A: the household's problem

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

We wish to maximize the sum of the utility and the discounted value function, subject to the budget constraint, given the state variable A and the input variables Y and X (employment status). The choice variables are consumption this period C and assets at the beginning of next period A'. So the value function is

$$V(A, Y, X) = \max_{C, A'} \Big[U(C) + \beta \mathbb{E}_{Y', X'} [V(A', Y', X') \mid Y, X] \Big]$$

The budget constraint is

$$C + \frac{A'}{1+r} = Y + A$$

Where income Y is either employment income or b (unemployment income) and Y' is subject to unemployment uncertainty. If $X = 1$ is employed and $X = 0$ is unemployed, the uncertainty in next periods' employment X' is characterized by the following probabilities.

$$\begin{aligned}
&\mathbb{P}(X' = 0 | X = 1) = p \\
&\mathbb{P}(X' = 1 | X = 1) = 1 - p \\
&\mathbb{P}(X' = 0 | X = 0) = 1 - q \\
&\mathbb{P}(X' = 1 | X = 0) = q
\end{aligned}$$

We then can separate the value function into two functions based on the current value of X – if we are currently employed (V_e) and if we are currently unemployed (V_u). We can then resolve the X part of the conditional expectation in the value functions to

$$\begin{aligned}
V_e(A, Y) &= \max_{C, A'} \Big[U(C) + \beta \mathbb{E}_{Y'} [(1-p) V_e(A', Y') + p V_u(A', b) \mid Y] \Big] \\
V_u(A, Y) &= \max_{C, A'} \Big[U(C) + \beta \mathbb{E}_{Y'} [q V_e(A', Y') + (1-q) V_u(A', b) \mid Y] \Big]
\end{aligned}$$

We know that the the equation of motion for Y' is based on the current Y:

$$\ln(Y'(Y)) = (1-\rho)\mu + \rho \ln(Y) + \varepsilon$$

Denoting the implicit random function Y' of Y as $Y'_\varepsilon(Y)$, let's simplify the value functions a bit more:

$$\begin{aligned}
V_e(A, Y) &= \max_{C, A'} \Big[U(C) + \beta \mathbb{E}_{\varepsilon} [(1-p) V_e(A', Y'_\varepsilon(Y)) + p V_u(A', b)] \Big] \\
V_u(A, Y) &= \max_{C, A'} \Big[U(C) + \beta \mathbb{E}_{\varepsilon} [q V_e(A', Y'_\varepsilon(Y)) + (1-q) V_u(A', b)] \Big]
\end{aligned}$$

Noting that the expectation is now over the normal random variable ε in the equation of motion for Y'.

Plugging in the budget constraint to get C in terms of known variables A and Y, and the choice variable A', we can eliminate C as a choice variable:

$$V_e(A,Y) = \max_{A'} \left[U\left(Y + A - \frac{A'}{1+r}\right) + \beta \mathbb{E}_\varepsilon[(1-p)V_e(A', Y'_\varepsilon(Y)) + pV_u(A', b)] \right]$$
$$V_u(A,Y) = \max_{A'} \left[U\left(Y + A - \frac{A'}{1+r}\right) + \beta \mathbb{E}_\varepsilon[qV_e(A', Y'_\varepsilon(Y)) + (1-q)V_u(A', b)] \right]$$

Where the utility function is

$$U(C) = \frac{C^{1-\gamma}}{1-\gamma}$$

And two more constraints must hold for consumption and assets which bound the possible maximizing values A' that we can utilize:

$$C \geq 0$$
$$A' \geq 0$$

B, C, D: Programming Setup

> There will be no version of the PolyBasis and PolyGetCoeff functions used in this problem set. Define the parameter values.

```julia
# Parameters Given (tell the compiler that this are static)
@consts begin
    γ = 2
    β = 0.98
    μ = 1
    bb = 0.4
    ρ = 0.9
    σ² = 0.05
    σ = σ²^(1/2)
    p = 0.05
    q = 0.25
    r = 0.01
end
```

E: Create the Grid

> Create a grid on A and $\tilde{Y}$. Feel free to use McKay's tauchen function as needed. Use 7 grid points for $\tilde{Y}$ and 1,000 grid points for A. (Please create the grid for $\tilde{Y}$, not log $\tilde{Y}$. Uniformity will make grading easier.) Choose reasonable values for the size of the grid (i.e., min and max points for each dimension). Note that the lower bound of the A grid in this case should be zero, since the borrowing constraint prevents A from being negative.

I translated the tauchen function to Julia (see Appendix). Below is the code to create the grid vectors.

```julia
# Create a grid for lnY
NY = 7;  # number of points in our grid for lnY
NstdY = 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, GridPY = tauchen(NY, log(μ), ρ, σ, NstdY)
GridY = e.^GridlnY
# tauchen() imported from Tauchen.jl

GridPY = GridPY'  # 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.

# Create a grid from 0 to upper bound
GridA_upper = 200
GridA_lower = 0
NA = 1_000  # number of points in our grid for A
GridA = range(GridA_lower, GridA_upper, 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]
```

Note that I have defined the grid in terms of $\log(Y)$ because the tauchen function and the equation of motion make it much easier to define a regular grid in $\log(Y)$. For all the following functions, I utilize $\log(Y)$, but then plot the results in linear Y simply by exponentiating the $\log(Y)$ grid.

F: The Bellman function

> Write a Bellman matlab function for this problem. Since we have two value functions (V_e and V_u), the best way to go here is to write two functions BellmanE and BellmanU, one for each value function. The main difference with the bellman function from PS8 is that you will not be passing the polynomial coefficients to the function. Instead, you will have arguments that you may want to call EVe and EVu.

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

```
# Utility function
U(C) = C^(1 - γ) / (1 - γ)

# This period's starting wealth
f(lnY_t, A_t) = exp(lnY_t) + A_t
# Savings this period based on next periods' assets
savings(A_{t+1}) = A_{t+1}/(1+r)

# Budget Constraint defines C_t(Y_t, A_t, A_{t+1}) = income - savings
c(lnY_t, A_t, A_{t+1}) = f(lnY_t, A_t) - savings(A_{t+1})
# Maximum A' could be for C>0, given Y and A
Aprime(lnY_t, A_t, C_t) = (1+r) * (exp(lnY_t) + A_t - C_t)
max_Ap(lnY_t, A_t) = Aprime(lnY_t, A_t, 0)
```

Now I'll define my Bellman functions – for both vector and interpolated function inputs. I later use the interpolated function versions because it is expensive to regenerate the interpolated functions. I can do this once, then pass it to the bellman functions for evaluations instead.

```
"""Vector Bellman function when Employed this period"""
function BellmanE(EVe::AbstractArray, EVu::AbstractArray, A_t::AbstractArray, lnY_t::AbstractArray, A_{t+1}::
    AbstractArray)
    # EMPLOYED: vector A', lnY, EVe, EVu
    C = c.(lnY_t, A_t, A_{t+1})
    # Interpolate EVe and EVu at A_{t+1}, lnY_t
    EVe2 = interpolate_EV(EVe, lnY_t, A_{t+1})
    EVu2 = interpolate_EV(EVu, lnY_t, A_{t+1})
    # P(emp | emp) = 1-p; P(unemp | emp) = P
    Ve = U.(C) .+ β*( (1-p)*EVe2 + p*EVu2 )
    return Ve
end
"""Scalar Bellman function when Employed this period.
    EVe, EVu are interpolation functions of (A_{t+1}, lnY_t)
"""
function BellmanE(EVe::Function, EVu::Function, A_t::Real, lnY_t::Real, A_{t+1}::Real)
    # EMPLOYED: vector A', lnY, EVe, EVu
    C = c(lnY_t, A_t, A_{t+1})
    # P(emp | emp) = 1-p; P(unemp | emp) = P
    Ve = U(C) + β*( (1-p)*EVe(A_{t+1}, lnY_t) + p*EVu(A_{t+1}, lnY_t) )
    return Ve
end


"""Vector Bellman function when Unemployed this period"""
function BellmanU(EVe::AbstractArray, EVu::AbstractArray, A_t::AbstractArray, A_{t+1}::AbstractArray)
    # UNEMPLOYED: vector A', lnY=bb, EVe, EVu
    C = c.(bb, A_t, A_{t+1})
    # Interpolate EVe and EVu at A_{t+1}, lnY_t
    lnY_t = repeat([bb], length(A_t))
    EVe2 = interpolate_EV(EVe, lnY_t, A_{t+1})
    EVu2 = interpolate_EV(EVu, lnY_t, A_{t+1})
    # P(emp | unemp) = q; P(unemp | unemp) = 1-q
    Vu = U.(C) .+ β*( q*EVe2 .+ (1-q)*EVu2 )
```

```
    return Vu
end
"""Scalar Bellman function when Unemployed this period.
    EVe, EVu are interpolation functions of (A_{t+1}, lnY_t)
"""
function BellmanU(EVe::Function, EVu::Function, A_t::Real, A_{t+1}::Real)
    # UNEMPLOYED: vector A', lnY=bb, EVe, EVu
    C = c(bb, A_t, A_{t+1})
    # P(emp | unemp) = q; P(unemp | unemp) = 1-q
    Vu = U(C) + β*( q*EVe(A_{t+1}, bb) + (1-q)*EVu(A_{t+1}, bb) )
    return Vu
end
```

G: The MaxBellman function

Optim.jl is the main optimization package in Julia. I define an Optim.jl type of maximization function. I showed in the last problem set that this method of maximization returns identical results to the manual golden search algorithm – it's possible to select the golden search algorithm within the Optim.jl package structure, but I let Optim.jl choose the default method.

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. This returns the maximizing A' and the maximal value of the Bellman equation, at this point in the grid.

```
"""Maximize the Bellman function using A_{t+1} given EVe, EVu functions and A_t, lnY scalars"""
function MyMaxSingleBellmanE(EVe, EVu, A_t, lnY_t)
    # Define a univariate function to maximize over A_{t+1}
    to_maximize(A_{t+1}) = BellmanE(EVe, EVu, A_t, lnY_t, A_{t+1})
    # Want there to be >0 consumption, so put upper bound
    # at maximum A' that results in C>0, given lnY_t, A_t
    upperA = min(GridA_upper, max_Ap(lnY_t, A_t) - 1e-3)
    # Find the maximizing A_{t+1} for this point in the A_t, lnY_t grid
    out = maximize(to_maximize, GridA_lower, upperA)
    V = maximum(out)
    A_{t+1} = maximizer(out)
    return A_{t+1}, V
end
function MyMaxSingleBellmanU(EVe, EVu, A_t)
    # Define a univariate function to maximize over A_{t+1}
    to_maximize(A_{t+1}) = BellmanU(EVe, EVu, A_t, A_{t+1})
    # Want there to be >0 consumption, so put upper bound
    # at maximum A' that results in C>0, given lnY_t=bb, A_t
    upperA = min(GridA_upper, max_Ap(bb, A_t) - 1e-3)
    # Find the maximizing A_{t+1} for A_t, lnY_t=bb
    out = maximize(to_maximize, GridA_lower, upperA)
    V = maximum(out)
    A_{t+1} = maximizer(out)
    return A_{t+1}, V
end
```

I also need to define a function that generates the interpolation functions for value functions. I will also use this later to create interpolation functions for A'.

```
function interpolate_EV(EV::AbstractArray)
    # Convert EV to matrix for interpolation
    EVmat = reshape(EV, NA, NY)
    # Create interpolation function (based on regular grids 1:NA and 1:NY)
    Interp = interpolate(EVmat, BSpline(Cubic(Line(OnGrid()))))
    # Scale the inputs to match the actual grids
    sInterp = Interpolations.scale(Interp, GridA, GridlnY)
    # Return interpolated function on scalar A_{t+1} and lnY_t
    EVinterp(A_{t+1}, lnY_t) = sInterp(A_{t+1}, lnY_t)
    return EVinterp
end
```

Using Julia's dot-notation for broadcasting a function over vectors, we can find the maximizing A' and maximal function value at all points in the grid:

```
"""Maximize the Bellman function using A_{t+1} given EVe, EVu, A_t, lnY vectors"""
function MyMaxBellmanE(EVe::AbstractArray, EVu::AbstractArray)
    # Create interpolation functions (functions of lnY_t, A_{t+1})
    EVefun = interpolate_EV(EVe)
    EVufun = interpolate_EV(EVu)
    # Define the function taking scalar A_t, lnY
```

```
    MaxBellmanVector(A_t, lnY_t) = MyMaxSingleBellmanE(EVefun, EVufun, A_t, lnY_t)
    # Broadcast this function over the grid
    out = MaxBellmanVector.(AA, YY)
    maxA = [x[1] for x in out]
    maxBell = [x[2] for x in out]
    return maxBell, maxA
end
function MyMaxBellmanU(EVe::AbstractArray, EVu::AbstractArray)
    # Create interpolation functions (functions of lnY_t, A_{t+1})
    EVefun = interpolate_EV(EVe)
    EVufun = interpolate_EV(EVu)
    # Define the function taking scalar A_t, lnY
    MaxBellmanVector(A_t) = MyMaxSingleBellmanU(EVefun, EVufun, A_t)
    # Broadcast this function over the grid
    out = MaxBellmanVector.(AA)
    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.

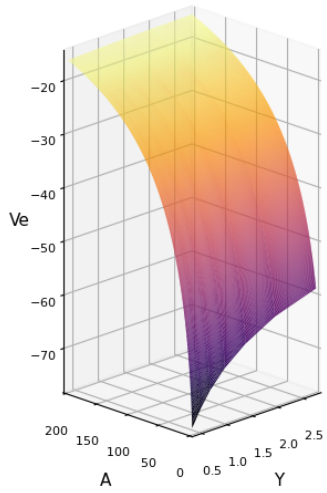
```julia
"""Update value function and A' lists of vectors with new vectors"""
function update_lists!(Velist, Vulist, Aelist, Aulist, Ve, Vu, Ae, Au)
    append!(Velist, [Ve])
    append!(Vulist, [Vu])
    append!(Aelist, [Ae])
    append!(Aulist, [Au])
end
function update_lists!(Velist, Vulist, Ve, Vu)
    append!(Velist, [Ve])
    append!(Vulist, [Vu])
end

"""Iterate over EV and Ap vectors to converge on the value function and Ap policy rule"""
function MyBellmanIteration(; verbose=true)
    # initial guess of the value functions (zero function)
    MAXIT = 2_000
    Velist, Vulist, Aelist, Aulist = [zeros(size(AA))], [zeros(size(AA))], [zeros(size(AA))], [zeros(size(AA))]
    for it = 1:MAXIT
        Ve, Ape = MyMaxBellmanE(Velist[end], Vulist[end])
        Vu, Apu = MyMaxBellmanU(Velist[end], Vulist[end])

        # 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
        EVe = reshape(Ve, NA, NY) * GridPY
        EVu = reshape(Vu, NA, NY) * GridPY

        # update our value functions
        update_lists!(Velist, Vulist, Aelist, Aulist, EVe[:], EVu[:], Ape, Apu)

        # see how much our policy rules and value functions have changed
        Aetest = maximum(abs.(Aelist[end] - Aelist[end-1]))
        Autest = maximum(abs.(Aulist[end] - Aulist[end-1]))
        Vetest = maximum(abs.(Velist[end] - Velist[end-1]))
        Vutest = maximum(abs.(Vulist[end] - Vulist[end-1]))

        if it % 50 == 0
            verbose ? println("iteration $it, Vetest = $Vetest, Vutest = $Vutest, Aetest = $Aetest, Autest = $Autest") : nothing
        end
        if max(Aetest, Autest, Vetest, Vutest) < 1e-5
            println("\nCONVERGED -- final iteration tests:")
            println("iteration $it, Vetest = $Vetest, Vutest = $Vutest, Aetest = $Aetest, Autest = $Autest")
            break
        end

        it == MAXIT ? println("\nMAX ITERATIONS REACHED ($MAXIT)") : nothing
    end

    return Velist, Vulist, Aelist, Aulist
end
```

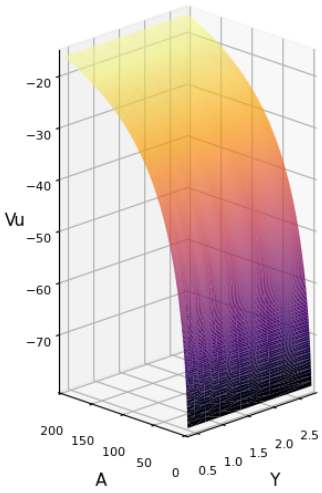
Then we can run the value function iteration and plot the results:

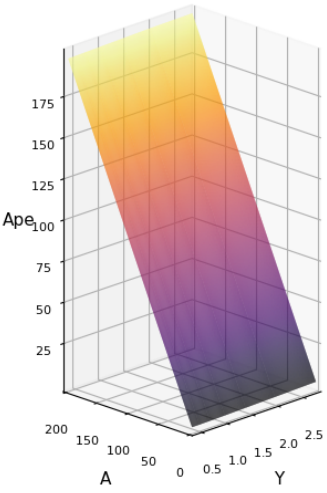
```julia
outH = @time MyBellmanIteration();
VeH = outH[1][end]; VuH = outH[2][end]; ApeH = outH[3][end]; ApuH = outH[4][end];

angle0 = (-45,30)
```

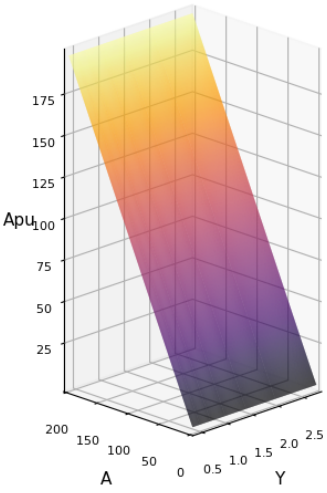
```
plotargs = (camera=(-45, 20), xlabel="Y", ylabel="A",
            legend=:none, aspect_ratio=[1,1,2])
pVe_H = surface(exp.(YY), AA, VeH, zlabel="Ve"; plotargs...)
pVu_H = surface(exp.(YY), AA, VuH, zlabel="Vu"; plotargs...)
pApe_H = surface(exp.(YY), AA, ApeH, zlabel="Ape"; plotargs...)
pApu_H = surface(exp.(YY), AA, ApuH, zlabel="Apu"; plotargs...)
pCe_H = surface(exp.(YY), AA, c.(YY, AA, ApeH), zlabel="Ce"; plotargs...)
pCu_H = surface(exp.(YY), AA, c.(bb, AA, ApuH), zlabel="Cu"; plotargs...)

pH1 = plot(pVe_H, pVu_H, pApe_H, pApu_H, pCe_H, pCu_H,
    layout=(3,2), size=(800, 1600))
savefig(pH1, "H-all_$GridA_upper")
```

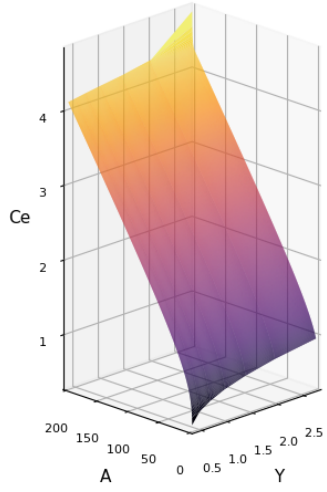
Resulting in these plots:

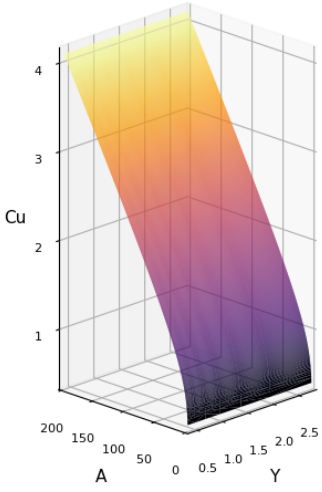

Ve
−20
−30
−40
−50
−60
−70
200
150
100
50
0
A
0.5
1.0
1.5
2.0
2.5
Y


Vu
−20
−30
−40
−50
−60
−70
200
150
100
50
0
A
0.5
1.0
1.5
2.0
2.5
Y


Ape
175
150
125
100
75
50
25
200
150
100
50
0
A
0.5
1.0
1.5
2.0
2.5
Y


Apu
175
150
125
100
75
50
25
200
150
100
50
0
A
0.5
1.0
1.5
2.0
2.5
Y


Ce
4
3
2
1
200
150
100
50
0
A
0.5
1.0
1.5
2.0
2.5
Y


Cu
4
3
2
1
200
150
100
50
0
A
0.5
1.0
1.5
2.0
2.5
Y

I: Howard acceleration

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

```
"""Iterate faster over the value function vectors to converge on the value function using Howard acceleration."""
function MyFasterBellmanIteration(; inner_mod = 32, verbose=false)
    println("\nStarting MyFasterBellmanIteration with inner_mod = $inner_mod")
    # initial guess of the value functions (zero function)
    MAXIT = 20_000; inner_it = 0
    Velist, Vulist, Aelist, Aulist = [zeros(size(AA))], [zeros(size(AA))], [zeros(size(AA))], [zeros(size(AA))]
    for it = 1:MAXIT
        # Every inner_mod iterations, get the maximizing A'
        if it % round(inner_mod) == 1
            Ve, Ape = MyMaxBellmanE(Velist[end], Vulist[end])
            Vu, Apu = MyMaxBellmanU(Velist[end], Vulist[end])
            update_lists!(Velist, Vulist, Aelist, Aulist, Ve, Vu, Ape, Apu)
            inner_it += 1
        else
            # V = bellman(EV)
            Ve = BellmanE(Velist[end], Vulist[end], AA, YY, Aelist[end])
            Vu = BellmanU(Velist[end], Vulist[end], AA, Aulist[end])
        end

        # EV = reshape(V, NA, NY) * GridPY
        EVe = reshape(Ve, NA, NY) * GridPY
        EVu = reshape(Vu, NA, NY) * GridPY

        # append!(EVlist, EV[:])
        update_lists!(Velist, Vulist, EVe[:], EVu[:])

        # see how much our policy rules and value functions have changed
        Aetest = maximum(abs.(Aelist[end] - Aelist[end-1]))
        Autest = maximum(abs.(Aulist[end] - Aulist[end-1]))
        Vetest = maximum(abs.(Velist[end] - Velist[end-1]))
        Vutest = maximum(abs.(Vulist[end] - Vulist[end-1]))

        if it % 50 == 0
            verbose ? println("iteration $it, Vetest = $Vetest, Vutest = $Vutest, Aetest = $Aetest, Autest = $Autest") : nothing
        end
        if max(Aetest, Autest, Vetest, Vutest) < 1e-5
            verbose ? println("\nCONVERGED in $it iterations, $inner_it maximization iterations -- final iteration tests:") : nothing
            verbose ? println("iteration $it, Vetest = $Vetest, Vutest = $Vutest, Aetest = $Aetest, Autest = $Autest") : nothing
            break
        end

        it == MAXIT ? println("\nMAX ITERATIONS REACHED ($MAXIT)") : nothing
    end

    return Velist, Vulist, Aelist, Aulist
end
```

Using Julia's timing functions, I tested how fast I could get the function using different modulos to moderate the inner maximization evaluations.

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

# Find which modulo for the bellman A' maximization results in shortest time
mods = 10:100
f(x::Real) = @elapsed @time MyFasterBellmanIteration(inner_mod=x; verbose=false);
f(x::AbstractArray) = @elapsed MyFasterBellmanIteration(inner_mod=x[1]; verbose=false);
```

```
times = f.(mods)
mintime0, minidx = findmin(times)
minmod = mods[minidx]

# Compare to unaccelerated iteration
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 32 resulted in 9.83 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 and helper functions:

```
1
2  struct SimReturn
3      Ap # Assets at beginning of next period
4      A  # Assets at beginning of period
5      Ytilde # Employment Income
6      Y  # Income = labor income*X + unemp income *(1-X)
7      C  # Consumption
8      X  # employment
9  end
10
11
12 # Define probability distributions for employment next period
13 distE = Binomial(1, 1-p)  # if emp this period, 1-p Prob of being emp next period
14 distU = Binomial(1, q)    # if unemp this period, q Prob of being emp next period
15
16 """Return employment status next period X' based on employment status this period X"""
17 function employment_next(X)
18     Xp = X==1 ? rand(distE, 1)[1] : rand(distU, 1)[1]
19     return Xp
20 end
```

And then write the function to simulate.

```
1  """
2  Sim = Simulate(ApeI, ApuI, Mode, T)
3      Simulates the model.
4      Inputs:
5      ApeI       Estimated A' emplyed policy rule (interpolated function of A_t, lnY_t)
6      ApuI       Estimated A' unemployed policy rule (interpolated function of A_t, lnY_t)
7      Mode       Mode = 'random' -> draw shocks
8                 Mode = 'irf'    -> impulse response function
9      T          # of periods to simulate
10 """
11 function Simulate(Ape, Apu, Mode, T)
12     Random.seed!(123);
13     A = zeros(T)  # Assets
14     Ytilde = zeros(T)  # Employment income
15     Y = zeros(T)  # Realized Income (employed or not)
16     X = zeros(T)  # Employment status {0,1}
17     A[1] = mean(GridA)
18     X[1] = 1  # Start employed
19
20     if Mode == "irf"
21         Ytilde[1] = σ
22         ε = zeros(T)
23     elseif Mode == "random"
24         Ytilde[1] = log(μ)
25         ε = σ * randn(T)
26     else
27         throw("Unrecognized Mode $Mode in Simulate()");
28     end
29
30     for t ∈ 2:T
31         # println("t=$t, A = $(A[t-1]), Y = $(Y[t-1]), X = $(X[t-1])")
32         # Employed?
33         X[t] = employment_next(X[t-1])
34         # Labor income
35         Ytilde[t] = lnY(Ytilde[t-1], ε[t])
36         # Realized income (considering unemployment)
37         Y[t] = X[t]*Ytilde[t] + (1-X[t])*log(bb)
38         # Realized Assets
39         ApFunc = X[t]==1 ? Ape : Apu
40         A[t] = max(ApFunc(A[t-1], Y[t-1]), 0)  # A_t ≥ 0
```

```
    end

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

    return SimReturn(Ap, A, Ytilde, Y, C, X)
end
```

K: The 10,000 period simulation

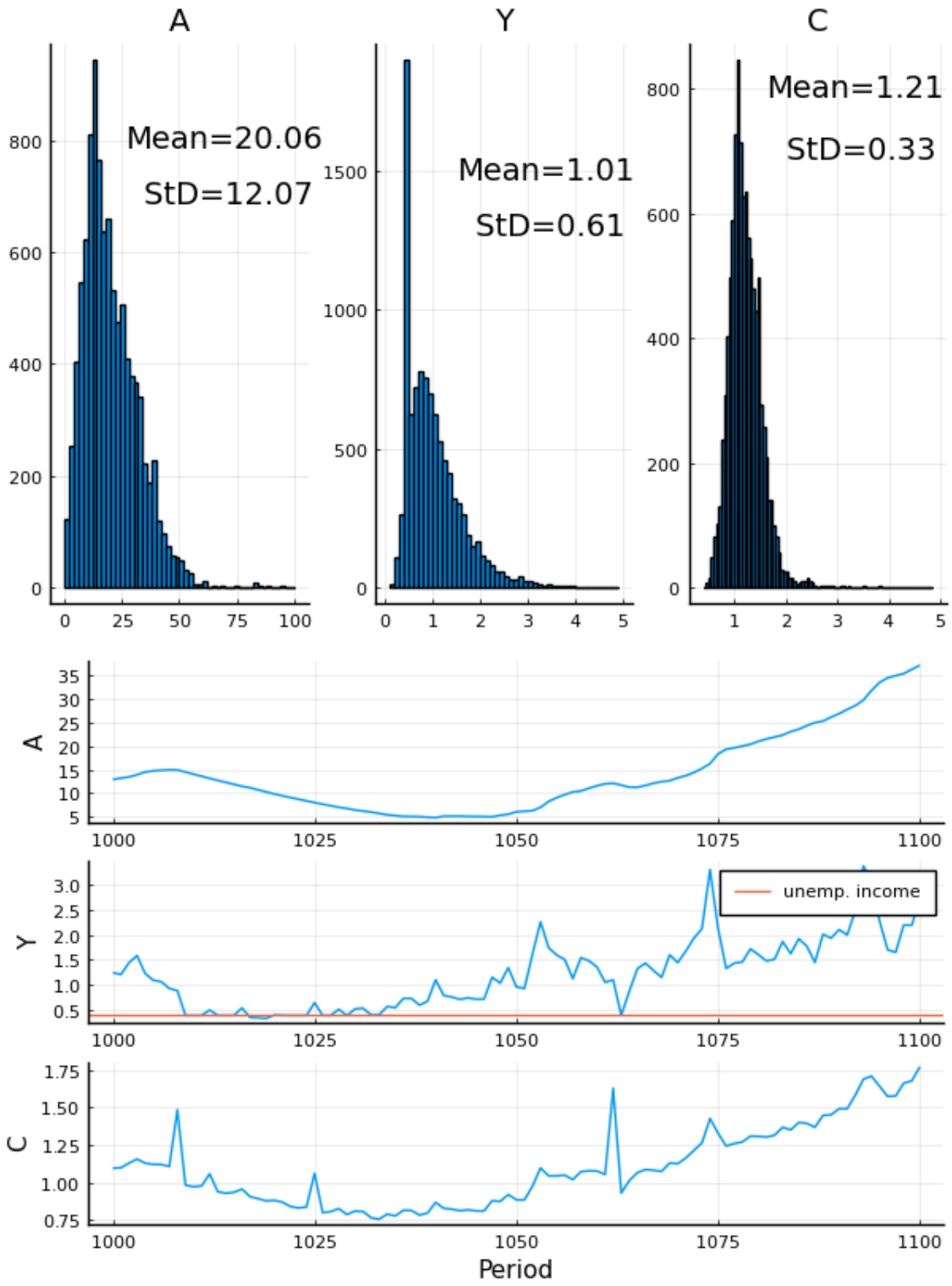
Using your new Simulate function, produce a 10,000 period simulation of the evolution of A, $\tilde{Y}$, C, and employment status for a single household. Report a histogram of A, Y , and C. Report the mean and standard deviation of each variable. Plot the evolution of A, Y , and C over a 100 period stretch starting from period 1000.

First simulate:

```
# Generate interpolated A' functions of Ape and Apu
ApeFunc(A, lnY) = interpolate_EV(ApeI)(A, lnY)
ApuFunc(A, lnY) = interpolate_EV(ApuI)(A, lnY)

# Simulate
nPeriods = 10_000
@time SimK = Simulate(ApeFunc, ApuFunc, "random", nPeriods);
```

Then create the outputs:

```
# 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])

# Report a histogram of A, Y , and C. (Note: Yt not log Yt .)
hk1 = histogram(SimK.A, title="A", label="") #, xlims=(61,64)
annotate!(hk1, [(0.7*maximum(SimK.A), 800, "Mean=$(round(meansk[1], digits=2))"),
                (0.7*maximum(SimK.A), 700, " StD=$(round(sdk[1], digits=2))")])
hk2 = histogram(e.^SimK.Y, title="Y", label="")
annotate!(hk2, [(0.7*maximum(e.^SimK.Y), 1500, "Mean=$(round(meansk[2], digits=2))"),
                (0.7*maximum(e.^SimK.Y), 1300, " StD=$(round(sdk[2], digits=2))")])
hk3 = histogram(SimK.C, title="C", label="") #, xlims=(31,35)
annotate!(hk3, [(0.7*maximum(SimK.C), 800, "Mean=$(round(meansk[3], digits=2))"),
                (0.7*maximum(SimK.C), 700, " StD=$(round(sdk[3], digits=2))")])
hk4 = plot(hk1, hk2, hk3, layout=(1,3))
savefig(hk4, "K-histograms")


# 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="")
hline!(pk2, [bb], label="unemp. income")
pk3 = plot(periods, SimK.C[periods], ylabel="C", xlabel="Period", label="")
pk4 = plot(pk1, pk2, pk3, layout=(3,1))
savefig(pk4, "K-evolutions")
```

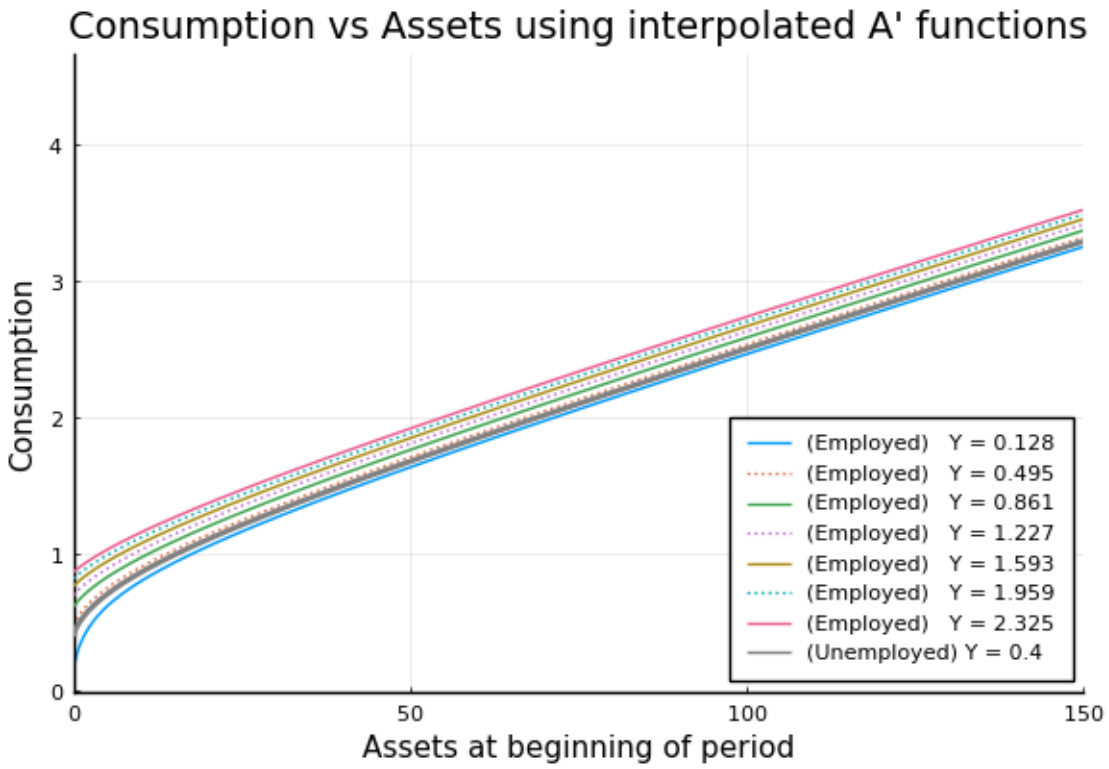


We can see that the assets are relatively much smoother compared to income and consumption, but have a much larger standard deviation because the path changes a lot.

L: Plot consumption

Plot consumption as a function of A for several values of $\tilde{Y}$. Do this for the a range of values for A that encompasses most of the mass in the histogram you report in part K. On the same figure, also plot consumption when unemployed as a function of A.

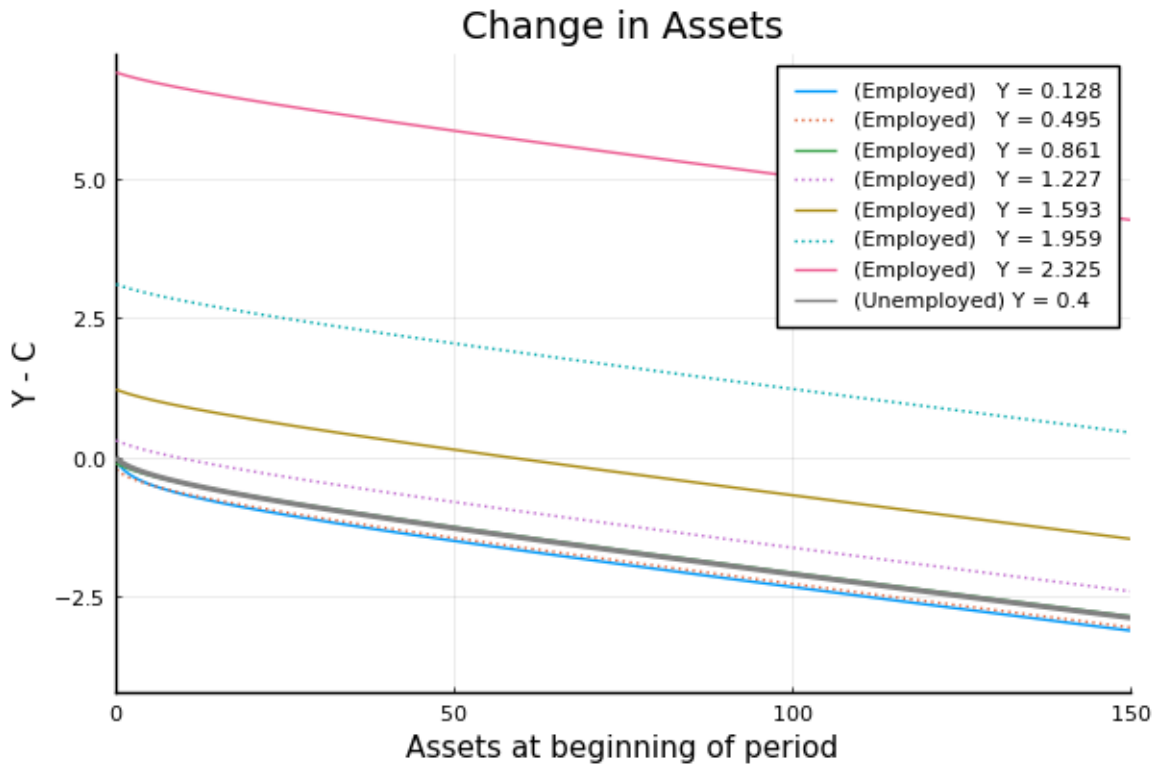
```
# Generate values of Ỹ to plot for
YmeanL = mean(e.^SimK.Ytilde); YsdL = std(e.^SimK.Ytilde);
Yvals = range(GridY[1], YmeanL + 2*YsdL, 7)

# Generate Consumption at values
cLe(A, lnY) = c(lnY, A, ApeFunc(A, lnY))  # employed
cLu(A) = c(log(bb), A, ApuFunc(A, log(bb)))  # unemployed
CL = [[cLe(a, log(y)) for a in GridA, y in Yvals] cLu.(GridA)]

# PLOT USING A' APPROXIMATION
labelsL = "(Employed)   Y = " .* string.(round.(Yvals, digits=3)')
labelsL = [labelsL ["(Unemployed) Y = $bb"]]
linestylesL = [:solid :dot :solid :dot :solid :dot :solid :solid]
linecolorsL = [repeat([:auto], outer=(1,7)) [:grey]]
linewidthsL = [repeat([:auto], outer=(1,7)) [2]]
pL1 = plot(GridA, CL,
     label=labelsL,
     linestyle=linestylesL,
     linecolor=linecolorsL,
     linewidth=linewidthsL,
     legend=:bottomright,
     xlims=(0,150),
     xlabel="Assets at beginning of period",
     ylabel="Consumption",
     title="Consumption vs Assets using interpolated A' functions")
savefig(pL1, "L-consumption")
```



M: Plot change in assets

Plot change in assets $Y - C$ when employed as a function A for several values of $\tilde{Y}$. Do this for the same range of values for A as part L. On the same figure, also plot change in assets when unemployed as a function of A. Do this for the average value of $\tilde{Y}$.

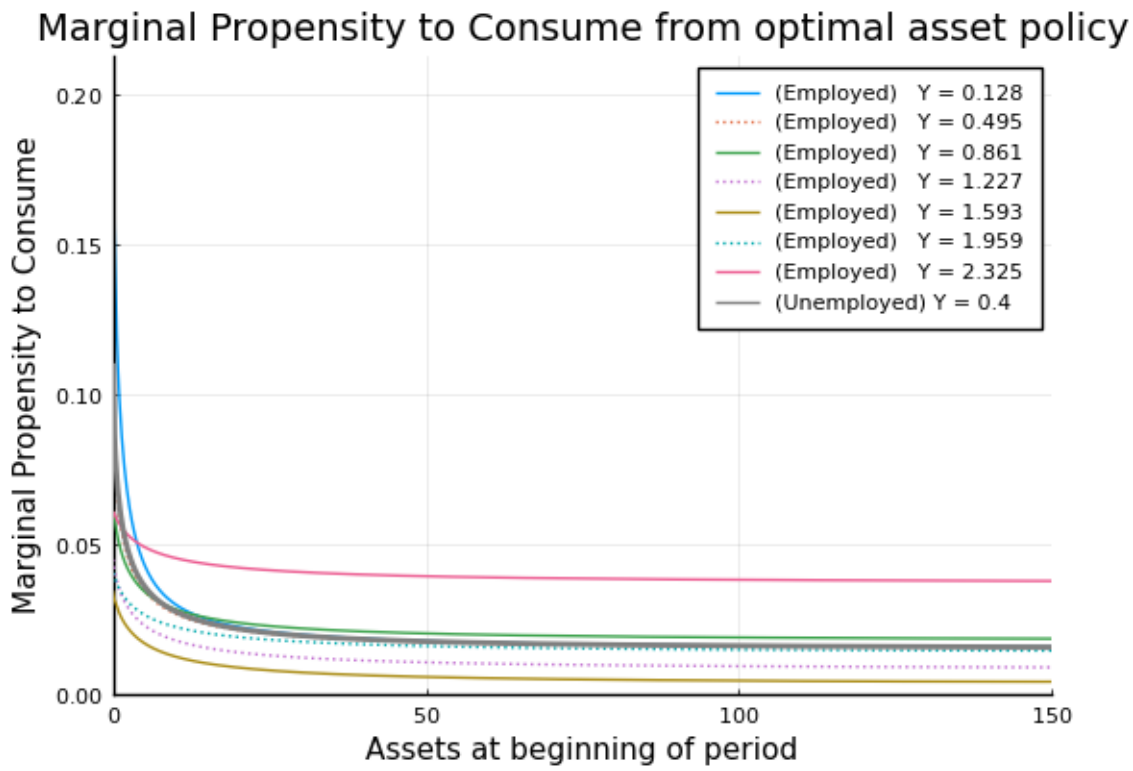
```
# PLOT USING A' APPROXIMATION
YM = [reshape(e.^YY, NA, NY) repeat([bb], NA)]

pM1 = plot(GridA, YM .- CL,
     label=labelsL,
     linestyle=linestylesL,
     linecolor=linecolorsL,
     linewidth=linewidthsL,
     legend=:topright,
     xlims=(0,150),
     xlabel="Assets at beginning of period",
     ylabel="Y - C", title="Change in Assets")
savefig(pM1, "M-changeinassets")
```



N: Plot the marginal propensity to consume

Plot the marginal propensity to consume when employed as a function of A for several values of $\tilde{Y}$. On the same figure, also plot the marginal propensity to consume when unemployed as a function of A (again for the average value of $\tilde{Y}$). 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.

```
# Get index of points that will be inside the domain after shifting up by one
idxGridA_inside = findall(GridA_lower .<= GridA .+1 .<= GridA_upper)
# Filter previous consumption matrix to just those rows
CL_inside = CL[idxGridA_inside, :]
# Generate new consumption matrix for the shifted up grid
GridA_inside = GridA[idxGridA_inside]
GridA_shift = GridA_inside .+ 1
CN = [[cLe(a, log(y)) for a in GridA_shift, y in Yvals] cLu.(GridA_shift)]
# PLOT USING A' APPROXIMATION on smaller subset of GridA
MPC = CN .- CL_inside
pN1 = plot(GridA_inside, MPC,
     label=labelsL,
     linestyle=linestylesL,
     linecolor=linecolorsL,
     linewidth=linewidthsL,
     legend=:topright,
     xlims=(0,150), ylims=(0, maximum(MPC)),
     title="Marginal Propensity to Consume from optimal asset policy",
     xlabel="Assets at beginning of period",
     ylabel="Marginal Propensity to Consume")
savefig(pN1, "N-mpc")
```



Appendix: Packages

```
import Pkg
Pkg.activate(pwd())
try
    using Optim, Parameters, Plots, Revise, DataFrames, Interpolations, Distributions, Random
    using Optim: maximum, maximizer
    pyplot()
catch e
    Pkg.add(["Plots", "PyPlot", "Optim", "Parameters", "Revise", "DataFrames", "Interpolations", "
    Distributions", "Random"])
    using Optim, Parameters, Plots, Revise, DataFrames, Interpolations, Distributions, Random
    using Optim: maximum, maximizer
    pyplot()
end
includet("Tauchen.jl")
```

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/

Optimization:
https://julianlsolvers.github.io/Optim.jl/stable/
```