ECON 899 – Problem Set 4

Danny, Mitchell, Ryan, Yobin, and Hiroaki

December 15, 2021

Exercise 1

1. Use the parametrization from the previous problem set. We continue to assume that labor supply is endogenous. Solve for the stationary equilibrium with social security ($\theta_0^{SS} = 0.11$) and without it ($\theta_N^{SS} = 0$) following the algorithm described in the lecture notes (*Step 1: Calculating the stationary competitive equilibrium*). Denote the initial distribution of agents over age, j, asset holdings, a, and productivity levels, z, by $\Gamma_0^{SS}(z, a, j; \theta_0^{SS})$. Denote the welfare of agents alive in the initial steady state by $V_0^{SS}(z, aj; \theta_0^{SS})$.

2. Compute the transition path of the economy using the algorithm in *Step 2: Solving for the transition path* in the lecture notes. Try N = 30 for the number of periods it approximately takes to get to the new steady state. Obtain and store the value function for the generations in the initial steady state, $V_0(z, a, j; \theta_0^{SS}, \theta_N^{SS})$. Plot the transition paths of interest rate, wage, capital and effective labor. Comment on the results you obtain.

   Answer: During the transition dynamics, capital adjusts gradually. Efficient labor supply jumps in the policy period, which induces jumps in the interest rate and wage at the beginning. Afterward, all variables smoothly transit to the new steady state. Figure 1 contains the results. ∎

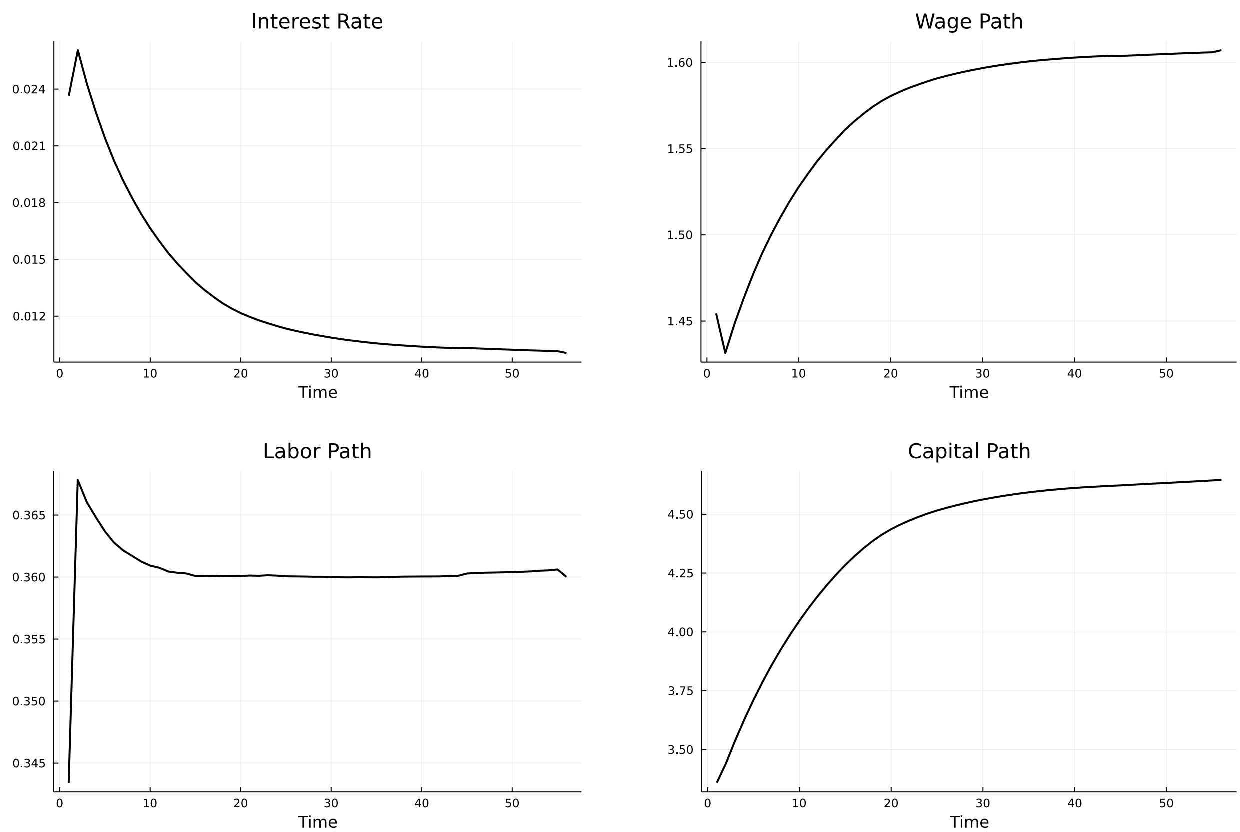

Figure 1: Transition Path: Unanticipated Shock

3. What fraction of the overall population would support the reform? Compute and plot the measure of consumption equivalent variation for each age, EV_j , using

$$EV_j = \sum_z \int_a EV(z, a, j)\Gamma_0^{SS}(z, a, j; \theta_0^{SS})da,$$

with

$$EV(z, a, j) = \left(\frac{V_0(z, a, j; \theta_0^{SS}, \theta_N^{SS})}{V_0^{SS}(z, a, j; \theta_0^{SS})}\right)^{\frac{1}{\gamma(1-\sigma)}}.$$

Discuss the results.

Answer:

All age groups prefer policy with social security, and younger generations benefit the most from having the social security system. Figure 2 shows the results.

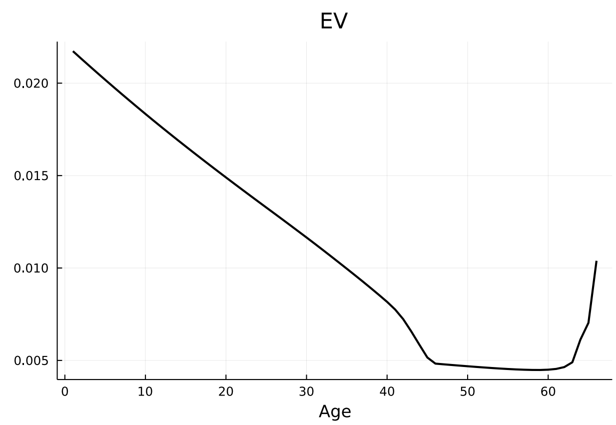

Figure 2: Consumption Equivalent Variation

■

Exercise 2

1. Instead of considering an unexpected elimination of the social security system, assume that in $t = 0$ the government credibly announces that it is going to abolish the public pension system starting from $t = 21$ onwards. Thus, all individuals retired keep their social security benefits, but future retirees anticipate that they will receive only part or no social security benefits. Repeat steps (1)-(3) of exercise 1 to study how agents readjust their plans and how political support changes for the anticipated reform in 21 years. You will have to increase the number of transition periods (try $N = 50$). Discuss your results.

   Answer: With an expected policy change, agents start to adjust from the announcement period, while they also make a relatively bidder adjustment when the policy change happens, especially for labor supply. But capital adjustment is smoother, and most of the adjustment happens before policy changes. With an expected policy change, the welfare loss is lower. Figures 3 and 4 display the results.

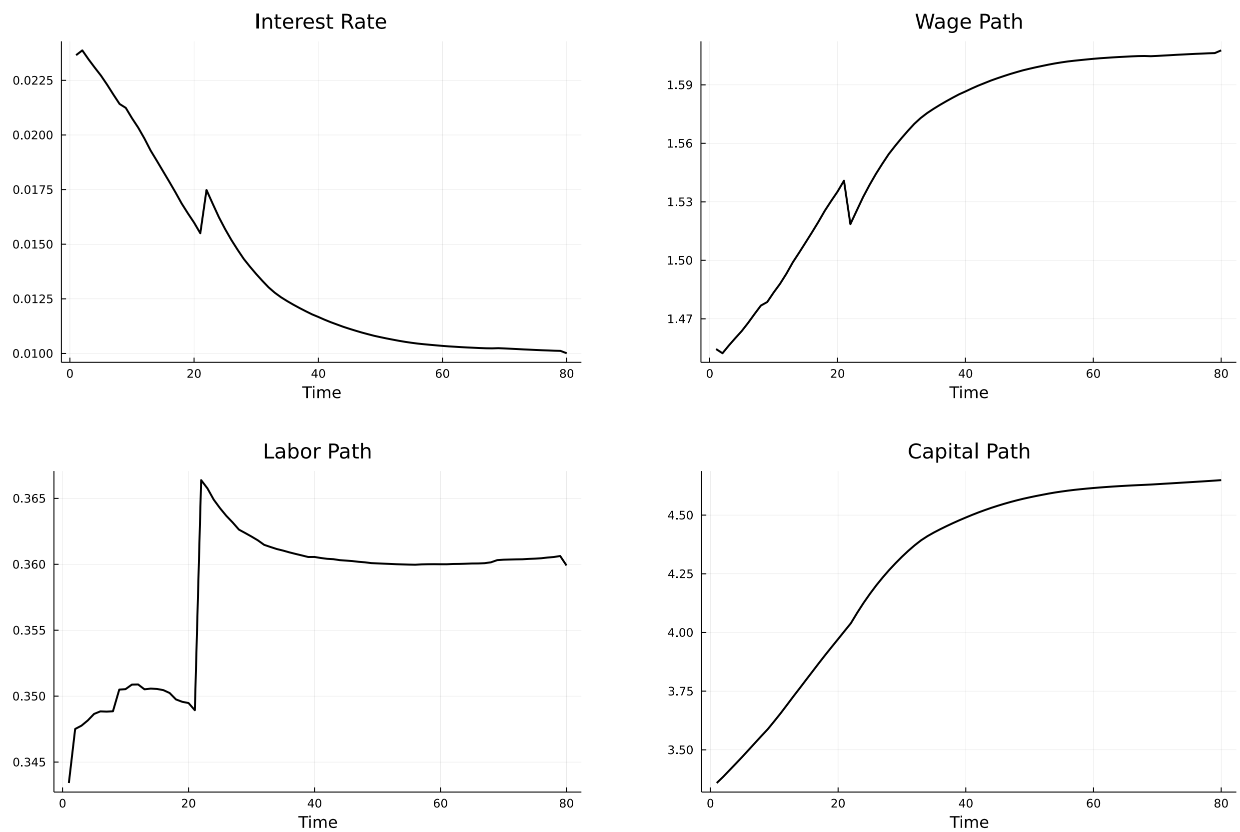

Figure 3: Transition Path: Anticipated Shock

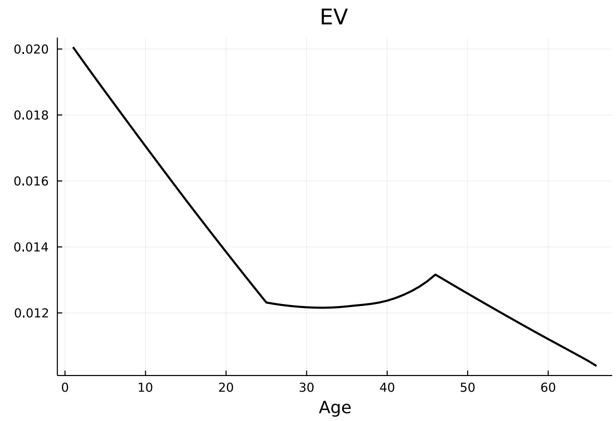

Figure 4: Consumption Equivalent Variation for Exercise 2

■

Appendix

The first code file runs the code.

```
## Changeing current directory
## cd(expanduser("~/pathtoEcon-899/ECON-899/"))
cd(expanduser("~/Box/Econ899/Git/ECON-899/")) # e.g. Hiroaki

using Distributed, SharedArrays, JLD

#add processes
workers()
addprocs(2)

@Distributed.everywhere include("./PS4/JuliaCode/conesa_kueger.jl");
#@Distributed.everywhere include("./conesa_kueger.jl");

#Exercise 1: (Problem Set 4)
@time out_K_path,out_Ft,out_vf_trans= TransitionPath(TrySaveMethod=false,Experiment=1)
#Exercise 2: (Problem Set 4)
@time out_K_path_Exp2,out_Ft_Exp2,out_vf_trans_Exp2=
    TransitionPath(TrySaveMethod=false,Experiment=2)

using Plots, LaTeXStrings

theme(:juno)
plot(1:81,out_K_path[:], ylabel="Aggregate Capital",
    xlabel="Time",label="Excercise 1")
    plot!(1:81,out_K_path_Exp2[:], legend=:bottomright, ylabel="Aggregate Capital",
        xlabel="Time",label="Excercise 2")
    savefig("./PS4/Document/Figures/ComparingTransitions.png")

function Exercise1Prob2(kpath; α=.36,δ=.06,N_final=66,J_R=46, TransitionNo=81)
    #Recalculating Aggregate Labor in the Inelastic case
        L=0.7543 #This was the converged value of inelastic labor supply
    #Functions for interest rate and wages
        r_mkt    ::Function          = (K, L) -> α*(Kˆ(α-1))*(Lˆ(1-α)) - δ
        w_mkt    ::Function          = (K, L) -> (1-α)*(Kˆα)*(Lˆ(-α))
    r_trans=[r_mkt(k,L) for k in kpath]
    w_trans=[w_mkt(k,L) for k in kpath]
    #Plotting Aggregate Capital
        plot(1:TransitionNo,out_K_path[:], ylabel="Aggregate Capital",
            xlabel="Time",legend=false)
        savefig("./PS4/Document/Figures/PathOfAggregateCapital.png")
        plot(1:TransitionNo,r_trans[:], ylabel="Interest Rate",
            xlabel="Time",legend=false)
        savefig("./PS4/Document/Figures/PathOfInterestRate.png")
        plot(1:TransitionNo,w_trans[:], ylabel="Wages",
            xlabel="Time",legend=false)
        savefig("./PS4/Document/Figures/PathOfWages.png")
end
Exercise1Prob2(out_K_path)


#Exercise 3 Problem 1
out_primStart, out_resStart= MarketClearing(use_Fortran=false, tol = 1e-3);
#out_primEnd, out_resEnd= MarketClearing(ss=false, use_Fortran=false, tol = 1e-3);
function SolveProblem3()
    @unpack nZ, nA, N_final, σ = out_primStart
    γ=1
    EV=zeros(nZ,nA,N_final)
    EVj=zeros(N_final)
    PortionWhoAreMadeBetterOff=0;
    for zi=1:nZ
        for ai=1:nA
            for ji=1:N_final
                EV[zi,ai,ji]=(out_vf_trans[ai,zi,ji,1] / out_resStart.val_fun[ai,zi,ji])ˆ(1/(γ*(1-σ)))
                EVj[ji]+=out_resEnd.F[ai,zi,ji]*EV[zi,ai,ji]
                if EV[zi,ai,ji]>1
                    PortionWhoAreMadeBetterOff+=out_resStart.F[ai,zi,ji]
                end
            end #age loop
        end #asset loop
    end #z loop
    print(EVj)
    plot(1:N_final,EVj, ylabel="Consumption Equivalent Variation",
        xlabel="Age",
        title="$(round(PortionWhoAreMadeBetterOff,digits=2)*100)% of the Population would
```

```
Support the Reform", legend=false)
        savefig("./PS4/Document/Figures/Exercise1Problem3.png")
end
SolveProblem3()
```

The second code file contains the relevant functions.

```
@everywhere using Parameters, DelimitedFiles, ProgressBars, SharedArrays, LinearAlgebra

# Define the primitives of the model
@everywhere @with_kw mutable struct  Primitives
    N_final ::Int64             = 66        # Lifespan of the agents
    J_R     ::Int64             = 46        # Retirement age
    n       ::Float64           = 0.011     # Population growth rate
    a_1     ::Float64           = 0         # Initial assets holding for newborns
    θ       ::Float64                       # Labor income tax rate
    γ       ::Float64                       # Utillity weight on consumption
    σ       ::Float64           = 2.0       # Coefficient of relative risk aversion
    α       ::Float64           = 0.36      # Capital share in production
    δ       ::Float64           = 0.06      # Capital depreciation rate
    β       ::Float64           = 0.97      # Discount factor

    # Parameters regarding stochastic processes
    z_H     ::Float64           = 3.0       # Idiosyncratic productivity High
    z_L     ::Float64           = 0.5       # Idiosyncratic productivity Low
    z_Vals  ::Array{Float64}    = [z_H, z_L] # Vector of idiosyncratic productivity values
    nZ      ::Int64             = 2         # Number of idiosynctatic productivity levels
    p_H     ::Float64           = 0.2037    # Probability of z_H at birth
    p_L     ::Float64           = 0.7963    # Probability of z_L at birth
    T       ::Int64             = 1         # time periods between policy announcement and enactment

    # Markov transition matrix for z
    Π       ::Array{Float64,2}  = [0.9261 1-0.9261; 1- 0.9811  0.9811]

    # Functions

    util  ::Function          = (c, l) -> ( c > 0 ) ? ((c^γ * (1 - l)^(1-γ))^(1-σ))/(1-σ) :
-Inf

    # Utility of a retiree
    # Todo: Remove the next 3 lines if everything is working
    # * Note im only using the utility of a worker and setign l = 0 to obtain the utility of a retiree
    # util_R  ::Function          = (c) -> c^(γ*(1-σ))/(1-σ)

    # Optimal labor supply note that last argument is x = (1+r)*a-a_next
    l_opt   ::Function          = (e, w, r, a, a_next) ->  (γ *(1-θ)*e*w-(1-γ)*( (1+r)*a -
a_next ) ) /( (1 - θ)*w*e)

    # Production technology
    w_mkt   ::Function          = (K, L) -> (1-α)*(K^α)*(L^(-α))         # Labor first order condition
    r_mkt   ::Function          = (K, L) -> α*(K^(α-1))*(L^(1-α)) - δ    # Capital first order condition

    # Government budget constraint
    b_mkt   ::Function          = (L, w, m) -> θ*w*L/m   # m is mass of retirees

    # Grids
    # Age efficiency profile
    η       ::Matrix{Float64}   = readdlm("./PS3/Data/ef.txt")
    nA      ::Int64             = 1500      # Size of the asset grid
    a_min   ::Float64           = 0.0       # lower bound of the asset grid
    a_max   ::Float64           = 75.0      # upper bound of the asset grid
    a_grid  ::Array{Float64}    = collect(range(a_min, length = nA, stop = a_max))
# asset grid

end # Primitives

# Structure mutable parameters and results of the model
@everywhere mutable struct Results
    w       ::Float64                       # Wage
    r       ::Float64                       # Interest rate
    b       ::Float64                       # Benefits
    K       ::Float64                       # aggregate capital
    L       ::Float64                       # aggregate labor
    μ       ::Array{Float64, 1}             # Distibution of age cohorts
    val_fun ::SharedArray{Float64, 3}             # Value function
    pol_fun ::SharedArray{Float64, 3}             # Policy function
```

```
    l_fun   ::SharedArray{Float64, 3}             # (effective) Labor policy function

    # ! This is a experiment, maybe it is useful to also save
    # ! the indices of the optimal policy function
    pol_fun_ind ::Array{Int64, 3}           # Policy function indices
    F           ::Array{Float64, 3}         # Distribution of agents over asset holdings
end # Results

# Function that initializes the model
function Initialize(; θArg = 0.11, γArg = 0.42)
    prim = Primitives(θ = θArg, γ = γArg)                # Initialize the primitives
    w = 1.05                                           # Wage guess
    r = 0.05                                           # Interest rate guess
    b = 0.2                                            # Benefits guess
    K = 4                                              # inital capital guess
    val_fun = SharedArray{Float64}(prim.nA, prim.nZ, prim.N_final)    # Initialize the value function
    pol_fun = SharedArray{Float64}(prim.nA, prim.nZ, prim.N_final)    # Initialize the policy function
    pol_fun_ind = SharedArray{Float64}(prim.nA, prim.nZ, prim.N_final)# Initialize the policy function indices
    l_fun = SharedArray{Float64}(prim.nA, prim.nZ, prim.N_final)

    # Before the model starts, we can set the initial value function at the end stage
    # We set the last age group to have a value function consuming all the assets and
    # with a labor supply 0 (i.e. no labor) and recieving a benefit of b
    last_period_value = prim.util.( prim.a_grid .* (1 + r) .+ b, 0 )
    val_fun[: ,: , end] = hcat(last_period_value, last_period_value)

    # Calculate population distribution across age cohorts
    μ = [1.0]
    for i in 2:prim.N_final
        push!(μ, μ[i-1]/(1.0 + prim.n))
    end
    μ = μ/sum(μ)

    L = Float64(sum(μ[1:(prim.J_R-1)]))                  # fixed aggregate labor

    # Finally we initialize the distribution of the agents
    F = zeros(prim.nA, prim.nZ, prim.N_final)
    F[1, 1, 1] = μ[1] * prim.p_H
    F[1, 2, 1] = μ[1] * prim.p_L

    # Initialize the results
    res = Results(w, r, b, K, L, μ, val_fun, pol_fun, l_fun, pol_fun_ind, F)
    return (prim, res)                             # Return the primitives and results
end

# Value funtion for the retirees
function V_ret(prim::Primitives, res::Results)

    # unpack the primitives and the results
    @unpack nA, a_grid, N_final, J_R, util, β = prim
    @unpack b, r = res

    # We obtain for every age group and asset holdings level the value function using backward induction
    for j in N_final-1:-1:J_R
        @sync @distributed for a_index in 1:nA
            a = a_grid[a_index]
            vals = util.(((1+r)*a + b).- a_grid, 0) .+ β*res.val_fun[:, 1, j+1]
            pol_ind = argmax(vals)
            val_max = vals[pol_ind]
            res.pol_fun_ind[a_index, :, j] .= pol_ind
            res.pol_fun[a_index, :, j] .= a_grid[pol_ind]
            res.val_fun[a_index, :, j] .= val_max
            res.l_fun[a_index, :, j] .= 0
        end # for a_index
    end # for j

    return prim, res

end # V_ret

# Value function for the workers
function V_workers(prim::Primitives, res::Results)

    # Unopack the primitives
    @unpack nA, nZ, z_Vals, η, N_final, J_R, util, β, θ, a_grid, Π, l_opt = prim
    @unpack r, w, b = res

    # First  we iterate over the age groups
    for j in ProgressBar(J_R-1:-1:1) # Progressbar for running in console

    #for j in N_final-1:-1:1 # Without progressbar for running in jupyter notebook
```

```
        # Next we iterate over the productivity levels
        @sync @distributed for z_index in 1:nZ
            z = z_Vals[z_index] # Current idiosyncratic productivity level
            #println("Solving for productivity type $z")
            e = z * η[j] # Worker productivity level (only for working age)
            LowestChoiceInd=1 #Exploiting monotonicity in the policy function
            # Next we iterate over the asset grid
            for a_index in 1:nA
                a = a_grid[a_index] # Current asset level
                cand_val = -Inf      # Initialize the candidate value
                cand_pol = 0         # Initialize the candidate policy
                cand_pol_ind = 1     # Initialize the candidate policy index
                l_pol = 0            # Initialize the labor policy

                # Next we iterate over the possible choices of the next period's asset
                #l_grid = l_opt.(e, w, r, a, a_grid) # Labor supply grid
                # if j == 20 && z_index == 2
                #     print("\n a = $a a_next reached:")
                # end
                for an_index in LowestChoiceInd:nA
                    a_next = a_grid[an_index]   # Next period's asset level
                    l = l_opt(e, w, r, a, a_next) #l_grid[an_index]        # Implied labor supply in current peri
                    if l < 0                    # If the labor supply is negative, we set it to zero
                        l = 0
                    elseif l > 1                # If the labor supply is greater than one, we set it to one
                        l = 1
                    end
                    c = w * (1 - θ) * e * l + (1 + r)*a - a_next # Consumption of worker (All people in this loop
                    if c < 0 # If consumption is negative than this (and all future a' values) are unfeasible
                        break
                    end

                    # exp_v_next = val_fun[an_index, :, j+1] * Π[z_index , :] # Expected value of next period
                    # exp_v_next = val_fun[an_index, 1, j+1] * Π[z_index , 1] + val_fun[an_index, 2, j+1] * Π[z_i

                    # calculate expected value of next period
                    exp_v_next = 0
                    for zi = 1:nZ
                        exp_v_next += res.val_fun[an_index, zi, j+1] * Π[z_index ,
zi]
                    end # zi

                    v_next = util(c, l) + β * exp_v_next # next candidate to value function

                    if v_next > cand_val
                        cand_val = v_next       # Update the candidate value
                        cand_pol = a_next       # Candidate to policy function
                        cand_pol_ind = an_index # Candidate to policy function index
                        l_pol = e*l             # Candidate to labor policy function
                    end # if v_next > cand_val

                end # Next period asset choice loop

                res.val_fun[a_index, z_index, j] = cand_val          # Update the value function
                res.pol_fun[a_index, z_index, j] = cand_pol          # Update the policy function
                res.pol_fun_ind[a_index, z_index, j] = cand_pol_ind # Update the policy function index
                res.l_fun[a_index, z_index, j] = l_pol               # Update the labor policy function
                LowestChoiceInd=copy(cand_pol_ind)
            end # Current asset holdings loop
        end # Productivity loop
    end  # Age loop

    return prim, res
end # V_workers

# If we want to speed up the code we can use Fortran
# the following function is a wrapper for the Fortran code

function V_Fortran(prim::Primitives, res::Results)
    @unpack r, w, b =res
    # PS3/FortranCode/conesa_kueger.f90
    # Compile Fortran code
    path = "./PS3/FortranCode/"
    run(`gfortran -fopenmp -O2 -o $(path)V_Fortran $(path)conesa_kueger.f90`)
    # run(`./T_op $q $n_iter`)
    run(`$(path)V_Fortran`)

    results_raw =  readdlm("$(path)results.csv");

    val_fun = zeros(prim.nA, prim.nZ, prim.N_final)    # Initialize the value function
```

```
    pol_fun = zeros(prim.nA, prim.nZ, prim.N_final)    # Initialize the policy function
    pol_fun_ind = zeros(prim.nA, prim.nZ, prim.N_final + 1)    # Initialize the policy function index
    consumption = zeros(prim.nA, prim.nZ, prim.N_final)    # Initialize the consumption function
    l_fun = zeros(prim.nA, prim.nZ, prim.N_final)    # Initialize the labor policy function
    for j in 1:prim.N_final
        range_a = (j-1) * 2*prim.nA + 1 : j * 2*prim.nA |> collect
        val_fun[:,:,j] = hcat(results_raw[range_a[1:prim.nA],end], results_raw[range_a[prim.nA+1:end],end])
        pol_fun_ind[:,:,j] = hcat(results_raw[range_a[1:prim.nA],end-1], results_raw[range_a[prim.nA+1:end],end-1
        pol_fun[:,:,j] = hcat(results_raw[range_a[1:prim.nA],end-2], results_raw[range_a[prim.nA+1:end],end-2])
        consumption[:,:,j]   = hcat(results_raw[range_a[1:prim.nA],end-3], results_raw[range_a[prim.nA+1:end],end
        l_fun[:,:,j] = hcat(results_raw[range_a[1:prim.nA],end-4], results_raw[range_a[prim.nA+1:end],end-4])
    end
    A_grid_fortran = results_raw[1:prim.nA,end-5]
    res.val_fun = val_fun
    res.pol_fun = pol_fun
    res.pol_fun_ind = pol_fun_ind
    res.l_fun = l_fun
    return A_grid_fortran, consumption

end # run_Fortran()


# Function to obtain the steady state distribution
function SteadyStateDist(prim::Primitives, res::Results)
    # Initialize the steady state distribution
    res.F[:,:,2:end] .= zeros(prim.nA, prim.nZ)
    # Unpack the primitives
    @unpack N_final, n, p_L, p_H, nZ, nA, Π, a_grid = prim

    # Finding relative size of each age cohort

    # Finding the steady state distribution
    for j in 2:N_final
        for z_ind in 1:nZ
            for a_ind in 1:nA
                a_next_ind = argmin(abs.(res.pol_fun[a_ind, z_ind, j-1].-a_grid))
                for zi = 1:nZ
                    res.F[a_next_ind, zi, j] += res.F[a_ind, z_ind, j-1] * Π[z_ind, zi] *
(res.μ[j]/res.μ[j-1])
                end # zi
            end
        end # z_ind
    end # j loop

end # SteadyStateDist

# Function to solve for market prices
function MarketClearing(; ss::Bool=true, use_Fortran::Bool=false, λ::Float64=0.7, tol::Float64=1e-2, err::Float64

    # initialize struct according to policies. Note that we are assuming that labor is supplied inelastically
    if ~ss
        prim, res = Initialize(θArg = 0, γArg=1)
    else
        prim, res = Initialize(γArg=1)
    end


    # unpack relevant variables and functions
    @unpack w_mkt, r_mkt, b_mkt, J_R, a_grid = prim

    n = 0 # loop counter

    # iteratively solve the model until excess savings converge to zero
    while err > tol

        # calculate prices and payments at current K, L, and F
        res.r = r_mkt(res.K, res.L)
        res.w = w_mkt(res.K, res.L)
        res.b = b_mkt(res.L, res.w, sum(res.μ[J_R:end]))

        # solve model with current model and payments
        if use_Fortran
            A_grid_fortran, consumption = V_Fortran(prim,res)
        else
            V_ret(prim, res);
            V_workers(prim, res);
        end
        SteadyStateDist(prim, res);

        # calculate aggregate capital and labor
        K = sum(res.F[:, :, :] .* a_grid)
```

```
        L = sum(res.F[:, :, :] .* res.l_fun) # Labor supply grid

        # calculate error
        err = maximum(abs.([res.K, res.L] - [K, L]))

        if (err > tol*10)
            # Leave λ at the default
        elseif (err > tol*2)
            λ = 0.9
        elseif (err > tol*1.1)
            λ = 0.95
        else
            λ=.99
        end

        # update guess
        res.K = (1-λ)*K + λ*res.K
        res.L = (1-λ)*L + λ*res.L

        n+=1

        println("$n iterations; err = $err, K = ", round(res.K, digits = 4), ", L = ",
        round(res.L, digits = 4), ", λ = $λ")

    end # while err > tol
    return prim, res
end # MarketClearing

# Function to calculate compensating variation
function Lambda(prim::Primitives, res::Results, W::Float64)

    # unpack necessary variables
    @unpack F, val_fun = res
    @unpack α, β = prim

    # calculate and return compensating variation
    λ = ( (W + (1/((1-α)*(1-β)))) ./ (val_fun .+ (1/((1-α)*(1-β)))) ).^(1/(1-α)) .-
1

    return F.*λ

end # Lambda

# Function that iterates backward on decision rules given a path of capital
function IterateBackward(primEnd::Primitives, resEnd::Results,
        primStart::Primitives, resStart::Results,
        K_path::Array{Float64}, N::Int64; Exp::Int64=1)
    #Above, "End" refers to values in the new steady state
    #"Experiment" is 1 for problem 1 of PSet4 and 2 for problem 2 of PSet4
    #Setting up matrices to store the transitions of value and policy functions
        pol_fun_trans = SharedArray{Float64}(primEnd.nA, primEnd.nZ, primEnd.N_final, N+1)
        val_fun_trans = SharedArray{Float64}(primEnd.nA, primEnd.nZ, primEnd.N_final, N+1)
    #Assume Convergence after N periods
        pol_fun_trans[:,:,:,N+1]=resEnd.pol_fun
        val_fun_trans[:,:,:,N+1]=resEnd.val_fun
        #Unpacking relevant parameters
        @unpack nA, nZ, z_Vals, η, a_grid, β, θ, N_final, J_R, util, Π, l_opt,
            r_mkt, w_mkt, b_mkt = primEnd
        @unpack μ, L=resEnd
    println("Iterating Backwards along Transition path...\n")
    for t in ProgressBar(N:(-1):1) #ProgressBar for running in console
        if Exp==1 || t>=21
            #Leave parameter values unchanged
        else #In experiment 2, the current regime stays until t==21
            θ=primStart.θ
        end
        K=K_path[t]
            ## L should be aggregate L ##########################
            r = r_mkt(K, L) #Since labor is inelatically supplied
            w = w_mkt(K, L)
            b = b_mkt(L, w, sum(μ[J_R:end]))
        #Assign the the value associated with life's last period
            last_period_value = util.( a_grid .* (1 + r) .+ b, 0 )
            for zi=1:2 #(N+1) is already filled in above)
                val_fun_trans[:,zi,N_final,t]=last_period_value
            end
        #First for retired folks:
        for j in N_final-1:-1:J_R
            @sync @distributed for a_index in 1:nA
                a = a_grid[a_index]
                vals = util.(((1+r)*a + b).- a_grid, 0) .+ β*val_fun_trans[:,1,j+1,t+1]
```

```
                pol_ind = argmax(vals)
                val_max = vals[pol_ind]
                pol_fun_trans[a_index, :, j, t] .= a_grid[pol_ind]
                val_fun_trans[a_index, :, j, t] .= val_max
                #res.l_fun[a_index, :, j] .= 0
                    #We do not need to record L since we assume it is perfectly inelastic
            end # for a_index
        end # for j
        #Then for the workers:
        for j in J_R-1:-1:1
            # Next we iterate over the productivity levels
            @sync @distributed for z_index in 1:nZ
                z = z_Vals[z_index] # Current idiosyncratic productivity level
                e = z * η[j] # Worker productivity level (only for working age)
                LowestChoiceInd=1 #Exploiting monotonicity in the policy function
                # Next we iterate over the asset grid
                for a_index in 1:nA
                    a = a_grid[a_index] # Current asset level
                    cand_val = -Inf     # Initialize the candidate value
                    cand_pol = 0        # Initialize the candidate policy
                    cand_pol_ind = 1    # Initialize the candidate policy index
                    l_pol = 0           # Initialize the labor policy
                    # Next we iterate over the possible choices of the next period's asset
                    for an_index in LowestChoiceInd:nA
                        a_next = a_grid[an_index]   # Next period's asset level
                        #Calculating the labor supply should be irrelevant if everything is working right
                            #since it is perfectly inelastic:
                            l = l_opt(e, w, r, a, a_next) #l_grid[an_index]
 # Implied labor supply in current period
                            if l < 0                    # If the labor supply is negative, we set it to zero
                                l = 0
                            elseif l > 1                # If the labor supply is greater than one, we set it to o
                                l = 1
                            end
                        c = w * (1 - θ) * e * l + (1 + r)*a - a_next # Consumption of worker (All people in this
                        if c < 0 # If consumption is negative than this (and all future a' values) are unfeasible
                            break
                        end
                        # calculate expected value of next period
                        exp_v_next = 0
                        for zi = 1:nZ
                            exp_v_next += val_fun_trans[an_index, zi, j+1,t+1] *
Π[z_index , zi]
                        end # zi
                        v_next = util(c, l) + β * exp_v_next # next candidate to value function
                        if v_next > cand_val
                            cand_val = v_next       # Update the candidate value
                            cand_pol = a_next       # Candidate to policy function
                            cand_pol_ind = an_index # Candidate to policy function index
                            l_pol = e*l             # Candidate to labor policy function
                        end # if v_next > cand_val
                    end # Next period asset choice loop

                    val_fun_trans[a_index, z_index, j, t] = cand_val         # Update the value function
                    pol_fun_trans[a_index, z_index, j, t] = cand_pol         # Update the policy function
                    LowestChoiceInd=copy(cand_pol_ind)
                end # Current asset holdings loop
            end # Productivity loop
        end  # Age loop for workers
    end #for t
    return pol_fun_trans, val_fun_trans
end #ends IterateBackward

#= I am just commenting this out so as not to delete anyhing
# Function that, given an aggregate capital path, infers prices and calculates a new path
function FillPath(prim::Primitives, res::Results, K_path::::Array{Float64}, N::Int64)

    # upack relevant primitives and results
    @unpack a_grid, N_final, nZ, Π, r_mkt, w_mkt, b_mkt = prim
    @unpack F, μ, L = res

    # initialize transition path distribution
    Ft = SharedArray{Float64}(prim.nA, prim.nZ, prim.N_final, N+1)
    Ft[:, :, :, 1]      = F;
    Ft[:, :, :, 2:end] .= 0;

    # loop through each possible combination of states and project agents'
    # choices, given K_path, from t=1 to t=N
    @async @distributed for t in 2:(N + 1) #I do not think that this can be @async
```

```
        # localize results struct for current time period
        newRes = copy(res)
        # acquire value and policy functions for current K
        K = K_path[t]
        newRes.r = r_mkt(K, L)
        newRes.w = w_mkt(K, L)
        newRes.b = b_mkt(L, res.w, sum(μ[J_R:end]))

        newPrim, newRes = V_ret(prim, newRes);
        newPrim, newRes = V_workers(newPrim, newRes);

        @unpack pol_fun = newRes

        for j in 1:N_final
            for zi in 1:nZ
                for ai in 1:nA
                    api = argmin(abs.(pol_fun[ai, zi, j-1].-a_grid))
                    for znext = 1:nZ
                        Ft[api, zi, j, t] += Ft[ai, zi, j-1, t] * Π[zi, znext] * (μ[j]/μ[j-1])
                    end # znext
                end # ai loop
            end # zi loop
        end # j loop

    end # t loop

    # calculate new capital path and return
    K_path = sum(Ft, dims = 1:3)

    return K_path, Ft
end

=#

#Alternate FillPath (Function that, given an aggregate capital path, infers prices and calculates a new path)
function FillPath(primStart::Primitives,resStart::Results, primEnd::Primitives,
    resEnd::Results, K_path::Array{Float64}, N::Int64; Ex::Int64=1)
    # initialize transition path distribution
    Ft = SharedArray{Float64}(primEnd.nA, primEnd.nZ, primEnd.N_final, N+1)
    Ft[:, :, :, 1]       = resStart.F;
    Ft[:, :, :, 2:end] .= 0; #This line will start the first cohort being born
                            #in each period with 0 assets
    pf_trans, vf_trans = IterateBackward(primEnd,resEnd,primStart,resStart,K_path,N, Exp=Ex)
    K_path_new=zeros(N+1) #Initialize new K_Path
        K_path_new[1]=resStart.K #Set first value to old steady state
    @unpack μ= resEnd
    @unpack N_final, nZ, nA, a_grid, Π  = primEnd
    for t in 2:(N+1)
        #Initialize "newborns" during each period of the transition with zero assets
        Ft[1, 1, 1, t] = μ[1] * primEnd.p_H
        Ft[1, 2, 1, t] = μ[1] * primEnd.p_L
        for j in 2:N_final
            for zi in 1:nZ
                for ai in 1:nA
                    api = argmin(abs.(pf_trans[ai, zi, j-1,t-1].-a_grid))
                    for znext = 1:nZ
                        Ft[api, zi, j, t] += Ft[ai, zi, j-1, t-1] * Π[zi, znext] *
(μ[j]/μ[j-1])
                    end # znext
                end # ai loop
                #Add the capital for this age and z shock to aggregate
                K_path_new[t] += sum(Ft[:, zi, j, t].*a_grid)
            end # zi loop
        end # j loop
        #calculate implied capital path
    end #End t loop
    # This is should be 1: sum(Ft, dims = 1:3)
    return K_path_new, Ft, vf_trans
end


# Function to calculate the transition path between two equilibria
function TransitionPath(;err::Float64=100.0, tol::Float64=1e-3, λ::Float64=0.70,
        N::Int64=80, SolveAgain::Bool=false, TrySaveMethod::Bool=true, Experiment::Int64=1)
    #Finding the two Steady States to transition between
    if TrySaveMethod
        if SolveAgain
            print("Solving for the steady state with Social Security...\n")
            primStart, resStart= MarketClearing(use_Fortran=false, tol = 1e-2);
            print("Solving for the steady state without Social Security...\n")
```

```
            primEnd, resEnd= MarketClearing(ss=false, use_Fortran=false, tol = 1e-2);
            save("/PS4/JuliaCode/SteadyStates.jld",
                "resStart", resStart, "resEnd",resEnd)
        else
            try
                resStart = load("./PS4/JuliaCode/SteadyStates.jld", "resStart")
                resEnd = load("./PS4/JuliaCode/SteadyStates.jld", "resEnd")
                primEnd = Initialize(θArg = 0, γArg=1)[1]
            catch
                print("Could not load the file with the saved steady states.
    Solving for the steady state with Social Security...")
                primStart, resStart= MarketClearing(use_Fortran=false, tol = 1e-2);
                print("Solving for the steady state without Social Security...")
                primEnd, resEnd= MarketClearing(ss=false, use_Fortran=false, tol =
1e-2);
                save("PS4/JuliaCode/SteadyStates.jld", "resStart", resStart, "resEnd",resEnd)
            end
        end
    else
        print("Solving for the steady state with Social Security...\n")
        primStart, resStart= MarketClearing(use_Fortran=false, tol = 1e-3);
        print("Solving for the steady state without Social Security...\n")
        primEnd, resEnd= MarketClearing(ss=false, use_Fortran=false, tol = 1e-3);
    end
    # guess the transition path between the two equilibria
        K₀, Kₜ = resStart.K, resEnd.K
        K_path = collect(range(K₀, Kₜ, length = N + 1))
        Ft=[0]; #Initialize variable name Ft
        vf_trans=copy(Ft)
    # generate new primitives and results structs for use in calculating
        # the transition path
        #newRes      = resEnd;
        #newRes.F    = resStart.F;
    # iteratively use the FillPath function to update capital path until convergence
    iter=1;
    while err > tol
        # pass current capital path to FillPath function
            #K_path_new, Ft = FillPath(primEnd, newRes, K_path, N)
            K_path_new , Ft, vf_trans = FillPath(primStart,resStart, primEnd, resEnd, K_path,
                N,Ex=Experiment)
        # test convergence and update
            err=maximum(abs.(K_path.-K_path_new))
            K_path=copy(K_path_new)
        print("$(iter) iterations; err=$(err)")
    end
    return K_path, Ft, vf_trans
end # TransitionPath
```