Structural Estimation Pset 4

Timothy Schwieg

The functions used to estimate this model are given below. Note that for the constrained optimization, the logit-transform is applied to transform the variables from the parameter space, which is bounded to $\mathbb{R}^4$ which is unbounded. This allows the use of Unconstrained optimization libraries which are much more effective. In order to aid with the optimization process, automatic differentiation is applied to the objectives.

```julia
  function BuildSim( μ::Real, α::Real, ρ::Real, σ::Real, β::Real,
                   S::Int64, T::Int64, initK::Real, c::Matrix{Real},
                   k::Matrix{Real}, w::Matrix{Real},
                   r::Matrix{Real}, y::Matrix{Real}, z::Matrix{Real},
                   u::Matrix{Float64})
    ▯ = σ*quantile.( Normal(), u)

    for s in 1:S
        z[s,1] = μ
        k[s,1] = initK#mean(macroData[:K])
        for t in 1:T
            #We have z shift up by one to deal with the fact we are
            #1-indexed.
            #z[1] = z_0
            z[s,t+1] = ρ*z[s,t] + (1-ρ)*μ + ▯[s,t]
            k[s,t+1] = α*β*exp(z[s,t+1])*k[s,t]^α
            w[s,t] = (1-α)*exp(z[s,t+1])*k[s,t]^α
            r[s,t] = α*exp(z[s,t+1])*k[s,t]^(α-1)
            c[s,t] = r[s,t]*k[s,t] + w[s,t] - k[s,t+1]
            y[s,t] = exp(z[s,t+1])*k[s,t]^α
        end
    end
end

function myVar( x::Vector{Real})
    return (sum( x[i]*x[i] for i in 1:100 ) - sum(x)*sum(x) / 100.0) / 99.0
end



function BuildMoments( dC::Vector{Float64}, dK::Vector{Float64},
                       dW::Vector{Float64}, dR::Vector{Float64},
                       dY::Vector{Float64}, sC::Matrix{Real},
                       sK::Matrix{Real}, sW::Matrix{Real},
                       sR::Matrix{Real}, sY::Matrix{Real},
                       momentBox::Vector{Real},S::Int64 )
    momentBox[1] = ( mean(mean(sC[i,:] for i in 1:S)) - mean(dC)) / mean(dC)
    momentBox[2] = (mean(mean(sK[i,:] for i in 1:S))- mean(dK) ) / mean(dK)
    momentBox[3] = (mean( mean(sC[i,:] ./ sY[i,:] for i in 1:S)) - mean( dC ./ dY) ) / mean( dC ./ dY)
    momentBox[4] = (mean([myVar(sY[i,:]) for i in 1:S]) - var( dY) ) / var(dY)
    momentBox[5] = ( mean([cor(sC[i,1:99],sC[i,2:100]) for i in 1:S]) - cor(dC[1:99],dC[2:100])) / cor(dC[1:99],dC[2:100])
    momentBox[6] = (mean( [cor(sC[i,:],sK[i,1:100]) for i in 1:S] ) - cor(dC,dK)) / cor(dC,dK)
end


```

```julia


function objective(μ::Real, α::Real, ρ::Real, σ::Real, β::Real,
                   S::Int64, T::Int64, initK::Real, c::Matrix{Real},
                   k::Matrix{Real}, w::Matrix{Real},
                   r::Matrix{Real}, y::Matrix{Real}, z::Matrix{Real},
                   u::Matrix{Float64}, dC::Vector{Float64}, dK::Vector{Float64},
                   dW::Vector{Float64}, dR::Vector{Float64},
                   dY::Vector{Float64}, W::Matrix{Real} )

    m = Moments( μ, α, ρ, σ, β, S, T, initK, c, k, w, r, y, z, u, dC, dK, dW, dR, dY, W )
    return dot( m, W*m)#sum( m[i]*m[i] for i in 1:6)
end

function Moments(μ::Real, α::Real, ρ::Real, σ::Real, β::Real,
                   S::Int64, T::Int64, initK::Real, c::Matrix{Real},
                   k::Matrix{Real}, w::Matrix{Real},
                   r::Matrix{Real}, y::Matrix{Real}, z::Matrix{Real},
                   u::Matrix{Float64}, dC::Vector{Float64}, dK::Vector{Float64},
                   dW::Vector{Float64}, dR::Vector{Float64},
                 dY::Vector{Float64}, W::Matrix{Real} )
    BuildSim( μ, α, ρ, σ, β, S, T, initK, c, k, w, r, y, z, u)
    m = Vector{Real}(undef,6)
    BuildMoments( dC, dK, dW, dR, dY, c, k, w, r, y, m, S)
    return m
end
```

The data is loaded, and objects are manipulated such that the optimization method can then work on them.

```julia
  macroData = DataFrame(load("data/NewMacroSeries.csv", header_exists=false, colnames=["C", "K", "W", "R", "Y"]))

S = 1000
T = 100
u = rand(Uniform(0,1),S,T)

▯ = Matrix{Real}(undef,S,T)
z = Matrix{Real}(undef,S,T+1)
k = Matrix{Real}(undef,S,T+1)
w = Matrix{Real}(undef,S,T)
r = Matrix{Real}(undef,S,T)
c = Matrix{Real}(undef,S,T)
y = Matrix{Real}(undef,S,T)

# The built in I will not cast to type Real
# which we need to differentiate.
W = Matrix{Real}(undef,6,6)
W .= 0
for i in 1:6
    W[i,i] = 1.0
end



f(x) = objective( LogitTransform(x[1],5.0, 14.0),
                  LogitTransform(x[2], .01, .99),
                  LogitTransform(x[3], -.99, .99),
                  LogitTransform(x[4], 0.01, 1.1),
                  .99, S, T, mean(macroData[:K]), c, k, w, r, y, z, u, macroData[:C], macroData[:K],
 ↪   macroData[:W], macroData[:R], macroData[:Y], W)

θ = [ InverLogit(5.0729,5.0, 14.0),
      InverLogit(.70216, .01, .99),
      InverLogit(.47972, -.99, .99),
      InverLogit(.05, 0.01, 1.1) ]
```

```

results = optimize(f, θ, Newton(), autodiff=:forward)
```

The results from this optimization are printed below:

```
Results of Optimization Algorithm
 * Algorithm: Newton's Method
 * Starting Point: [4.8077582340735585,-0.877412372562471, ...]
 * Minimizer: [-0.19288333200456412,0.32513578293163214, ...]
 * Minimum: 4.331495e-06
 * Iterations: 69
 * Convergence: true
   * |x - x'| ≤ 0.0e+00: false
     |x - x'| = 2.27e-07
   * |f(x) - f(x')| ≤ 0.0e+00 |f(x)|: false
     |f(x) - f(x')| = 1.56e-10 |f(x)|
   * |g(x)| ≤ 1.0e-08: true
     |g(x)| = 6.33e-13
   * Stopped by an increasing objective: false
   * Reached Maximum Number of Iterations: false
 * Objective Calls: 227
 * Gradient Calls: 227
 * Hessian Calls: 69
```

This minimum corresponds to the following parameter values estimated:

μ	9.932646978878989
α	0.42103613802768947
ρ	0.9193643618762394
σ	0.08951209454130482

The final values of the moments are given below:

$$\widehat{m}_n = \begin{pmatrix} 0.00073009135798180 42 \\ -0.0007376556190434634 \\ -0.0017558655381804127 \\ -9.688593198418377 \times 10^{-9} \\ 0.00029393694814873174 \\ -0.00029131201720979767 \end{pmatrix}$$

The Jacobian of the moment function is then estimated via automatic differentiation, and the variance-covariance matrix estimated to compute the standard errors.

```
answer = [LogitTransform(x[1],5.0, 14.0),
          LogitTransform(x[2], .01, .99),
          LogitTransform(x[3], -.99, .99),
          LogitTransform(x[4], 0.01, 1.1)]

m(x) = Moments( x[1], x[2], x[3], x[4], .99, S, T,  mean(macroData[:K]), c, k, w, r, y, z, u, macroData[:C],
 ↪   macroData[:K], macroData[:W], macroData[:R], macroData[:Y], W)

```

```
mom = m(answer)

J = ForwardDiff.jacobian( m, answer )

varMat = (1/S)*inv( J' * W*J)
stdErrors = [sqrt(varMat[i,i]) for i in 1:4]
```

These errors are given below:

$$\begin{pmatrix} 0.1604770701111893 \\ 0.009499100361935424 \\ 0.048231362793496046 \\ 0.020361964338113037 \end{pmatrix}$$

The optimal Weighting matrix is then constructed by using the E matrix as suggested in the notebook, and then summing over the outer-product of each simulation's contributions.

```
  dC = macroData[:C]
dK = macroData[:K]
dW = macroData[:W]
dR = macroData[:R]
dY = macroData[:Y]

E = Matrix{Real}(undef,6,S)
for i in 1:S
    E[1,i] = (mean(c[i,:]) - mean(dC)) / mean(dC)
    E[2,i] = (mean(k[i,:])- mean(dK) ) / mean(dK)
    E[3,i] = (mean(c[i,:] ./ y[i,:] ) - mean( dC ./ dY) ) / mean( dC ./ dY)
    E[4,i] = (myVar(y[i,:]) - var( dY) ) / var(dY)
    E[5,i] = ( cor(c[i,1:99],c[i,2:100])-cor(dC[1:99],dC[2:100])) / cor(dC[1:99],dC[2:100])
    E[6,i] = (cor(c[i,:],k[i,1:100]) - cor(dC,dK)) / cor(dC,dK)
end


wHat = convert( Matrix{Real},inv((1/S)*sum( E[:,i]*E[:,i]' for i in 1:S)))
```

The second stage of optimiation procedes as the first did, but with a different matrix specified.

```
fOpt(x) = objective( LogitTransform(x[1],5.0, 14.0),
                  LogitTransform(x[2], .01, .99),
                  LogitTransform(x[3], -.99, .99),
                  LogitTransform(x[4], 0.01, 1.1),
                  .99, S, T, mean(macroData[:K]), c, k, w, r, y, z, u, macroData[:C], macroData[:K],
 ↪   macroData[:W], macroData[:R], macroData[:Y], wHat)
resultsOpt = optimize(fOpt, x, Newton(), autodiff=:forward)

xOpt = results.minimizer

#μ, α, ρ, σ
answerOpt = [LogitTransform(xOpt[1],5.0, 14.0),
          LogitTransform(xOpt[2], .01, .99),
          LogitTransform(xOpt[3], -.99, .99),
          LogitTransform(xOpt[4], 0.01, 1.1)]
```

The results of the optimization are printed below:

```
Results of Optimization Algorithm
 * Algorithm: Newton's Method
 * Starting Point: [-0.19288333200456412,0.32513578293163214, ...]
 * Minimizer: [-0.19275858830581438,0.3251357829316348, ...]
 * Minimum: 9.999802e-01
 * Iterations: 8
 * Convergence: true
   * |x - x'| ≤ 0.0e+00: true
     |x - x'| = 0.00e+00
   * |f(x) - f(x')| ≤ 0.0e+00 |f(x)|: true
     |f(x) - f(x')| = 0.00e+00 |f(x)|
   * |g(x)| ≤ 1.0e-08: false
     |g(x)| = 2.93e+01
   * Stopped by an increasing objective: false
   * Reached Maximum Number of Iterations: false
 * Objective Calls: 175
 * Gradient Calls: 175
 * Hessian Calls: 8
```

The values computed are as follows:

μ	9.932646978878989
α	0.42103613802768947
ρ	0.9193643618762394
σ	0.08951209454130482

The final values of the moments are given below:

$$\widehat{m}_n = \begin{pmatrix} 0.0007300913579818042 \\ -0.0007376556190434634 \\ -0.0017558655381804127 \\ -9.688593198418377 \times 10^{-9} \\ 0.00029393694814873174 \\ -0.00029131201720979767 \end{pmatrix}$$

The standard errors are computed by the same procedure:

```julia
mOpt(x) = Moments( x[1], x[2], x[3], x[4], .99, S, T,  mean(macroData[:K]), c, k, w, r, y, z, u, macroData[:C],
 ↪   macroData[:K], macroData[:W], macroData[:R], macroData[:Y], wHat)

momOpt = mOpt(answerOpt)

JOpt = ForwardDiff.jacobian( m, answerOpt )
varMatOpt = (1/S)*inv( JOpt' * wHat*JOpt )
stdErrorsOpt = [sqrt(varMatOpt[i,i]) for i in 1:4]
```

They are given below:

$$\begin{pmatrix} 0.003052707245892253 \\ 4.9593863300683934 \times 10^{-11} \\ 0.0018265043575507948 \\ 0.0005365640009272416 \end{pmatrix}$$