A: the household's problem

Write the household's problem recursively. Be sure to state what variables are chosen and all the constraints.

See the code used in part F

B: The PolyBasis function

I am completing this problem set in Julia. Note that Julia allows function overloading – I can define the same function twice, with different input types, and the compiler will know which I want to use based on the types of arguments I use. This is very commonly used in Julia. I use it below to define PolyBasis for

```julia
PolyBasis(A::AbstractArray, lnY::AbstractArray) = [ones(size(A)) A lnY A .^ 2 A .* lnY lnY .^ 2]  # n x 6
PolyBasis(A::Real, lnY::Real) = [1 A lnY A^2 A * lnY lnY^2]  # 1 x 6
```

C: The PolyGetCoeff function

```
PolyGetCoef(A, lnY, V) = PolyBasis(A, lnY) \ V  # 6 x 1
```

D: Parameters

```julia
# Parameters Given (tell the compiler that this are static)
@consts begin
    γ = 2
    β = 0.94
    μ = 1
    bb = 0.4
    ρ = 0.9
    σ² = 0.01
    σ = σ²^(1 / 2)
    r = 0.05
    Z = 20
end
α = 1 / 3  # allow α to change for later problems
```

E: Create the Grid

Create a grid on A and log Y . Feel free to use McKay's tauchen function as needed. Use 7 grid points for log Y and 100 grid points for A. (Please create the grid for log Y , not Y . Uniformity will make grading easier.) Calculate the steady-state of A using the first-order conditions of the sequence problem and the budget constraint. Then create an equally spaced grid from $0.05\bar{A}$ to $1.95\bar{A}$, where $\bar{A}$ is the steady state.

I translated the tauchen function to Julia (see Appendix).

```julia
# Create a grid for lnY
nY = 7;  # number of points in our grid for lnY
n_sdlnY = 2;  # number of standard deviations to cover with the grid
# Note that we need to use log(μ) due to the formula used in tauchen()
GridlnY, GridPlnY = tauchen(nY, log(μ), ρ, σ, n_sdlnY)
# tauchen() imported from Tauchen.jl

GridPlnY = GridPlnY'  # this is a 7 x 7 transition matrix for which the columns sum to 1
# the (i,j) element is the probability of moving from j to i.

# Using the BC and EE, solve for the steady state asset level
Aₛₛ = Z * (β * Z * α)^(α / (1 - α))

# Create a grid from 0.05Aₛₛ to 1.95Aₛₛ
nA = 100  # number of points in our grid for A
GridA = range(0.05 * Aₛₛ, 1.95 * Aₛₛ, length=nA)

# Cartisian product of the grids, then decompose
AY = [(a, y) for y ∈ GridlnY for a ∈ GridA]
AA = [a for (a, y) ∈ AY]
YY = [y for (a, y) ∈ AY]
```

F: The Bellman function

I'll first setup the problem by defining some of the necessary functions.

```julia
# Labor Income equation of motion (will use in simulations)
lnY(lnYₜ₋₁, εₜ) = (1 - ρ) * log(μ) + ρ * lnYₜ₋₁ + εₜ
# Utility function
U(C) = C^(1 - γ) / (1 - γ)
# This period's wealth
f(lnYₜ, Aₜ; α=α) = exp(lnYₜ) + Aₜ
# Budget Constraint defines Cₜ(Yₜ, Aₜ, Aₜ₊₁) = income - savings
# savings = Sₜ = (Aₜ₊₁/Z)^(1/α) from the savings technology
savings(Aₜ₊₁; α=α) = (Aₜ₊₁ / Z)^(1 / α)
c(lnYₜ, Aₜ, Aₜ₊₁; α=α) = f(lnYₜ, Aₜ; α=α) - savings(Aₜ₊₁; α=α)
# Maximum A' could be for C>0, given Y and A
Aprime(lnYₜ, Aₜ, Cₜ; α=α) = Z * (exp(lnYₜ) + Aₜ - Cₜ)^α
max_Ap(lnYₜ, Aₜ; α=α) = Aprime(lnYₜ, Aₜ, 0; α=α)
```

Now I'll define my Bellman functions – for both scalar and vector inputs:

```julia
"""
V = Bellman(b, Aₜ, lnYₜ, Aₜ₊₁; α=α)
  Evaluate the RHS of the Bellman equation

    Inputs
    b     6 x 1 coefficients in polynomial for E[ V(A',lnY') | lnY ]
    Aₜ     n-vector of current assets A
    lnYₜ   n-vector of current labor income
    Aₜ₊₁   n-vector of this period's savings (A')
    α     scalar savings technology parameter

    Output
    V     n-vector of value function
"""
function Bellman(b::AbstractArray, Aₜ::Real, lnYₜ::Real, Aₜ₊₁::Real; α=α)
    # Scalar A' and lnY, vector of coefficients b
    C = c(lnYₜ, Aₜ, Aₜ₊₁; α=α)
    u = U(C)
    V = u + β * (PolyBasis(Aₜ₊₁, lnYₜ)*b)[1]
    return V
end
function Bellman(b::AbstractArray, Aₜ::AbstractArray, lnYₜ::AbstractArray, Aₜ₊₁::AbstractArray; α=α)
    # Vector A' and lnY, vector of coefficients b
    C = c.(lnYₜ, Aₜ, Aₜ₊₁; α=α)
    u = U.(C)
    V = u .+ β * (PolyBasis(Aₜ₊₁, lnYₜ) * b)
    return V
end
```

G: The MaxBellman function

There are two ways that I will define the MaxBellman function. First I will follow McKay's Matlab Golden-search approach. Then I will define the way I would have done it in Julia given the general problem of maximizing a function using Julia's Optim.jl pacakge. Both return equivalent results, so I will just show the output of the Julia Optim.jl versions for the rest of the problem set.

The McKay Golden Search translation

First, define a vector-updating function based on the indicator vector:

```julia
"""Update elements of A with elments from B if the index I is 1 for that element"""
function update_A_with_B!(A, B, I)
    for i ∈ eachindex(I)
        if I[i] == 1
            A[i] = B[i]
        end
    end
end
```

Then the translation of McKay's MaxBellman function is

```julia
"""
[V, Ap] = MaxBellman(b; α=α)
    Maximizes the RHS of the Bellman equation using golden section search

    Inputs
    b         6-vector coefficients in polynomial for E[ V(K',Z') | Z ]

    Globals
    GridlnY     initial vector of lnY
    GridPlnY    transition matrix of lnY
    GridA       initial vector of A
    AY          cartisian product of AxlnY
    AA          decomposed A vector from AY
    YY        decomposed lnY vector from AY
"""
function MaxBellman(b; α=α)
    p = (sqrt(5) - 1) / 2

    A = first(GridA) .* ones(size(AA))
    # f(lnYₜ, Aₜ; α=α) -> f.(lnYY, AA; α=α)
    D = min.(max_Ap.(YY, AA; α=α) .- 1e-3, last(GridA))
    # -1e-3 so we always have positve consumption.
    B = p * A .+ (1 - p) * D
    C = (1 - p) * A .+ p * D

    fB = Bellman(b, AA, YY, B)
    fC = Bellman(b, AA, YY, C)

    MAXIT = 1000
    for it_inner = 1:MAXIT
        # Stop loop if we have converged
        if all(D - A .< 1e-6)
            break
        end

        I = fB .> fC

        # for indicies where fB > fC
        update_A_with_B!(D, C, I)
        update_A_with_B!(C, B, I)
        update_A_with_B!(fC, fB, I)
        update_A_with_B!(B, p * C .+ (1 - p) * A, I)
        update_A_with_B!(fB, Bellman(b, AA, YY, B), I)

        # for indicies where fB <= fC
        update_A_with_B!(A, B, .~I)
        update_A_with_B!(B, C, .~I)
        update_A_with_B!(fB, fC, .~I)
        update_A_with_B!(C, p * B .+ (1 - p) * D, .~I)
        update_A_with_B!(fC, Bellman(b, AA, YY, C), .~I)
    end

    # At this stage, A, B, C, and D are all within a small epsilon of one
    # another.  We will use the average of B and C as the optimal level of
    # savings.
    Aₜ₊₁ = (B .+ C) ./ 2

    # Make sure that the choices of next period's assets are bounded to
    # be within the upper and lower bounds of the grid.
    Aₜ₊₁ = max.(Aₜ₊₁, first(GridA))  # lower bound
    Aₜ₊₁ = min.(Aₜ₊₁, last(GridA))   # upper bound

    # evaluate the Bellman equation at the optimal savings policy to find the new
    # value function.
    V = Bellman(b, AA, YY, Aₜ₊₁)
    return V, Aₜ₊₁
end
```

The Optim.jl maximization

The most straightforward way to use Optim.jl's optimization algorithms is to first define the scalar maximization of the Bellman equation at one point in the A-Y grid:

```julia
"""Maximize the Bellman function using Aₜ₊₁ given b, Aₜ, lnY scalars"""
function MyMaxSingleBellman(b, Aₜ, lnYₜ; lbA=first(AA), ubA=last(AA), α=α)
    # Create a univariate function to maximize using Optim's maximize()
    to_maximize(Ap) = Bellman(b, Aₜ, lnYₜ, Ap; α=α)
    # Want there to be >0 consumption, so put upper bound
    # at maximum A' that results in C>0, given lnYₜ, Aₜ
    upperA = min(ubA, max_Ap(lnYₜ, Aₜ; α=α) - 1e-3)
    # Find the maximizing A', and maximal value
    out = maximize(to_maximize, lbA, upperA)
    maxBell = maximum(out)
    maxA = maximizer(out)
    return maxA, maxBell
end
```

This returns the maximizing A' and the maximal value of the Bellman equation, at this point in the grid. Then, using Julia's dot-notation for broadcasting a function over vectors, we can find the maximizing A' and maximal value at all points in the grid:

```julia
"""Maximize the Bellman function using Aₜ₊₁ given b, Aₜ, lnY vectors"""
function MyMaxBellman(b; α=α)
    # Define the function taking scalar Aₜ, lnY
    MaxBellmanVector(Aₜ, lnYₜ) = MyMaxSingleBellman(b, Aₜ, lnYₜ; α=α)
    # Broadcast over this function
    out = MaxBellmanVector.(AA, YY)
    # Extract the maximizing A' and maximal values from the return vector
    maxA = [x[1] for x in out]
    maxBell = [x[2] for x in out]
    return maxBell, maxA
end
```

H: The value function iteration for-loop

Write the value function iteration for-loop for this problem. Once the iterations converge, plot the value function, policy function, and the consumption on the meshgrid of (A, Y) using the surf function in Matlab.

The McKay value function iteration translation

```
function value_function_iteration()
    # initial guess of the coefficients of the polynomial approx to the value function (zero function)
    b = b0 = zeros(6, 1)

    Aₜ₊₁0 = zeros(size(AA))
    V0 = zeros(size(AA))
    V, Aₜ₊₁ = V0, Aₜ₊₁0
    MAXIT = 2000
    ifinal = 0
    for it = 1:MAXIT

        V, Aₜ₊₁ = MaxBellman(b; α=α)

        # take the expectation of the value function from the perspective of the previous Z
        # Need to reshape V into a 20 by 7 array where the rows correspond different levels
        # of assets and the columns correspond to different levels of income.
        # need to take the dot product of each row of the array with the appropriate column
        # of the Markov chain transition matrix
        EV = reshape(V, nA, nY) * GridPlnY

        # update our polynomial coefficients
        b = PolyGetCoef(AA, YY, EV[:])

        # see how much our policy rule, value function, coefficients have changed
        Atest = maximum(abs.(Aₜ₊₁0 .- Aₜ₊₁))
        Vtest = maximum(abs.(V0 .- V))
        btest = maximum(abs.(b0 .- b))
        b0 = b; V0 = V; Aₜ₊₁0 = Aₜ₊₁; ifinal = it

        # Break when we have converged on all three criteria
        if max(Vtest, Atest, btest) < 1e-5
            break
        end
    end

    return Dict(:Ap => Aₜ₊₁, :V => V, :b => b, :i => ifinal)
end

@time out = value_function_iteration();  # 0.707526 seconds, 106 iterations
```

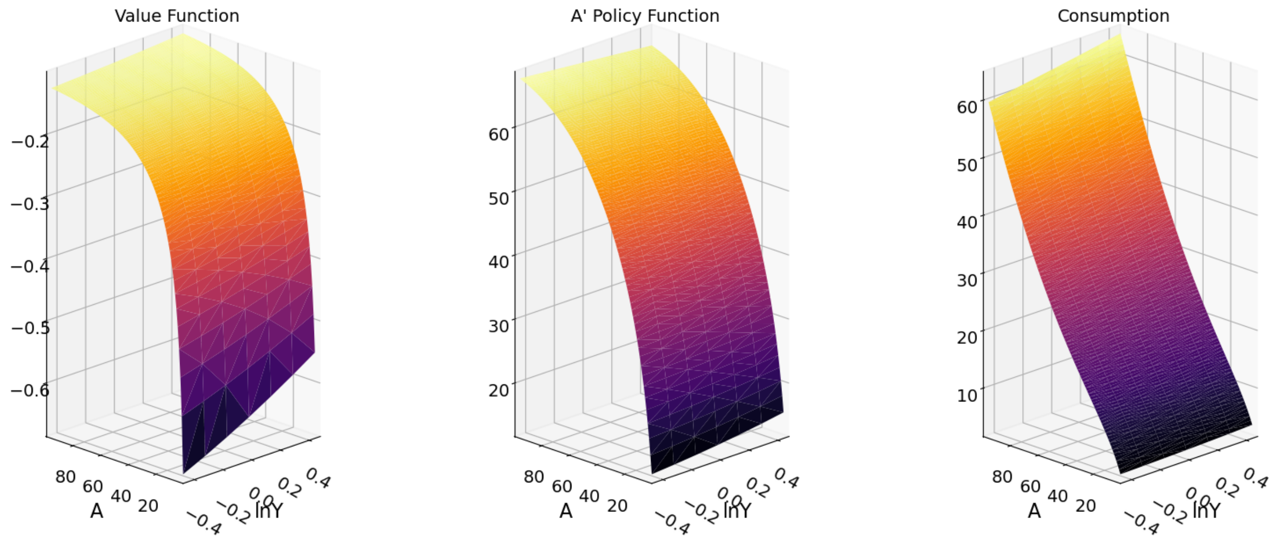
Iterating over the Optim.jl maximization

```julia
"""Iterate over the polynomial coefficients to converge on the value function"""
function MyBellmanIteration()
    # initial guess of the coefficients of the polynomial approx to the value function (zero function)
    b = zeros(6)
    Aₜ₊₁0 = zeros(size(AA))
    MAXIT = 2000
    # Keep all the iterations for inspection after convergence
    Vlist, Alist, blist = [zeros(size(AA))], [zeros(size(AA))], [b]
    for it = 1:MAXIT
        # println("b = $b")
        V, Aₜ₊₁ = MyMaxBellman(b; α=α)
        append!(Vlist, [V])
        append!(Alist, [Aₜ₊₁])

        # take the expectation of the value function from the perspective of the previous A
        # Need to reshape V into a 100x7 array where the rows correspond different levels
        # of assets and the columns correspond to different levels of income.
        # need to take the dot product of each row of the array with the appropriate column
        # of the Markov chain transition matrix
        EV = reshape(V, nA, nY) * GridPlnY

        # update our polynomial coefficients
        b = PolyGetCoef(AA, YY, EV[:])
        append!(blist, [b])

        # see how much our policy rule has changed
        Atest = maximum(abs.(Aₜ₊₁0 .- Aₜ₊₁))
        Vtest = maximum(abs.(Vlist[end] - Vlist[end-1]))
        btest = maximum(abs.(blist[end] - blist[end-1]))
        if max(Atest, Vtest, btest) < 1e-5
            break
        end

        Aₜ₊₁0 = copy(Aₜ₊₁)
    end
    return Alist, Vlist, blist
end
```

Then we can run the value function iteration and plot the results:

```julia
@time Alist, Vlist, blist = MyBellmanIteration();
println("MyBellmanIteration Finished in $(length(blist)-1) iterations");
Aₜ₊₁_, V_, b_ = last(Alist), last(Vlist), last(blist)
plotargs = (camera=(-45, 20), xlabel="lnY", ylabel="A",
            legend=:none, aspect_ratio=[1,1,2])
# Value Function
pH1 = surface(YY, AA, V_, title="Value Function"; plotargs...)
# Policy Function
pH2 = surface(YY, AA, Aₜ₊₁_, title="A' Policy Function"; plotargs...)
# Consumption
CC = c.(YY, AA, Aₜ₊₁_)
pH3 = surface(YY, AA, CC, title="Consumption"; plotargs...)
# Put them all together
pH4 = plot(pH1, pH2, pH3, layout=(1,3), size=(1600, 800),
            tickfontsize=14, labelfontsize=16,
            xtickfontrotation = -30)
savefig(pH4, "H-all")
```

Resulting in these plots:



I: Howard acceleration

I implemented Howard acceleration using the Julia Optim version of the functions.

```julia
"""Iterate faster over the polynomial coefficients to converge on the value function using Howard
    acceleration."""
function MyFasterBellmanIteration(; inner_mod = 500)
    # initial guess of the coefficients of the polynomial approx to the value function (zero function)
    b = zeros(6)
    MAXIT = 20000; inner_it = 0
    Vlist, Alist, blist = [zeros(size(AA))], [zeros(size(AA))], [b]
    for it = 1:MAXIT
        # Every inner_mod iterations, get the maximizing A'
        if it % inner_mod == 1
            V, Aₜ₊₁ = MyMaxBellman(b; α=α)
            append!(Vlist, [V])
            append!(Alist, [Aₜ₊₁])
            inner_it += 1
        else
            V = Bellman(b, AA, YY, Alist[end])
        end

        EV = reshape(V, nA, nY) * GridPlnY
        b = PolyGetCoef(AA, YY, EV[:])
        append!(blist, [b])

        # see how much our policy rule has changed
        Atest = maximum(abs.(Alist[end] - Alist[end-1]))
        Vtest = maximum(abs.(Vlist[end] - Vlist[end-1]))
        btest = maximum(abs.(blist[end] - blist[end-1]))
        if max(Atest, Vtest, btest) < 1e-5
            println("Converged in $it iterations, $inner_it maximization iterations")
            break
        end
    end
    return Alist, Vlist, blist
end
```

Using Julia's timing functions, I tested how fast I could get the

```julia
# warm up function (precompile)
MyFasterBellmanIteration(inner_mod=100);

# Find which modulo for the bellman A' maximization results in shortest time
mods = 10:10:1000
f(x) = @elapsed MyFasterBellmanIteration(inner_mod=x);
times = f.(mods)
mintime0, minidx = findmin(times)
minmod = mods[minidx]

# Compare to
mintime1 = @elapsed MyBellmanIteration()
multiplier = round(mintime1 / mintime0, digits=2)
println("Howard acceleration with mod $minmod resulted in $multiplier times faster convergence")
```

Howard acceleration with mod 80 resulted in 10.39 times faster convergence compared to the unaccelerated Optim iterations.

J: The Simulate function

First I create a struct to hold the values I want to keep.

```julia
struct SimReturn
    Ap
    A
    Y
    C
end
# Create an instance like this: SimReturn(Ap2, A, Y, C)
```

And then write the function to simulate.

```julia
"""
Sim = Simulate(bKp, Mode, T)
    Simulates the model.
    Inputs:
    bKp       Polynomial coefficients for polynomial for Kp policy rule
    Mode      Mode = 'random' -> draw shocks
              Mode = 'irf'    -> impulse response function
    T         # of periods to simulate
"""
function Simulate(bAp, Mode, T; α=α)
    A = zeros(T); Y = zeros(T)
    A[1] = Aₛₛ

    if Mode == "irf"
        Y[1] = σ
        ε = zeros(T)
    elseif Mode == "random"
        Y[1] = 0
        ε = σ * randn(T)
    else
        throw("Unrecognized Mode $Mode in Simulate()");
    end

    Ap1 = PolyBasis(A[1:(T-1)], Y[1:(T-1)]) * bAp

    for t ∈ 2:T
        Aptemp = (PolyBasis(A[t-1], Y[t-1]) * bAp)[1]
        Apmax = max_Ap(Y[t-1], A[t-1])
        A[t] = (PolyBasis(A[t-1], Y[t-1]) * bAp)[1]
        Y[t] = lnY(Y[t-1], ε[t])
    end

    # Compute quantities from state variables
    Ti = 2:(T-1)
    Ap2 = A[Ti .+ 1]
    A = A[Ti]
    Y = Y[Ti]
    C = c.(Y, A, Ap2; α=α)

    return SimReturn(Ap2, A, Y, C)
end
```

K: The 10,000 period simulation

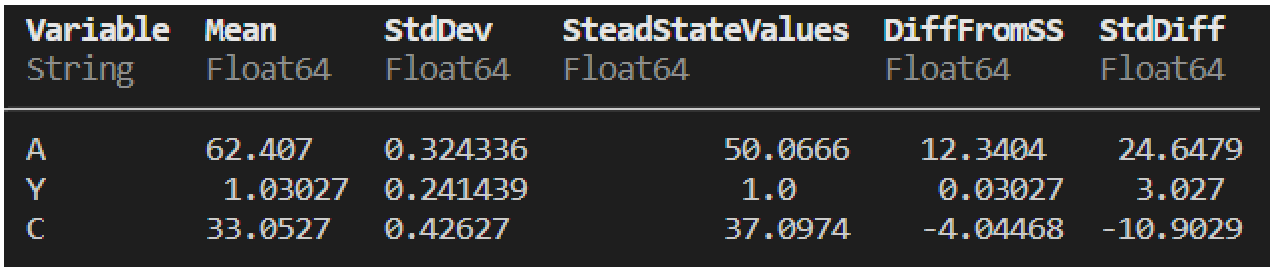
Using your new Simulate function, produce a 10,000 period simulation of the evolution of A, Y, and C. Report a histogram of A, Y , and C. (Note: Yt not log Yt .) Report the mean and standard deviation of each variable. Plot the evolution of A, Y , and C over a 100 period stretch starting from period 1000. How do these mean values compare to the steady-state values calculated earlier?

First simulate:

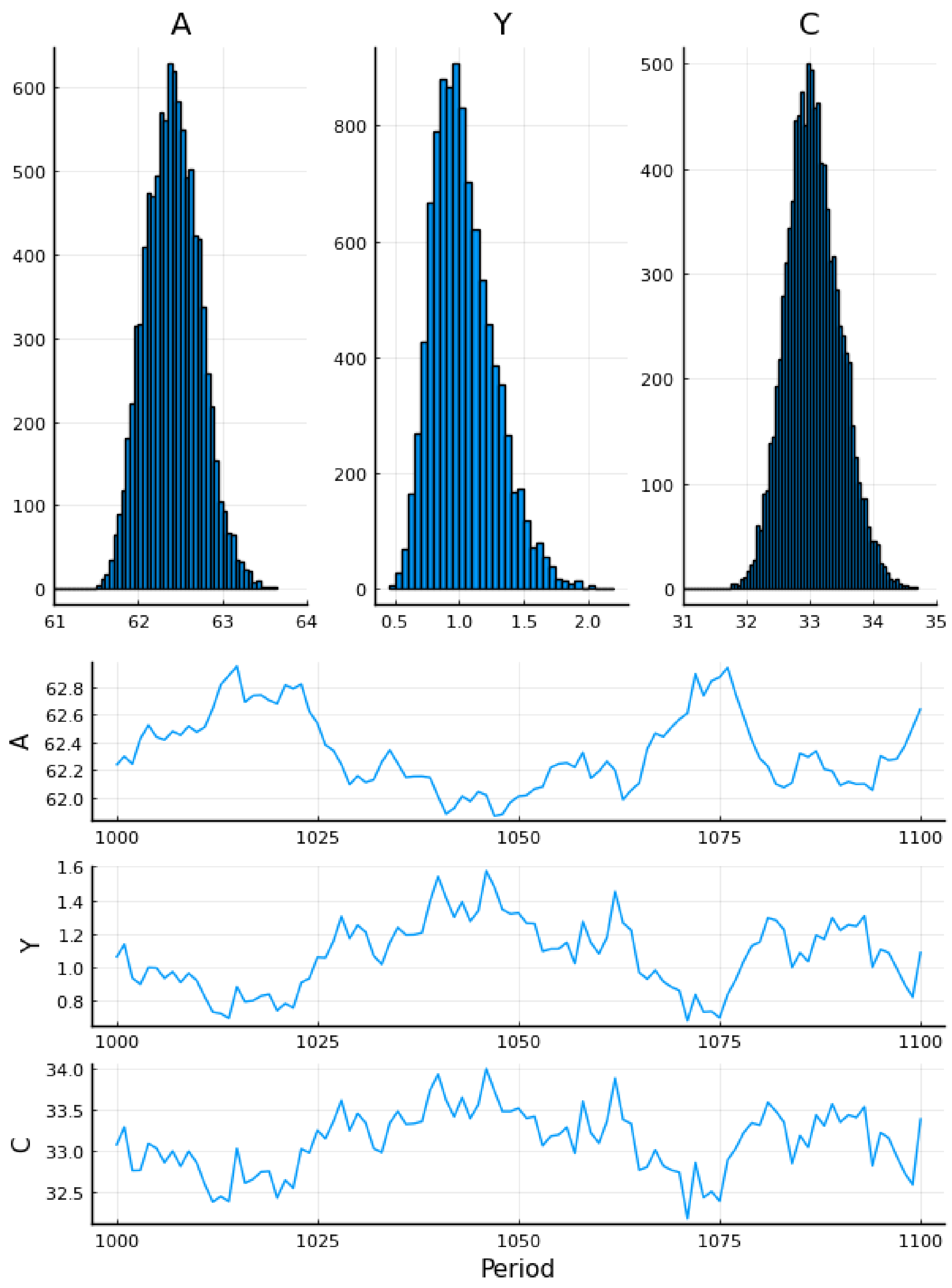
```
bAp2 = PolyGetCoef(AA, YY, Ap2)
nPeriods = 10000
@time SimK = Simulate(bAp2, "random", nPeriods);
```

Then create the outputs:

```
# Report a histogram of A, Y , and C. (Note: Yt not log Yt .)
hk1 = histogram(SimK.A, title="A", label="", xlims=(61,64))
hk2 = histogram(e.^SimK.Y, title="Y", label="")
hk3 = histogram(SimK.C, title="C", label="", xlims=(31,35))
hk4 = plot(hk1, hk2, hk3, layout=(1,3))

# Report the mean and standard deviation of each variable.
varsk = ["A", "Y", "C"]
meansk = mean.([SimK.A, e.^SimK.Y, SimK.C])
sdk = std.([SimK.A, e.^SimK.Y, SimK.C])

# Plot the evolution of A, Y , and C over a 100 period stretch starting from period 1000
periods = 1000:1100
pk1 = plot(periods, SimK.A[periods], ylabel="A", label="")
pk2 = plot(periods, (e.^SimK.Y)[periods], ylabel="Y", label="")
pk3 = plot(periods, SimK.C[periods], ylabel="C", xlabel="Period", label="")
pk4 = plot(pk1, pk2, pk3, layout=(3,1))

# How do these mean values compare to the steady-state values calculated earlier?
# A comparison
Adiffk = meansk[1] - Aₛₛ
# Y comparison
Ydiffk = meansk[2] - μ
# C comparison
Cdiffk = meansk[3] - c(μ, Aₛₛ, Aₛₛ)
diffs = [Adiffk, Ydiffk, Cdiffk]
oldmeans = [Aₛₛ, μ, c(μ, Aₛₛ, Aₛₛ)]
stddiffs = [Adiffk/Aₛₛ*100, Ydiffk/μ*100, Cdiffk/c(μ, Aₛₛ, Aₛₛ)*100]
dfk = DataFrame(Variable = varsk,
                Mean = meansk,
                StdDev = sdk,
                SteadStateValues = oldmeans,
                DiffFromSS = diffs,
                StdDiff = stddiffs)
```



```
Variable  Mean      StdDev    SteadStateValues  DiffFromSS  StdDiff
String    Float64   Float64   Float64           Float64     Float64

A         62.407    0.324336           50.0666     12.3404   24.6479
Y          1.03027  0.241439            1.0         0.03027   3.027
C         33.0527   0.42627            37.0974     -4.04468 -10.9029
```



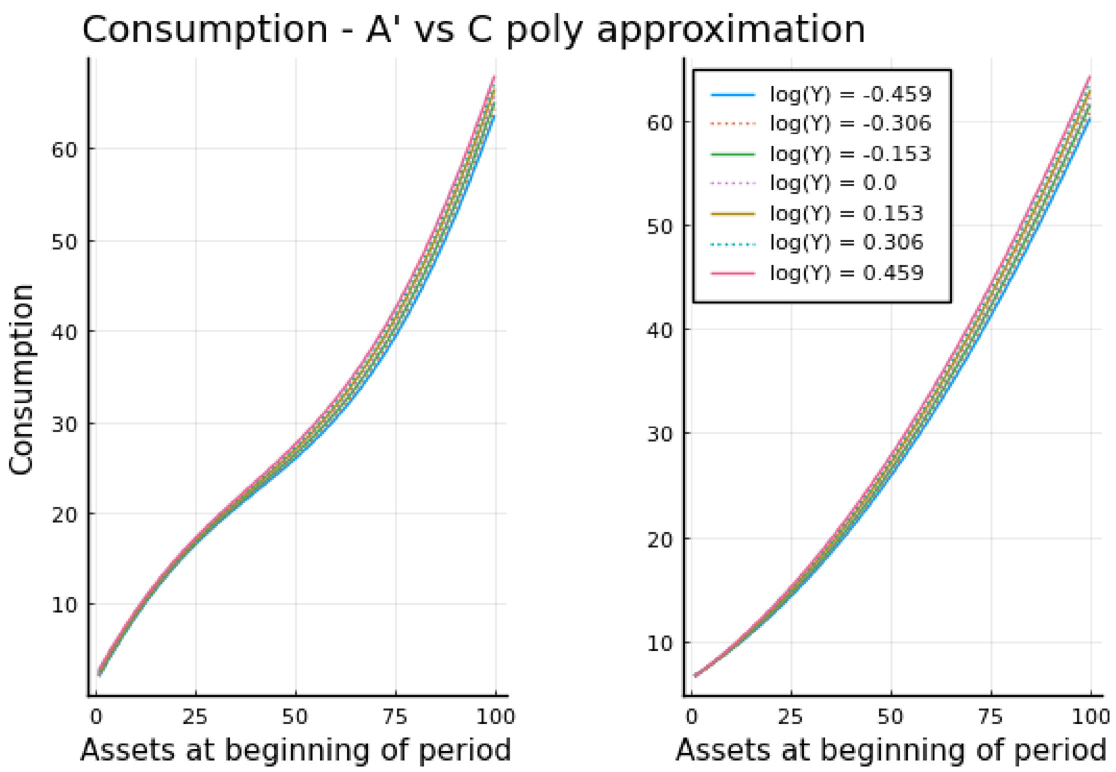
The mean of income (Y) is pretty close to the given mean of 1 in the problem setup, only about 2.4% higher. The mean of assets (A) in the simulation is somewhat larger than the steady state value – about 12 units above the steady state value, which is about 24.7% higher. The mean of consumption (C) in the simulation is a bit lower than the consumption implied by the mean income and the steady state asset level – the mean from the simulation is about 11% below the steady state level.

L: Plot consumption

Plot consumption as a function of A for several values of Y . Do this for the entire range of A on your grid.

There are two methods of computing the consumption: the way McKay does it is to use the polynomial approximation of A' (his Kp), and use the formula for consumption and the approximate policy function to estimate optimized consumption. I believe this is the correct way to go. This is represented by the left graphs in the next three answers. The other way would be to approximate C by creating a new polynomial approximation of C and using that to calcuate the following values. That is represented by the right graphs. I believe these are overly smooth due to approximation error from the polynomial approximation.

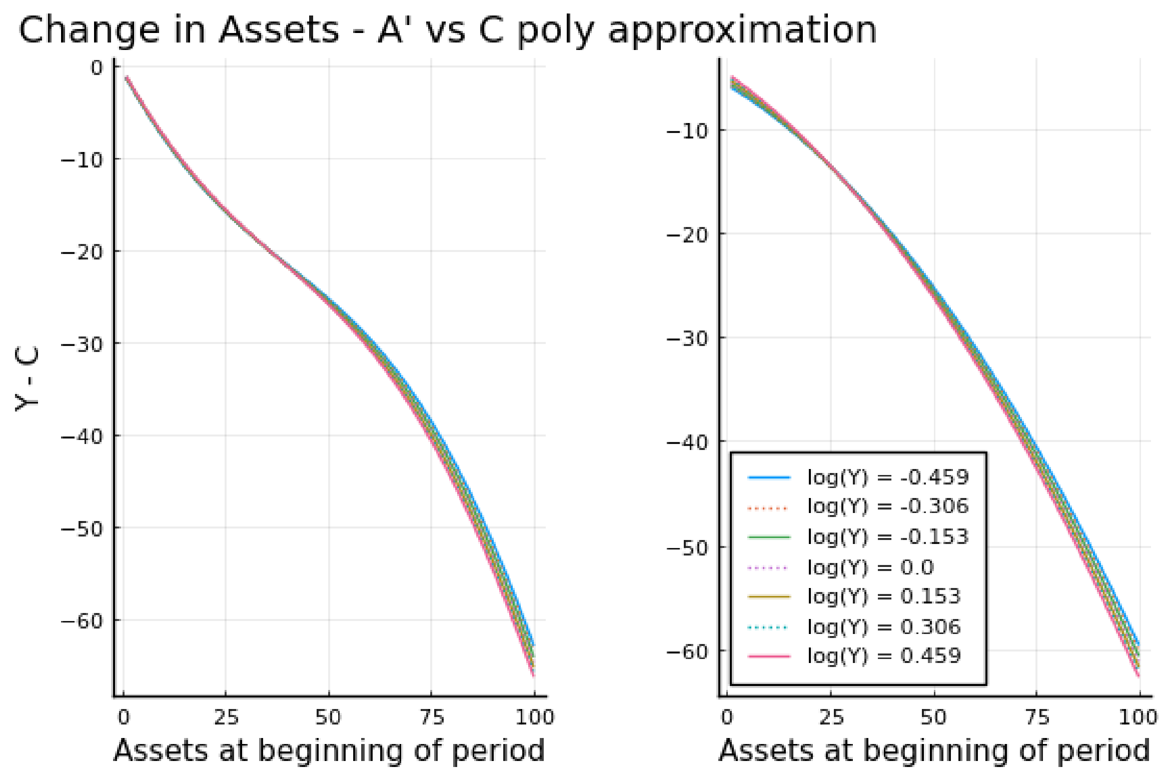
```julia

# PLOT USING A' APPROXIMATION
bAp2 = PolyGetCoef(AA, YY, Ap2)
cL(A, lnY) = c(lnY, A, (PolyBasis(A, lnY) * bAp2)[1])
labelsL = "log(Y) = " .* string.(round.(GridlnY', digits=3))
linestylesL = [:solid :dot :solid :dot :solid :dot :solid]
pL1 = plot(reshape(cL.(AA, YY), nA, nY),
     label=labelsL,
     linestyle=linestylesL,
     legend=:topleft,
     title="Consumption based on asset policy approximation",
     xlabel="Assets at beginning of period",
     ylabel="Consumption")

# PLOT USING NEW C APPROXIMATION (and underlying A' approximation in cL())
bC = PolyGetCoef(AA, YY, cL.(AA, YY))
CL = PolyBasis(AA, YY)*bC
pL2 = plot(reshape(CL, nA, nY),
     label=labelsL,
     linestyle=linestylesL,
     legend=:topleft,
     title="Consumption based on consumption approximation",
     xlabel="Assets at beginning of period",
     ylabel="Consumption")


pL3 = plot(pL1, pL2, layout=(1,2))
savefig(pL3, "L-consumption")
```


Consumption - A' vs C poly approximation
Consumption
Assets at beginning of period
Assets at beginning of period
log(Y) = -0.459
log(Y) = -0.306
log(Y) = -0.153
log(Y) = 0.0
log(Y) = 0.153
log(Y) = 0.306
log(Y) = 0.459
0
25
50
75
100
10
20
30
40
50
60

M: Plot change in assets

Plot change in assets Y − C as a function A for several values of Y . Do this for the entire range of A on your grid.

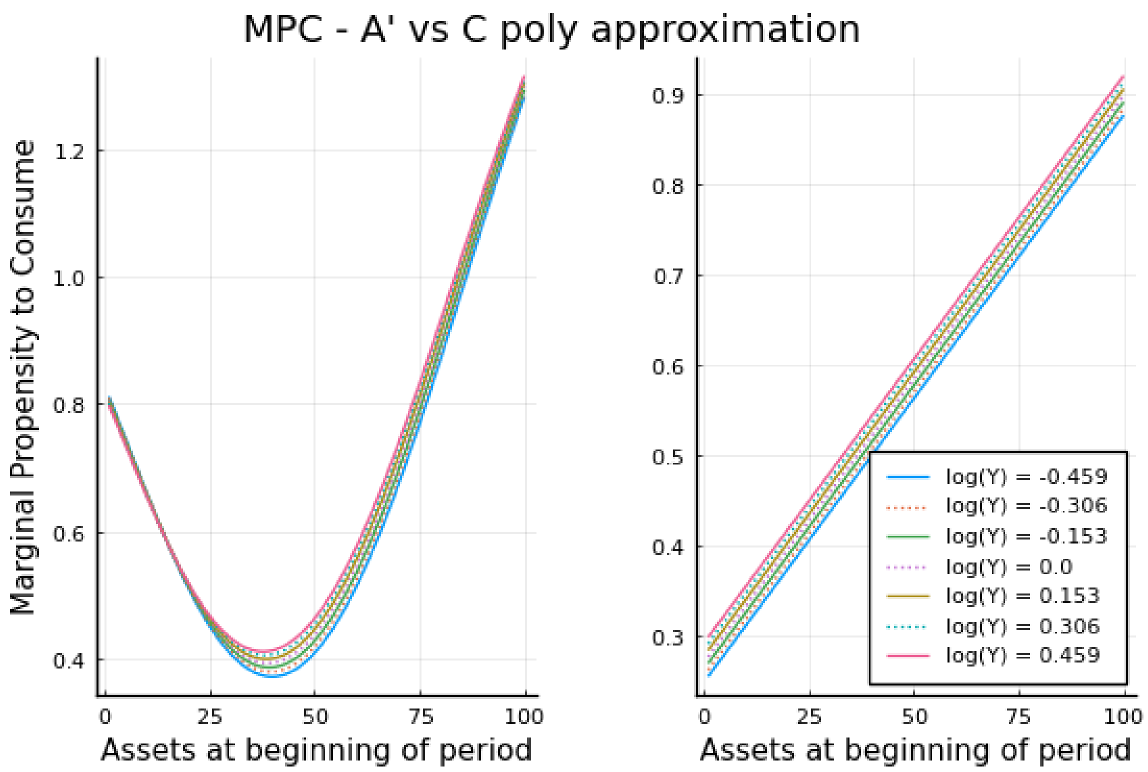
```julia
# PLOT USING A' APPROXIMATION
pM1 = plot(reshape(ℯ.^YY .- cL.(AA, YY), nA, nY),
     label=labelsL,
     linestyle=linestylesL,
     legend=:bottomleft,
     title="Change in Assets based on asset policy approximation",
     xlabel="Assets at beginning of period",
     ylabel="Y - C")


# PLOT USING NEW C APPROXIMATION (and underlying A' approximation in cL())
pM2 = plot(reshape(ℯ.^YY .- CL, nA, nY),
     label=labelsL,
     linestyle=linestylesL,
     legend=:bottomleft,
     title="Change in Assets based on consumption approximation",
     xlabel="Assets at beginning of period",
     ylabel="Y - C")


pM3 = plot(pM1, pM2, layout=(1,2))
savefig(pM3, "M-changeinassets")
```



N: Plot the marginal propensity to consume

Plot the marginal propensity to consume as a function of A for several values of Y . Do this for the entire range of A on your grid. You can approximate the marginal propensity to consume as the extra consumption in the period that results from a windfall gain of 1 unit of A. Does this plot make economic sense? (Hint: It might not due to the limitations of the polynomial approximation methods we are using in this problem set.)

```julia
# PLOT USING A' APPROXIMATION (Swooshy MPC)
MPC1 = cL.(AA .+ 1, YY) .- cL.(AA, YY)
pN1 = plot(reshape(MPC1, nA, nY),
     label=labelsL,
     linestyle=linestylesL,
     legend=:none,
    #  title="Marginal Propensity to Consume based on optimal asset policy",
     xlabel="Assets at beginning of period",
     ylabel="Marginal Propensity to Consume")

# PLOT USING NEW C APPROXIMATION (Straight line MPC)
CN = PolyBasis(AA .+ 1, YY) * bC
MPC2 = PolyBasis(AA .+ 1, YY) * bC .- PolyBasis(AA, YY) * bC
pN2 = plot(reshape(MPC2, nA, nY),
     label=labelsL,
     linestyle=linestylesL,
     legend=:bottomright,
    #  title="Marginal Propensity to Consume",
     xlabel="Assets at beginning of period",
    #  ylabel="MPC"
     )

pN3 = plot(pN1, pN2, layout=(1,2),
            title=[" "^40*"MPC - A' vs C poly approximation" ""])
savefig(pN3, "N-mpc")
```


MPC - A' vs C poly approximation
Marginal Propensity to Consume
Assets at beginning of period
Assets at beginning of period
log(Y) = -0.459
log(Y) = -0.306
log(Y) = -0.153
log(Y) = 0.0
log(Y) = 0.153
log(Y) = 0.306
log(Y) = 0.459

O: Explore α

(Optional) Explore how the solution method runs into trouble if you try to increase towards 1. As you do this, you may want to vary the range of assets on the grid and also the polynomial basis. If you consider higher order polynomials than 2nd order, it may be interesting for you to plot the value function for a particular value of Yt as a function of At during intermediate steps in the value function iteration. You may start seeing cases where the value function becomes slightly non-monotonic. You can think about how this will lead the golden search algorithm to run into problems. (This is the problem that we couldn't get around easily in writing the problem.)

I did not complete this comparison.

Appendix: Packages

```julia
import Pkg
Pkg.activate(pwd())
try
    using Optim, Parameters, Plots, Revise, DataFrames, Pluto
    pyplot()
catch e
    Pkg.add(["Plots", "PyPlot", "Optim", "Parameters", "Revise", "DataFrames", "Pluto"])
    using Optim, Parameters, Plots, Revise, DataFrames
    pyplot()
end
includet("Tauchen.jl")  # Installs Distributions if not installed
```

Appendix: Tauchen

```julia
try
    using Distributions
catch e
    import Pkg; Pkg.add(["Distributions"])
    using Distributions
end
cdf_normal(x) = cdf(Normal(),x)

"""
Function tauchen(N,mu,rho,sigma,m)

    Purpose:    Finds a Markov chain whose sample paths
                approximate those of the AR(1) process
                    z(t+1) = (1-rho)*mu + rho * z(t) + eps(t+1)
                where eps are normal with stddev sigma

    Format:     {Z, Zprob} = Tauchen(N,mu,rho,sigma,m)

    Input:      N       scalar, number of nodes for Z
                mu      scalar, unconditional mean of process
                rho     scalar
                sigma   scalar, std. dev. of epsilons
                m       max +- std. devs.

    Output:     Z       N*1 vector, nodes for Z
                Zprob   N*N matrix, transition probabilities

        Martin Floden
        Fall 1996

        This procedure is an implementation of George Tauchen's algorithm
        described in Ec. Letters 20 (1986) 177-181.
"""
function tauchen(N,mu,rho,sigma,m)
    Zprob = zeros(N,N);
    a     = (1-rho)*mu;

    ZN = m * sqrt(sigma^2 / (1 - rho^2))
    Z = range(-ZN + mu, ZN + mu, N)
    zstep = Z[2]-Z[1]

    for j ∈ 1:N, k ∈ 1:N
        if k == 1
            Zprob[j,k] = cdf_normal((Z[1] - a - rho * Z[j] + zstep / 2) / sigma)
        elseif k == N
            Zprob[j,k] = 1 - cdf_normal((Z[N] - a - rho * Z[j] - zstep / 2) / sigma)
        else
            Zprob[j,k] = cdf_normal((Z[k] - a - rho * Z[j] + zstep / 2) / sigma) -
                         cdf_normal((Z[k] - a - rho * Z[j] - zstep / 2) / sigma);
        end
    end

    return Z, Zprob
end
```

Appendix: References

```
References:
https://alisdairmckay.com/Notes/NumericalCrashCourse/FuncApprox.html
https://alisdairmckay.com/Notes/NumericalCrashCourse/index.html

Expectations:
https://quantecon.github.io/Expectations.jl/dev/
```