Answer: The converged value is the following, and our code is attached in the appendix. The values are similar with results in the note. The R^2 is 0.99991.

$$\log K' = \begin{cases} 0.078 + 0.969 \log K & \text{if } z = z_g \\ 0.063 + 0.973 \log K & \text{if } z = z_b \end{cases}$$

■

Appendix

The first code file runs the code:

```
using LinearAlgebra, Parameters, Plots, LaTeXStrings, Latexify
theme(:juno)
default(fontfamily="Computer Modern", framestyle=:box) # LaTex-style

include("krusell_smith.jl")

SolveModel()
```

The second code file contains the relevant functions.

```
using LinearAlgebra, Parameters, Interpolations

@doc """
    The following function recieves the followig input:

        - d_z:Array{Float64,1} the duration of states [d_g, d_b]
        - d_unemp:Array{Float64,1} the duration of unemployment in each state [d_unemp_g, d_unemp_b]
        - u:Array{Float64,1} the fraction of people unemployed inn each state [u_g, u_b]

    and returns:

        - Π:{Array,2} the transition matrix Π_z'ze'e
        - An entry of this matix should be read as:
            - Π_z'ze'e[z',e'] = probability of transitioning from state (z,e) to state (z',e')
    """
function trans_mat(d_z, d_u, u)

    d_z = ( d_z .- 1 )./d_z #So 7/8 of the time you will stay

    # transition probabilities between states: [[π_gg, π_gb][π_bg, π_bb]]
    Π_z = [d_z[1] 1-d_z[1]; 1-d_z[2] d_z[2]]

    # transition probabilities
    Π = zeros(4,4)
    d1 = Diagonal( (d_u .- 1) ./ d_u )
    Π[3:4, 3:4] = d1 + (d1 .* Diagonal([ 0.75, 1.25]))[:,end:-1:1]
    Π[1:2, 3:4] = 1 .- Π[3:4, 3:4]
    Π[3:4, 1:2] = (u .- Π[3:4, 3:4] .* u')./(1 .- u')
    Π[1:2, 1:2] = 1 .- Π[3:4, 1:2]

    return (Π .* repeat(Π_z', 2,2) , Π_z)
end
function test(d_z ;n=1000000) #This function double checks that the above method
#produces "Good times" of the proper average length
    d_z = ( d_z .- 1 )./d_z
    Π_z = [d_z[1] 1-d_z[1]; 1-d_z[2] d_z[2]]
    data=zeros(n)
    data[1]=1;
    for ii=2:n
        r=rand()
        if data[ii-1]==1 && r<Π_z[1,1]
            data[ii]=1
        elseif data[ii-1]==1
            data[ii]=0
        elseif data[ii-1]==0 && r<Π_z[2,1]
            data[ii]=1
        else
            data[ii]=1
        end
    end
    #Find the average length of good times
        LengthOfGoodTimes=[]
        LOGT=0;
        ii=200
        while ii<=n
            if data[ii]==1
                LOGT+=1
            elseif data[ii]==0
                if LOGT!=0
                    push!(LengthOfGoodTimes,LOGT)
                    LOGT=0
```

```
                end
            end
            ii+=1
        end
        return sum(LengthOfGoodTimes)/length(LengthOfGoodTimes)
end
#test([8 8])




# Set up the primitives
@with_kw mutable struct Primitives
    # Parameters of the model

    # Stochastic processes
    nZ     ::Int64          = 2                                  # number of states
    nE     ::Int64          = 2                                  # number of employment status
    d_z    ::Array{Float64} = [8, 8]                             # Duration of states [d_g, d_b]
    d_u    ::Array{Float64} = [1.5, 2.5]                         # Duration of unemployment in each state [d
    u      ::Array{Float64} = [0.04, 0.1]                        # Fraction of people unemployed in each sta
    z_val  ::Array{Float64} = [1.01, 0.99]                       # aggregate technology shocks
    e_val  ::Array{Int64}   = [1, 0]                             # employment status
    e_bar  ::Float64        = 0.3271                             # labor efficiency per unit of time worked)
    L_vals ::Array{Float64} = e_bar .* [1 - u[1] , 1 - u[2]]
# Aggregate Labor supply
    # # # Preferences
    β      ::Float64          = 0.99                              # Discount factor
    util   ::Function         = (c) -> (c > 0) ? log(c) : -Inf   # Utility function

    # # # Production
    α      ::Float64          = 0.36                              # Capital labor ratio
    y      ::Function         = (z, k, l) -> z * k^α * l^(1-α)   # Production function
    δ      ::Float64          = 0.025                             # Capital depreciation rate
    ē      ::Float64          = 0.3271                            # Labor efficiency (per hour worked)

    # # # Initial Conditions
    nk     ::Int64            = 20                                # Number of grid points for capital
    k_min  ::Float64          = 10.0                             # Minimum capital
    k_max  ::Float64          = 15.0                              # Maximum capital
    # Change: using a logaritythmic grid to better deal with concavity of the value function
    # k_grid ::Array{Float64}   = exp.(range(log(k_min), stop=log(k_max), length=nk))
# Capital grid
    k_grid ::Array{Float64,1} = range(k_min, length = nk, stop = k_max)# Capital grid

    #L_g    ::Float64          = 1 - u[1]                          # Aggregate labor from the good state
    #L_b    ::Float64          = 1 - u[2]                          # Aggregate labor from the bad state
    #π      ::Float64          = (d_z[1] - 1) / d_z[1]             # Long-run probability of being in good s
    #L_ss   ::Float64          = L_ss = π*L_g + (1-π)*L_b
    K_ss   ::Float64          = 11.55 #(α/((1/β) + δ - 1))^(1/(1-α))*L_ss

    K_min  ::Float64          = 10.0
    K_max  ::Float64          = 15.0
    nK     ::Int64            = 11                                 # Number of grid points for capital
    # Change: using a logaritythmic grid to better deal with concavity of the value function
    # K_grid ::Array{Float64}   = exp.(range(log(K_min), stop=log(K_max), length=nK))
    K_grid ::Array{Float64,1} = range(K_min, length = nK, stop = K_max) # Aggregate Capital grid

    T      ::Int              = 10000                             # Number of periods
    T_burn ::Int              = 1000                              # Number of periods to discard
    N      ::Int              = 5000                              # Number of agents

    # First order conditions of the firm
    w_mkt  ::Function         = (K, L, z) -> (1 - α)*z*(K/L)^α
    r_mkt  ::Function         = (K, L, z) -> α*z*(K/L)^(α-1)
# Wage rate


    # Conjecture of functional form for h1:
    # The congecture will be a log linear function recieves the following input:
    #   - z::Int64 the technology shock
    #   - a::Array{Float64} log linear coefficients in case of good productivity shock
    #   - b::Array{Float64} log linear coefficients in case of bad productivity shock
    k_forecast ::Function     = (z, a, b, k_last) -> ( z == 1 ) ? exp(a[1]+a[2]*log(k_last)) :
exp(b[1]+b[2]*log(k_last))

end

function generate_shocks(prim)
    # TODO: This part is dependent on being two states with symmetric distributions, should be generalized
```

```
    @unpack N, T, T_burn, d_z, d_u, u = prim

    # Transition Matrices
    Π, Π_z  = trans_mat(d_z, d_u, u)

    Π = Π'

    z_seq  ::Array{Int64, 1}  = vcat(1, zeros(T-1+T_burn))                # Technology shocks
    for t ∈ 2:length(z_seq)
        temp = rand(1)[1]
        z_seq[t] = (temp < Π_z[ Int(z_seq[t-1])]) ? 1 : 2          # Generate the sequence of shocks
    end

    employment ::Array{Int64, 2} = zeros(N, T+T_burn)
    employment[:,1] .= 1                             # Agent's employment status
    for t ∈ 2:T+T_burn
        z_last = z_seq[t-1]
        z_now  = z_seq[t]
        for n ∈ 1:N
            temp = rand(1)[1]
            e_last = employment[n, t-1]
            ind_1 = 2(1 - e_last) + z_last
            prob_emp = Π[ind_1, z_now]
            employment[n,t] = ( temp < prob_emp ) ? 1 : 0
        end
        # if t < 100
        #     println("t = ", t, " Total employed = ", sum(employment[:, t]))
        # end
    end

    return (Π, Π_z,  z_seq, employment)
end

# Set structure of the model regarding stochastic shocks
struct Shocks
    Π      ::Array{Float64,2}                                     # Transition matrix Π_z'ze'e
    Π_z    ::Array{Float64,2}                                     # Transition matrix Π_z'z
    z_seq  ::Array{Int64, 1}                                      # Technology shocks
    employment      ::Array{Int64, 2}                                      # Agent's employment status
end

# Structure to hold the results
mutable struct Results
    # TODO: Generalize sizes
    # We are going to define the val_fun and pol_fun as 4-dimenstional objects
    # v[:,:,z,e] gives the value functon for all posiible (k,K) combiantions for a particular (z,e) combination
    val_fun ::Array{Float64, 4} # Value function
    pol_fun ::Array{Float64, 4} # Policy function

    # we are also going to generate an iteration object for each of the functons
    # these will actually be a collection of interpolation objects for each combiantion (z,e)
    # We will store this objects in a dictionary, the idea is that the dictionary keys are (i,j)
    # the index of z and e this is convenient for accesing purpuses later on
    val_fun_interp ::Dict
    pol_fun_interp ::Dict
    a       ::Array{Float64}    # log linear coefficients in case of good productivity shock
    b       ::Array{Float64}    # log linear coefficients in case of bad productivity shock

    # We can pre_allocate the forecast of capital and we wont have to calculate it every time
    k_forecast_grid ::Array{Float64, 2} # Grid of capital for the forecast

    # We also need to store the saving behavior of all the agens in the economy
    V       ::Array{Float64, 2} # Saving Behavior of all agents

end



# Function to initialize the model
function Initialize()

    # Initialize the primitives
    prim = Primitives()
    @unpack nZ, nK, k_grid, K_grid, T, N, k_forecast = prim
    # Initialize the shocks
    Π, Π_z = trans_mat(prim.d_z, prim.d_u, prim.u)
# Transition matrix

    # Initialize the results
    # Initialize the value function and the policy function
    val_fun = zeros(prim.nk, prim.nK, prim.nZ, prim.nE)
```

```
    pol_fun = copy(val_fun)
    val_fun_interp = Dict()
    pol_fun_interp = Dict()
    # TODO: Generalize sizes
    for i ∈ 1:prim.nZ
        for j ∈ 1:prim.nE
            val_fun_interp[(i,j)] = LinearInterpolation( (k_grid, K_grid) , val_fun[:,:, i, j], extrapolation_bc=
            pol_fun_interp[(i,j)] = LinearInterpolation( (k_grid, K_grid) , pol_fun[:,:, i, j], extrapolation_bc=
        end
    end
    # Initialize the regression coefficients
    a = [0.095, 0.999]
    b = [0.085, 0.999]

    k_forecast_grid = zeros(nK, nZ)
    k_forecast_grid[:, 1] = k_forecast.(1, Ref(a), Ref(b), K_grid)
    k_forecast_grid[:, 2] = k_forecast.(2, Ref(a), Ref(b), K_grid)

    V = zeros(N, T)

    Π, Π_z, z_seq, employment = generate_shocks( prim )

    shocks = Shocks(Π, Π_z, z_seq, employment)
    res = Results(val_fun, pol_fun, val_fun_interp, pol_fun_interp, a, b, k_forecast_grid, V)

    return (prim, res, shocks)
end


# Populate Bellman
function Bellman(prim::Primitives, res::Results, shocks::Shocks; use_dense_grid::Bool=false)
    # retrieve relevant primitives and results
    @unpack k_grid, K_grid, nk, nK, nZ, nE, ē, w_mkt, r_mkt, β, δ, k_forecast, z_val, e_val, u, y, util, k_min, k
prim
    @unpack a, b, val_fun, val_fun_interp, k_forecast_grid = res
    @unpack Π = shocks

    # loop through aggregate shocks
    for zi = 1:nZ

        # save aggregate shock and relevant variables
        z = z_val[zi]   # productivity
        L = (1 - u[zi])*ē         # aggregate effective labor

        # loop through aggregate capital
        for Ki = 1:nK
            # save remaining aggregate state space variables
            K = K_grid[Ki]  # aggregate capital

            # calculate prices
            r, w = r_mkt(K, L, z), w_mkt(K, L, z)

            # estimate next period capital
            # Knext = k_forecast(z, a, b, K)
            Knext = k_forecast_grid[Ki, zi]


            # ! Can be the case that Knext > Kmax in that case we need to decide if
            # ! we want to censurate the value of Knext or use extrapolation with the
            # ! interpolation object
            # ! I think we should extrapolate because in the example thta I ran
            # ! the last 3 K values will be the same if we censor
            # * For now I will censor to see if it works but:
            # * Testing  extrapolation

            Knext = min(Knext, prim.K_max)
            # loop through individual state spaces
            for ei = 1:nE

                # initialize last candidate (for exploiting monotonicity)
                cand_last = 1

                # determine shock index from z and e index
                ezi = 2*(zi - 1) + ei
                e = e_val[ei]       # employment status

                # loop through capital holdings
                for ki = 1:prim.nk

                        # save state space variables
                        k = k_grid[ki]      # current period capital
```

```
                        cand_max = -Inf     # intial value maximum
                        pol_max  = 1        # policy function maximum
                        budget   = r*k + w*e*ē + (1-δ)*k

                        if use_dense_grid

                            if ki == 1
                                k_grid_dense = range(prim.k_min, stop= prim.k_max, step=0.01)
                            else
                                k_grid_dense = range(prim.k_min, stop= prim.k_max, step=0.01)
                            end


                            # We can use interpolated versions of the value function to find the next period valu
                            c_pos = budget .- k_grid_dense
                            k_grid_dense = k_grid_dense[c_pos .> 0 ]
                            c_pos = c_pos[c_pos .> 0 ]
                            utils = util.( c_pos )
                            fut_vals = [res.val_fun_interp[(i, j)].(k_grid_dense, Knext) for i ∈
1:2 for j ∈ 1:2];
                            fut_vals = hcat(fut_vals...)
                            # if ki == 1 && Ki == 1
                            #     println(k_grid_dense)
                            # end
                            exp_val_next = [shocks.Π[ezi, :]' * fut_vals[i,:] for i ∈
1:size(fut_vals,1)]

                            cont_val = utils + β*exp_val_next
                            cand_last = k_grid_dense[argmax(cont_val)]
                            cand_max = maximum(cont_val)
                            pol_max  = argmax(cont_val)
                        else
                            # loop through next period capital
                            for kpi = cand_last:prim.nk
                                knext = prim.k_grid[kpi]
                                c = budget - knext

                                # if consumption is negative, skip loop
                                if c < 0
                                    continue
                                end
                                # calculate value at current loop
                                # Calculate the exptecte value of continuation
                                # For this we will use the interpolated version of the value function
                                # since K_tomorrow may not be in the grid
                                # println(knext, " ---- ", Knext)
                                # * Testing: Use extrapolation
                                fut_vals = [res.val_fun_interp[(i, j)](knext, Knext) for i ∈
1:2 for j ∈ 1:2]
                                exp_val_next = shocks.Π[ezi, :]' * fut_vals
                                val = util(c) + β*exp_val_next

                                # update maximum candidate
                                if val > cand_max
                                    cand_max = val
                                    pol_max  = kpi
                                end

                            end # capital policy loop
                        end
                        # update value/policy functions
                        res.val_fun[ki, Ki, zi, ei] = cand_max
                        if use_dense_grid
                            res.pol_fun[ki, Ki, zi, ei] = k_grid_dense[pol_max]
                        else
                            res.pol_fun[ki, Ki, zi, ei] = k_grid[pol_max]
                        end

                    end # individual capital loop
            end # idiosyncratic shock loop
        end # aggregate capital loop
    end # aggregate shock loop
    for i ∈ 1:prim.nZ
        for j ∈ 1:prim.nE
            res.val_fun_interp[(i,j)] = LinearInterpolation( (k_grid, K_grid) , res.val_fun[:,:, i, j], extrapola
            # TODO: Maybe move this to the final stage and interpolate just once
            # TODO: Experiment and report speed gains.
            res.pol_fun_interp[(i,j)] = LinearInterpolation( (k_grid, K_grid) , res.pol_fun[:,:, i, j], extrapola
        end
    end
end # Bellman function
```

```
# Solve consumer's problem: Bellman iteration function
function V_iterate(prim::Primitives, res::Results, shocks::Shocks; use_dense_grid::Bool=false, err::Float64 =
100.0, tol::Float64 = 1e-3)
    n = 0 # iteration counter

    while err > tol
        v_old = copy(res.val_fun)
        Bellman(prim, res, shocks)
        err         = maximum(abs.(v_old .- res.val_fun))
        n += 1
        if n % 100 == 0
            println("Iteration: ", n, " --- ", err)
        end
        if n > 2000
            println("WARNING: Bellman iteration did not converge in 1000 iterations.")
            break
        end
    end

    if err <= tol
        println("Bellman iteration converged in ", n, " iterations.")
    end

end # Bellman iteration

# simulate a time series using the Bellman solution
function Simulation(prim::Primitives, res::Results, shocks::Shocks)
    @unpack T, T_burn, K_ss, N, z_val, u, k_forecast = prim
    @unpack pol_fun, pol_fun_interp = res
    @unpack Π, Π_z, z_seq, employment = shocks

    # begin with good z and steady state for aggregate capital
    K_agg = K_ss

    # In the first period all agents are the same therefore we can initialize the
    # first row of out matrix like:
    temp_V = zeros(N, T+T_burn)
    # The deciction of every agnet is the the decition rule evaluated at
    # the steady state for an angent employed in the good state
    temp_V[:, 1] .= pol_fun_interp[ (1, 1) ]( K_agg, K_agg)

    # We alredy have the time series for shocks we can iterate over it
    for t ∈ 2:T + T_burn
        # println("Simulation: ", t, " K_agg = " , K_agg)
        z = z_seq[t]
        # Now we iterate over all agents for this period
        for n ∈ 1:N
            e = employment[n, t]
            # individual agent decition based on holdings , aggregate capital, productivity shock and employment
            temp_V[n, t] = pol_fun_interp[(z, e+1)]( temp_V[n, t - 1], K_agg)
        end
        # Update aggregate capital at the end of the period
        K_agg = sum(temp_V[:, t])/N
    end # loop over z

    # Update V in the results discarding the burn-in periods
    res.V = temp_V[:, T_burn + 1:T + T_burn]

end # Simulation

# Do (auto)regression with simulated data
function auto_reg(prim::Primitives, res::Results, shocks::Shocks)
    @unpack nZ, N, T, T_burn, k_forecast = prim
    @unpack V, a, b = res
    @unpack z_seq = shocks

    # Remove the burn-in periods in the sequence of productivity shocks
    z_seq = z_seq[T_burn+1:T+T_burn]

    # Calculate aggregate for each period and take logarithms
    K_agg_ts = sum(V, dims=1)/N
    log_K_agg_ts = log.(K_agg_ts)
    #=
    # simulate each agent's holdings for T time periods
    for t = 2:T
        if t % 1000 == 0
            println("Period: ", t, ", K₀ = ", round(K₀, digits = 2))
        end
    end
    =#
```

```
    # Store resutls
    reg_coefs = Dict()

    # Estimate a regression
    for iz ∈ 1:nZ
        K_agg_ts_state = log_K_agg_ts[z_seq .== iz]

        K_agg_next = K_agg_ts_state[2:end]
        K_agg_now = K_agg_ts_state[1:end-1]

        # Create reggression matrix
        X = hcat( ones(length(K_agg_next)), K_agg_now)
        # Calculate regression coefficients
        reg_coefs[iz] = (X'X)^(-1) * X' * K_agg_next
    end

    # Calcualte R_squared
    k_forecasted = zeros(1, T)
    for i in 1:T
        k_forecasted[i] = prim.k_forecast(z_seq[i], reg_coefs[z_seq[i]], reg_coefs[z_seq[i]], K_agg_ts[i])
    end

    SS_res = sum( (K_agg_ts - k_forecasted).^2 )
    mean_K_agg_ts = sum(K_agg_ts)/T
    SS_tot = sum( (K_agg_ts .- mean_K_agg_ts).^2 )
    R_squared = 1 - SS_res/SS_tot

    return reg_coefs, R_squared
end


# Outer-most function that iterates to convergence
function SolveModel(; max_n_iter=30, tol = 1e-4, err = 100, I = 1, use_dense_grid::Bool=false)

        # initialize environment
        println("Initializing Model")
        prim, res, shocks = Initialize()
        println("Model initialized")

        # @unpack a, b = res

        λ = 0.95

        i=1;
        while err>tol && i<=max_n_iter
        #for i in 1:n_iter # Start with few iterations just to see whats happening

            # given current coefficients, solve consumer problem
            V_iterate(prim, res, shocks; use_dense_grid=use_dense_grid)

            # given consumers' policy functions, simulate time series
            println("Simulating time series")
            V = Simulation(prim, res, shocks)

            # Do (auto)regression with simulated data
            reg_coefs, R_squared = auto_reg(prim, res, shocks)

            # Get error:
            err = sum( (res.a - reg_coefs[1]).^2) + sum( (res.b - reg_coefs[2]).^2 )


            # Update regression coefficients
            res.a = reg_coefs[1] #* (1 - λ) + λ * res.a
            res.b = reg_coefs[2] #* (1 - λ) + λ * res.b

            # Re-calculate forcasted capital values
            k_forecast_grid = zeros(prim.nK, prim.nZ)
            k_forecast_grid[:, 1] = prim.k_forecast.(1, Ref(res.a), Ref(res.b), prim.K_grid)
            k_forecast_grid[:, 2] = prim.k_forecast.(2, Ref(res.a), Ref(res.b), prim.K_grid)

            res.k_forecast_grid = k_forecast_grid

            println("Iteration: ", i, " --- ", err, " --- a = ", res.a, " --- b = ", res.b, " --- R² = ", R_squar
            i+=1
        end

        # Iterate until convergence

end # Model solver
```