This problem set was completed by Danny Edgel, Mitchell Valdes Bobes, Ryan Mather, and Yobin Timilsena.

I. Consider the same environment as Huggett (1993, JEDC) except assume that there are enforceable insurance markets regarding the idiosyncratic shocks to earnings and that there are no initial asset holdings. Solve for a competitive equilibrium. What are prices? What is the allocation? (Hint: think about the planner's problem and then decentralize).

Answer: Under the assumptions of enforceable insurance markets + locally non-satiated preferences, the basic first and second welfare theorems hold. Hence, we will solve the planner's problem for allocations and then decentralize by setting asset prices that support the allocations as a CE.

The planner's problem can be written as

$$\mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t [\pi(e)u(c_{e,t}) + \pi(u)u(c_{u,t})] \quad \text{s.t.} \quad \pi(e)c_{e,t} + \pi(u)c_{u,t} \leq \pi(e)y_t(e) + \pi(u)y_t(u), \ \forall t$$

The first order conditions are

$$[c_{e,t}] : \beta^t \pi(e)u'(c_{e,t}) = \lambda \pi(e) \qquad [c_{u,t}] : \beta^t \pi(u)u'(c_{u,t}) = \lambda \pi(u)$$

Combined, we have that $u'(c_{e,t}) = u'(c_{u,t}) \Leftrightarrow c_{e,t} = c_{u,t} = \bar{c}$.

Plugging into the BC, we get $\bar{c} = y_t(u) + \pi(e)[y_t(e) - y_t(u)]$. For instance, in Part II, we are given $\pi(e) = 0.94$, $y_t(e) = 1$ and $y_t(u) = 0.5$, which would imply $\bar{c} = 0.97$.

The decentralized EE for an individual i is $\beta^t u'(c_t^i) = \lambda q_t \Leftrightarrow q_{t+1} = \beta q_t = \beta^{t+1} q_0$. ■

II. Now compute Huggett (1993, JEDC) with incomplete markets. The following takes you through the steps of solving a simple general equilibrium model that generates an endogenous steady state wealth distribution. The basic algorithm is to: 1) taking a price of discount bonds $q \in [0, 1]$ as given , solve the agent's dynamic programming problem for her decision rule $a' = g_\theta(a, s; q)$ where $a \in A$ are asset holdings, $s \in S \subset R_{++}$ is exogenous earnings, and θ is a parameter vector; 2) given the decision rule and stochastic process for earnings, solve for the invariant wealth distribution $\mu^*(A, S; q)$; 3) given μ^*, check whether the asset market clears at q (i.e. $\int_{A,S} g_\theta(a, s; q)\mu^*(da, ds; q) = 0$). If it is, we are done. If not (i.e. it is not within an acceptable tolerance), then bisect $[0, 1]$ in the direction that clears the market (e.g. if $\int_{A,S} a' \mu^*(da, ds; q) > 0$), then choose a new price $\hat{q} = q + [1 - q]/2$ and go to step 1.

4. After finding fixed points of the T and T^* operators, answer the following questions:

   a. Plot the policy function $g(a, s)$ over a for each s to verify that there exist $\hat{a}$ where $g(\hat{a}, s) < \hat{a}$ as in Figure 1 of Huggett. (Recall this condition establishes an upper bound on the set A necessary to obtain an invariant distribution).

   Answer: The policy function is graphed below in Figure 1. As can be seen, there does exist an $\hat{a}$ beyond which modeled agents always, whether employed or unemployed, dissave on net. ■

   b. What is the equilibrium bond price? Plot the cross-sectional distribution of wealth for those employed and those unemployed on the same graph.

   Answer: The equilibrium bond price is $q = 0.9943074$, and the resulting cross-sectional distribution of wealth is shown in Figure 2. ■

   c. Plot a Lorenz curve. What is the gini index for your economy? Compare them to the data. For this problem set, define wealth as current earnings (think of this as direct deposited into your bank, so it is your cash holdings) plus net assets. Since market clearing implies aggregate assets equal zero, this wealth definition avoids division by zero in computing the Gini and Lorenz curve.

   Answer: The Lorenz curve is shown in Figure 3, and implies a Gini coefficient of 0.29969592. The true Income Gini coefficient for the US economy in 2020 was .458,[1] so if we take this to be the relevant statistic

[1] U.S. Census Bureau, Current Population Survey, 1968 to 2021 Annual Social and Economic Supplements (CPS ASEC), Table F-4

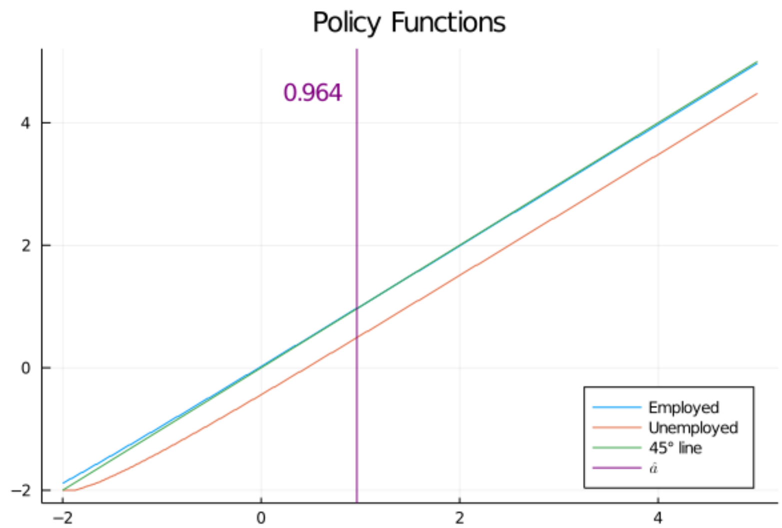

Figure 1: Problem 4(a)

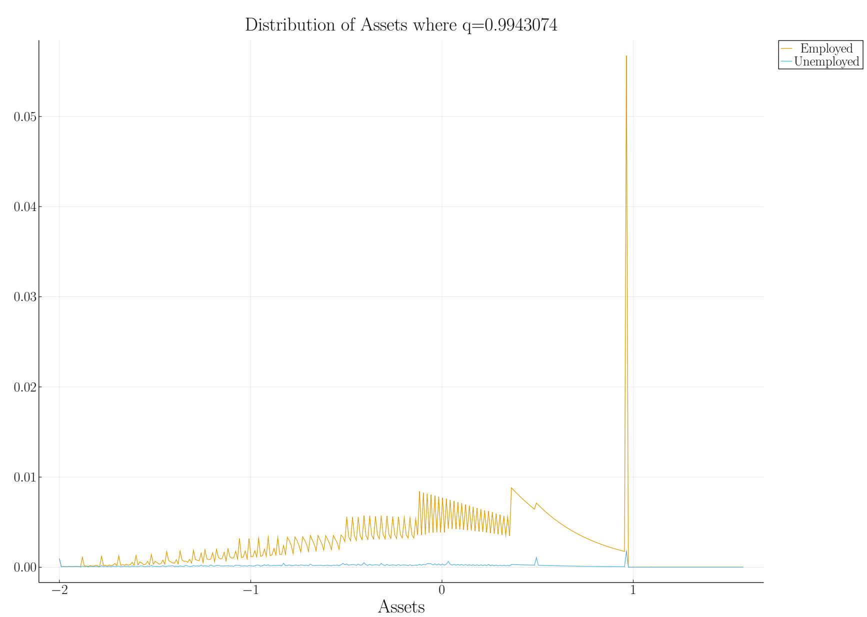

Figure 2: Problem 4(b)

against which to compare our model results, our model explains about 65% of inequality as captured by the Gini Coefficient. ∎

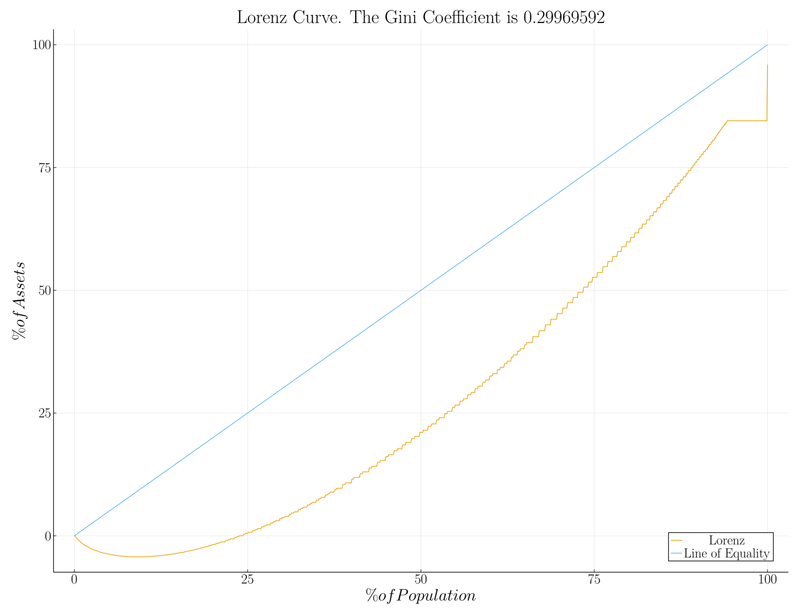

Figure 3: Problem 4(c)

III. (a) Plot $\lambda(a, s)$ across a for both $s = e$ and $s = u$ in the same graph.

Answer:

∎

(b) What is W^{FB}? What is $W^{INC} = \sum_{(a,s)\in A\times S} \mu(a, s)\nu(a, s)$? What is WG?

Answer:

∎

(c) What fraction of the population would favor changing to complete markets? That is $\sum_{(a,s)\in A\times S} \mathbb{1}_{\lambda(a,s)\geq 0}(a, s)\mu(a, s)$.

Answer:

∎

Code Appendix

This is the “computation” code that does most of the numerical work involved for the solutions above:

```
#keyword-enabled structure to hold model primitives
@everywhere @with_kw struct Primitives
    β::Float64 = 0.9932 #discount rate
    α::Float64 = 1.5 #capital share
    S_grid::Array{Float64,1} = [1, 0.5] #Earnings when employed and unemployed
    Π::Array{Float64,2} = [.97 .03; .5 .5] #Transition Matrix between employment and unemployment
    na::Int64 = 700 #Number of asset grid points
    a_grid::Array{Float64,1} = collect(range(-2.0,length=na,5.0))
end

#structure that holds model results
@everywhere mutable struct Results
    val_func::Array{Float64, 2} #value function
```

```
    pol_func::Array{Float64, 2} #policy function
    q::Float64
    q_Bounds::Array{Float64,1}
end

#function for initializing model primitives and results
function Initialize()
    prim = Primitives() #initialize primtiives
    val_func = zeros(prim.na,2) #initial value function guess
    pol_func = zeros(prim.na,2) #zeros(prim.nk,2) #initial policy function guess
    q = (prim.β+1)/2
    q_Bounds=[prim.β, 1]
    res = Results(val_func, pol_func, q, q_Bounds) #initialize results struct
    prim, res #return deliverables
end

#Making a function for the inner loop
@everywhere module IL

    function Find_ap(S_index,a_index,res,prim)
        #unpack model primitives
            a_grid, β, α, na, Π, S_grid = prim.a_grid, prim.β, prim.α,
                prim.na, prim.Π, prim.S_grid #unpack model primitives
        #Utility Function
            function U(x)
                if x<0
                    return -Inf
                else
                    return (x^(1-α)-1)/(1-α)
                end
            end
        #Exploiting Monotonicity of V
        budget=S_grid[S_index] + a_grid[a_index];
        #Search for val_max in the found interval.
        max_val,max_ap=-Inf,0
        for ap_index=1:na
            val=U(budget - res.q*a_grid[ap_index]) +
                β*(transpose(Π[S_index,:])*[ res.val_func[ap_index,1] ; res.val_func[ap_index,2]])
            if val>max_val
                max_val=val;
                max_ap=ap_index;
            elseif val<max_val
                break #The Value function is now declining
            end
        end
        return [max_val, max_ap]
    end
end


#Bellman Operator
function Bellman(prim::Primitives,res::Results)
    @unpack na, a_grid = prim
    v_next = zeros(na,2) #next guess of value function to fill
    for S_index=1:2
        out=pmap(a_index -> IL.Find_ap(S_index,a_index,res,prim),1:na)
        for a_index=1:na #Unpacking the pmap results
            v_next[a_index,S_index]=out[a_index][1]
            res.pol_func[a_index,S_index]=out[a_index][2]
        end
    end
    v_next #return next guess of value function
end



#Value function iteration
function V_iterate(prim::Primitives, res::Results; tol::Float64 = 1e-4, err::Float64 =
100.0)
    n = 0 #counter
    while err>tol #begin iteration
        v_next = Bellman(prim, res) #spit out new vectors
        err = abs.(maximum(v_next.-res.val_func))/abs(v_next[prim.na, 1]) #reset error level
        res.val_func = .8*v_next+.2*res.val_func #update value function
        n+=1
        if mod(n,50)==0
            println("Value Function iteration $(n), Error $(err)")
        end
    end
    println("Value function converged in ", n, " iterations.")
end
```

```
#Market clearing for assets/sets a new q
function MC_assets(prim,res; dist_tol::Float64 = 1e-6, dist_err::Float64 = 100.0, ES_tol=1e-2, Done=false)
    @unpack Π, na, a_grid= prim
    TransMat=zeros(2*na,2*na) #The first na points are for employed folks, and the next na are for unemployed
    for a_index=1:na
        TransMat[Int64(res.pol_func[a_index,1]),a_index]=Π[1,1] #Savings choice for those moving from emp->emp
        TransMat[Int64(res.pol_func[a_index,1])+na,a_index]=Π[1,2] #Savings choice for those moving from emp->une
        TransMat[Int64(res.pol_func[a_index,2]),a_index+na]=Π[2,1] #Savings choice for those moving from unemp->e
        TransMat[Int64(res.pol_func[a_index,2])+na,a_index+na]=Π[2,2] #Savings choice for those moving from unemp
    end
    Dist=ones(2*na)*(1/(2*na)) #
    Dist_new=copy(Dist)
    while dist_err>dist_tol
        for i=1:20  #Iterate until we reach the steady-state distribution
            Dist_new=TransMat*Dist_new
        end
        dist_err=abs.(maximum(Dist_new.-Dist))
        Dist=copy(Dist_new)
    end
    #Find Excess Supply and reset q
        ExcessSupply=transpose(Dist)*vcat(a_grid,a_grid)

        if abs(ExcessSupply)>ES_tol
            #Do variant of Bisection Method
            if ExcessSupply<0
                res.q_Bounds[2]=res.q
                #Weight slightly toward old q to avoid wild fluctuations
                res.q=res.q_Bounds[1]*.3+ res.q_Bounds[2]*.7
            else
                res.q_Bounds[1]=res.q
                res.q=res.q_Bounds[1]*.7 +res.q_Bounds[2]*.3
            end
            print("Excess Supply: $(ExcessSupply), q:$(res.q)")
        else
            Done=true
        end
        return Done
end

#solve the model
function Solve_model() #prim::Primitives, res::Results)
    prim, res = Initialize()
    converged=false
    Outer_loop_Iter=1
    while ~converged && Outer_loop_Iter<1000
        println("Beginning Asset Clearing Loop $(Outer_loop_Iter)")
        V_iterate(prim, res)
        converged=MC_assets(prim,res)
        Outer_loop_Iter+=1
    end
    return prim, res
end


#Get Distribution for Plotting
function FindDist_ForPlot(prim,res; dist_tol::Float64 = 1e-6, dist_err::Float64 =
100.0,)
    @unpack Π, na, a_grid= prim
    TransMat=zeros(2*na,2*na) #The first na points are for employed folks, and the next na are for unemployed
    for a_index=1:na
        TransMat[Int64(res.pol_func[a_index,1]),a_index]=Π[1,1] #Savings choice for those moving from emp->emp
        TransMat[Int64(res.pol_func[a_index,1])+na,a_index]=Π[1,2] #Savings choice for those moveing from emp->un
        TransMat[Int64(res.pol_func[a_index,2]),a_index+na]=Π[2,1] #Savings choice for those moving from unemp->e
        TransMat[Int64(res.pol_func[a_index,2])+na,a_index+na]=Π[2,2] #Savings choice for those moving from unemp
    end
    Dist=ones(2*na)*(1/(2*na)) #
    Dist_new=copy(Dist)
    while dist_err>dist_tol
        for i=1:20  #Iterate until we reach the steady-state distribution
            Dist_new=TransMat*Dist_new
        end
        dist_err=abs.(maximum(Dist_new.-Dist))
        Dist=copy(Dist_new)
    end
    return Dist, Dist[1:na].+Dist[(na+1):2*na]
end
##############################################################################
```

This code calls the “computation” code above and then prints some figures:

```
#Getting the Parellel Ready
    using Distributed #, SharedArrays
    #Re-initializing the workers
        rmprocs(workers())
        addprocs(6)
    @everywhere using Parameters
#Saving Details
    include("Compute_Draft1.jl")
#Solve the Model
    #initialize primitive and results structs
    @time out_primitives, out_results = Solve_model() #solve the model!
    @unpack val_func, pol_func = out_results
    @unpack a_grid, na, S_grid = out_primitives

#Plotting results
using Plots, LaTeXStrings #import the libraries we want
Plots.plot(a_grid, val_func[:,1], title="Value Function", label="Employed")
    plot!(a_grid, val_func[:,2], label="Unemployed")
    Plots.savefig("Value_Functions.png")
    #Plotting Policy functions
        function PolicyPolots()
            a_hat=0
            for ai=1:na
                if a_grid[Int64.(pol_func[ai,1])]<=a_grid[ai]
                    a_hat=[a_grid[ai]];
                    break
                end
            end
            Plots.plot(a_grid, a_grid[Int64.(pol_func[:,1])], title="Policy Functions", label="Employed")
                plot!(a_grid, a_grid[Int64.(pol_func[:,2])], label="Unemployed")
                plot!(a_grid,a_grid, label="45 line", legend=:bottomright)
                vline!(a_hat, label=L"\hat{a}",color=:purple)
                annotate!(a_hat[1]-.15, 4.5, text("$(round(a_hat[1],digits=3))", :purple, :right, 12))
                Plots.savefig("Policy_Functions.png")
        end
        PolicyPolots()
    #Plotting Distribution
        function DistPlots()
            TS_Distribution, SS_WealthDistribution=FindDist_ForPlot(out_primitives,out_results)
            MaxNonZero=1
            ForDistPlot=copy(TS_Distribution)
            for i=1:na
                if ForDistPlot[i]==0
                    ForDistPlot[i]=NaN
                end
                if ForDistPlot[na+i]==0
                    ForDistPlot[na+i]=NaN
                end
                if SS_WealthDistribution[i]!=0
                    MaxNonZero=copy(i)
                end
            end
            Plots.plot(a_grid[1:MaxNonZero], ForDistPlot[1:MaxNonZero], title="Distribution of Assets
where q=$(round(out_results.q,digits=8))",
                label="Employed")
                plot!(a_grid[1:MaxNonZero], ForDistPlot[(na+1):na+MaxNonZero], label="Unemployed", xlabel="Assets
                Plots.savefig("Distribution.png")
            #Lorenz Curve
            n_lorenz=1000
            Lorenz=zeros(n_lorenz,2)
                Lorenz[:,1]=collect(range(0,length=n_lorenz,1)) #First column is percent of population
                i=1
                for a_index=1:na
                    if sum(SS_WealthDistribution[1:a_index])<=Lorenz[i,1]
                        Lorenz[i,2]=Lorenz[i,2]+TS_Distribution[a_index]*(a_grid[a_index]+S_grid[1]) +
                            TS_Distribution[na+a_index]*(a_grid[a_index]+S_grid[2]) #Second column is cumulative
                    else
                        while sum(SS_WealthDistribution[1:a_index])>Lorenz[i,1]
                            i+=1
                            Lorenz[i,2]=Lorenz[i-1,2]+0; #copy over the previous cumulative wealth
                        end
                    end
                    Lorenz[i,2]=Lorenz[i,2]+TS_Distribution[a_index]*(a_grid[a_index]+S_grid[1]) +
                        TS_Distribution[na+a_index]*(a_grid[a_index]+S_grid[2])
                end
                #Calculating Gini
                Gini=sum(Lorenz[:,1].-Lorenz[:,2])/(sum(Lorenz[:,1].-Lorenz[:,2])+sum(Lorenz[:,1]))
```

```
                #Lorenz[:,2]=Lorenz[:,2]./Lorenz[n_lorenz,2] #express cumulative assets as a percentage
                #print(Lorenz)
                Plots.plot(100*Lorenz[:,1],100*Lorenz[:,2], title="Lorenz Curve.
The Gini Coefficient is $(round(Gini,digits=8))",
                xlabel="% of Population",
                    ylabel="% of Assets", legend=:bottomright, label="Lorenz")
                    plot!(100*Lorenz[:,1],100*Lorenz[:,1], label="Line of Equality")
                    Plots.savefig("Lorenz.png")
        end
        DistPlots()
#
```