 2.8099139282034e-7
Diff: 0.005648434983113404 , Iter: 39 , K: 4.980346679691138
iter: 1 error: 2.6113653550652316e-7
Diff: 0.005648434979475425 , Iter: 40 , K: 4.980346679689319
iter: 1 error: 2.434198203005167e-7
Diff: 0.005648434977656436 , Iter: 41 , K: 4.9803466796884095
iter: 1 error: 2.2681450495554145e-7
Diff: 0.005648434976746941 , Iter: 42 , K: 4.980346679687955
iter: 1 error: 2.1138375139173604e-7
Diff: 0.005648434976292194 , Iter: 43 , K: 4.980346679687727
iter: 1 error: 1.9754883862077577e-7
Diff: 0.00564843497606482 , Iter: 44 , K: 4.980346679687614
iter: 1 error: 1.8463633288945402e-7
Diff: 0.005648434975951133 , Iter: 45 , K: 4.980346679687557
iter: 1 error: 1.7266983309838224e-7
Diff: 0.00564843497589429 , Iter: 46 , K: 4.980346679687528
iter: 1 error: 1.618654570378908e-7
Diff: 0.005648434975865868 , Iter: 47 , K: 4.980346679687514
iter: 1 error: 1.5171902223443112e-7
Diff: 0.005648434975851657 , Iter: 48 , K: 4.980346679687507
iter: 1 error: 1.4256150393521239e-7
Diff: 0.005648434975844552 , Iter: 49 , K: 4.9803466796875036
iter: 1 error: 1.3394973308322733e-7
Diff: 0.005648434975840999 , Iter: 50 , K: 4.980346679687502
iter: 1 error: 1.2590104958576376e-7
Diff: 0.005648434975839223 , Iter: 51 , K: 4.980346679687501
iter: 1 error: 1.1853359715985334e-7
Diff: 0.0056484349758383345 , Iter: 52 , K: 4.9803466796875
iter: 1 error: 1.1159789803372178e-7
Diff: 0.005648434975837446 , Iter: 53 , K: 4.9803466796875

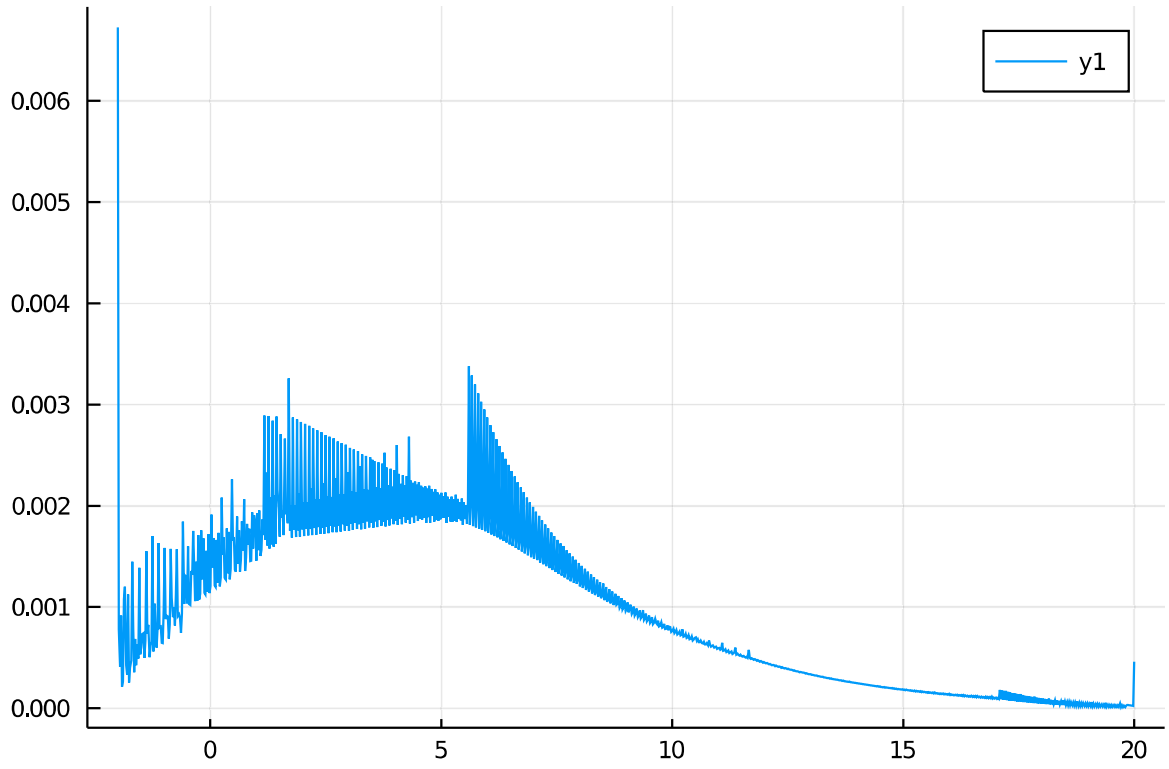
In [115]:
```
Wstar = W(Kstar)
Rstar = R(Kstar)
println("Equil kapital: $Kstar, Equil wage: $Wstar, Equil interest: $Rstar")
```

```
Equil kapital: 4.9803466796875, Equil wage: 1.1407577316068565, Equil interest: 0.04884
167815986652
```

In [116]:
```
v, pol, pol_arg = VFI(vstar, agrid, Kstar, tol = 5e-5)
ivd = ergodic(get_trans(pol_arg, agrid))
plot(agrid, ivd[1:na] + ivd[na+1:end])
```

```
iter: 1 error: 1.050718942963158e-7
```

Out[116]:



Compared to part a, there s less aggregate capital. Note in this distro the mass at the largest grid point is non zero. Ideally you should not have that there and should increase your max grid point. I got lazy though.

Q2

In [2]:
```
### Params
const P = [0.9 0.1 ; 0.1 0.9]
const Z = [0.8 1.2]
const β, δ, γ = 0.95, 0.1, 2.0
;
```

In [3]:
```
function F(k)
    return k^0.35
end

function u(c)
    return c^(1-γ) / (1-γ)
end
;
```

In [4]:
```
### Discretize grid of choice var (k)
kmin = 1e-3
kmax = δ^(-1/0.65) / 5
nk = 1000

kgrid = collect(range(kmin, kmax, length = nk))
;
```

(a)

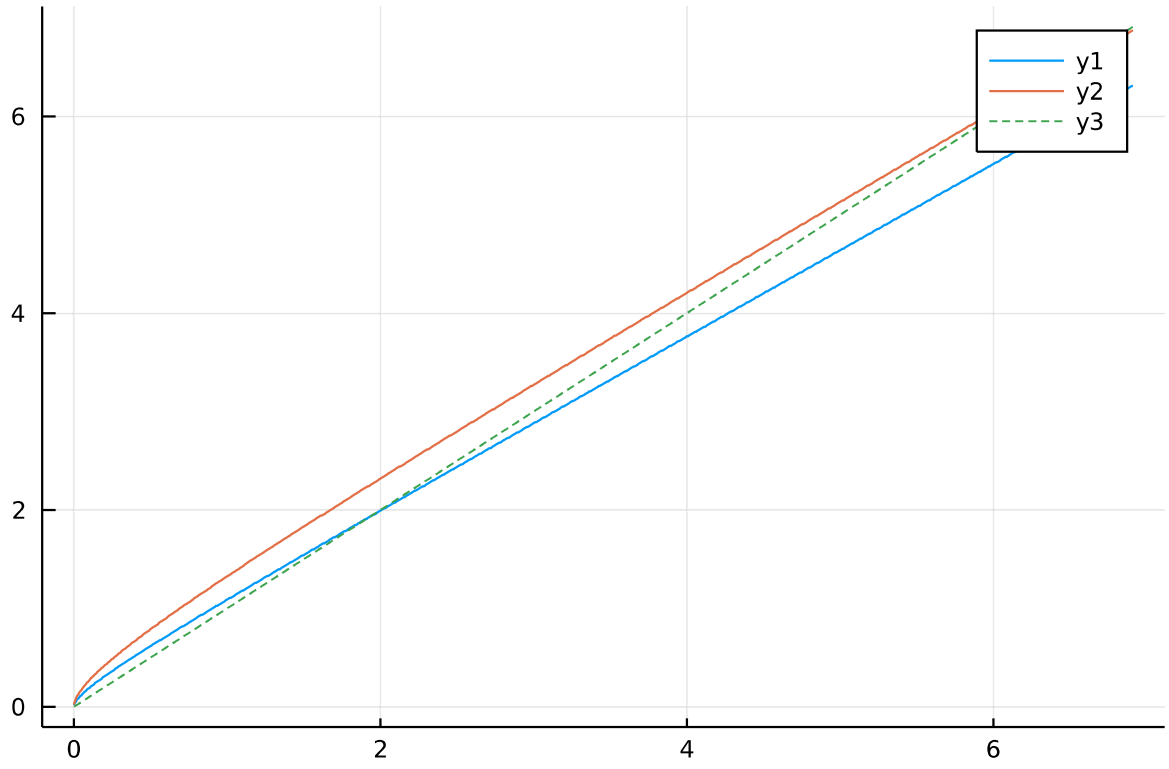
In [5]:
```
### Bellman operator
function T(v, grid)
    Tv = zero(v) ### Placeholder
    K = zero(v) ### Placeholder for policy
    for (iz, z) in enumerate(Z)
        prob = P[iz, :]
        for (i, know) in enumerate(grid) ### Tv as function of k
            ynow = z * F(know) + (1-δ) * know
            val_next = zero(grid[grid .< ynow])
            for (ik, knext) in enumerate(grid[grid .< ynow]) ### Evaluating at each (fea
sible) choice of knext
                val_next[ik] = u(ynow - knext) + β * v[ik, :]' * prob
            end
            opt, arg = mymax(val_next)
            Tv[i, iz] = opt
            K[i, iz] = grid[arg]
        end
    end
    return Tv, K
end

### Value function iteration
function VFI(vguess, grid; tol = 1e-4, maxiter = 1000, show = 1)
    ### Initialize
    i = 0
    err = 1
    vnow = vguess
    K = zero(grid)

    while i < maxiter && err > tol ### Loop until 2 iterations are close enough, or max
 iteration is reached
        vnext, K = T(vnow, grid)
        err = maximum(abs.(vnext - vnow)) ### Supnorm
        i += 1

        vnow = vnext

        if i % 50 == 1 && show == 1 ### print some stuff so we dont get impatient waitin
g
            println("iter: ", i, " error: ", err)
        end
    end

    if i == maxiter
        println("maxiter reached")
    end

    return vnow, K
end
;
```

In [8]:
```
vguess = hcat(kgrid, kgrid) / (1-β) ### Arbitrary guess
V, K = VFI(vguess, kgrid)
;
```

```
iter: 1 error: 16.841684435731352
iter: 51 error: 0.581021470843492
iter: 101 error: 0.04463825885606809
iter: 151 error: 0.003434653860219683
iter: 201 error: 0.0002642793383067499
```

In [9]:
```
plot(kgrid, K)
plot!(kgrid, kgrid, ls = :dash)
```

Out[9]:

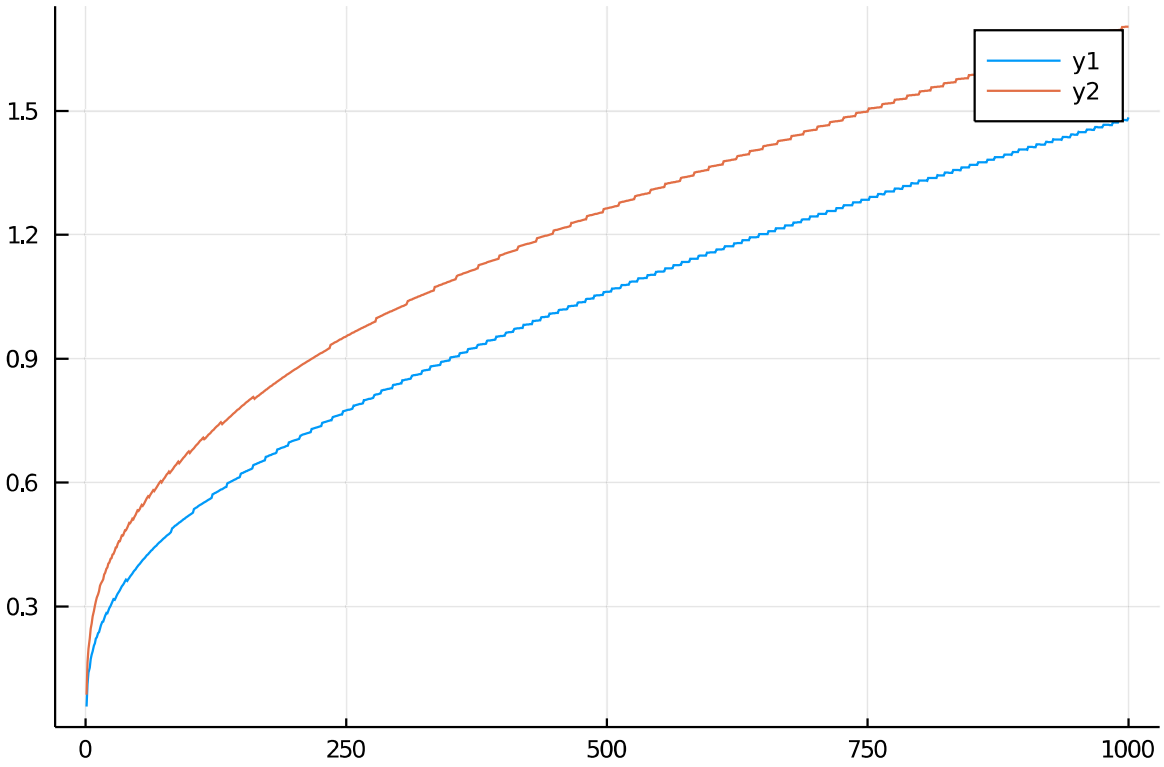


In [10]:
```
function get_cpol(K, kgrid)
    C = zero(K)
    for (iz, z) in enumerate(Z)
        for (i, know) in enumerate(kgrid)
            ynow = z * F(know) + (1-δ) * know
            C[i, iz] = ynow - K[i, iz]
        end
    end
    return C
end
;
```

In [11]:
```
C = get_cpol(K, kgrid)
;
```

In [12]:
```
plot(C)
```

Out[12]:



(b)

In [50]:
```
TT = 100000
P_mc = MarkovChain(P)
z_sim = simulate(P_mc, TT)
;
```

In [19]:
```
### Im using linear interpolation here. Alternatively, you could keep track of where k i
s moveing on kgrid,
### but this seems like a lot less work.
K_interp = [Spline1D(kgrid, K[:, 1], k=1), Spline1D(kgrid, K[:, 2], k=1)]
C_interp = [Spline1D(kgrid, C[:, 1], k=1), Spline1D(kgrid, C[:, 2], k=1)]
;
```

In [51]:
```
know = 4.0
k_sim = zeros(TT-1)
c_sim = zeros(TT-1)
F_sim = zeros(TT-1)
I_sim = zeros(TT-1)
w_sim = zeros(TT-1)
r_sim = zeros(TT-1)
for i in 1:TT-1
    knext = K_interp[z_sim[i]](know)
    k_sim[i] = know
    F_sim[i] = Z[z_sim[i]] * F(know)
    c_sim[i] = C_interp[z_sim[i]](know)
    I_sim[i] = knext - (1-δ) * know
    w_sim[i] = 0.65 * Z[z_sim[i]] * F(know)
    r_sim[i] = 0.35 * Z[z_sim[i]] * F(know) / know - δ

    know = knext
end
;
```

In [54]:
```
km = mean(k_sim)
cm = mean(c_sim)
println("Average K: $km , Average C: $cm")
```

```
Average K: 3.9522155017910663 , Average C: 1.2236691647765805
```

In [55]:
```
sc = std(c_sim) / std(F_sim)
si = std(I_sim) / std(F_sim)
println("Relative std of I: $si , Relative std of C: $sc")
```

```
Relative std of I: 0.5032108305031756 , Relative std of C: 0.5555638562981029
```

In [56]:
```
cf = cor(c_sim, F_sim)
rf = cor(r_sim, F_sim)
println("Corr between C and F: $cf , Corr between R and F: $rf")
```

```
Corr between C and F: 0.9498728602737422 , Corr between R and F: 0.07538957274635556
```

In [ ]: