Advanced Topics Macro I - Assignment 4

Andrea Titton & Alessandro Zona Mattioli

November 2020

Question 1

The main code that runs the solution of our problem is stored in run-four.jl. We start by defining a process called Aiyagari, where we store the main elements of the model at hand, including the parameters and the discretization of the y_t process, using the Rouwenhorst method.

We then move to solving the Partial equilibrium problem, via the function policysolve, which is in turn a wrapper around the two different solvers endgrid and iterate_pfi.

We start by describing the endogenous grid method. This is stored in week-four/algos/policy/endgrid.jl.

First the function loads the elements of the model such as parameters and utility functions, takes a first guess for a'' then starts the iterations. At first, the code solves linearly for $a(a', y)$ and interpolates on $a(a', y)$ and $a\prime$ to derive the next guess of a''. The interpolation is done via fromMtoFn which simply takes two vectors and a function and return the surface linear interpolation over it.

We do this for each element of the support of y. We iterate until convergence and then return the policy as a function.

```
1
2  function endgrid(
3      a_grid::Vector{Float64}, ai::Aiyagari, R::Float64, w::Float64;
4      n_steps=1_000, verbose=false, tol=1e-3, max_iter=1_000)
5
6      u, u′, inv_u′ = make_u(ai)
7      @unpack β, a_, y = ai
8
9      Γ = y.P
10     support_y = y.S
11     cond_dens(x) = Γ[get_closest(support_y, x), :]
12
13     ys = collect(support_y)
14     shocks = y.transformation.(ys)
15
16     T, N = length(ys), length(a_grid)
17
18     policy = (a, y) -> a + y # Initial guess
19
20     as_origin = zeros(N, T)
21     prev_policy = zeros(N, T)
22
23     for iter in 1:max_iter
24
25         @threads for i in 1:N
26             for j in 1:T
```

```
                a_p = a_grid[i]

                vals = @. u′(R * a_p + w * shocks - policy(a_p, ys))

                c = β * R * E(vals, cond_dens(ys[j]))

                as_origin[i, j] = (inv_u′(c) - w * ys[j] + a_p) / R
            end
        end

        eval_policy = zeros(N, T)

        for j in 1:T
            itps = LinearInterpolation(as_origin[:, j], a_grid,
                extrapolation_bc=Line())
            eval_policy[:, j] = @. max(itps(a_grid), a_)
        end

        err_distance = matrix_distance(eval_policy, prev_policy)

        if err_distance < tol
            verbose && print("Found policy in $ iter iterations (|x -
                x'| = $ (@sprintf("%.4f", err_distance))\n")
            return (a, y) -> max(policy(a, y), a_)
        end

        policy = fromMtoFn(a_grid, ys, eval_policy)
        prev_policy = eval_policy

    end

    @warn "Could not find policy in $ max_iter iterations with
        tolerance $ tol"

    return policy
end
```

Concerning the other method, we store it in week-four/algos/policy/pfi.jl. The procedure is quite similar to the previous function. The main difference is that within the loop we set up right and left hand sides of the non linear equation and then call a solver to find the a' which can solve the non-linear system.

```

function iterate_pfi(
    as::Vector{Float64}, ai::Aiyagari, R::Float64, w::Float64;
    max_iter=1_000, tol=1e-3, verbose=false
)

    u, u′, inv_u′ = make_u(ai)
    @unpack β, a_, y = ai

```

```
    support_y = y.S
    ys = collect(support_y)
    shocks = y.transformation.(ys)
    cond_dens(x) = Γ[get_closest(support_y, x), :]
    Γ = y.P

    N, T = length(as), length(ys)

    a_grid = copy(as)
    domain = cartesian(a_grid, ys)

    a_prime = ones(N, T)

    a′ = fromMtoFn(a_grid, ys, a_prime)

    for iter in 1:max_iter

        next_a_prime = copy(a_prime)

        @threads for i in 1:N
            t_a = a_grid[i]
            for j in 1:T
                t_y = ys[j]

                vals(a_p) = @. u′(R * a_p + w * shocks - a′(a_p, shocks
                    ))
                rhs(a_p) = β * R * E(vals(a_p), cond_dens(t_y))
                lhs(a_p) = u′(R * t_a + w * t_y - a_p)

                a0 = a′(t_a, t_y)
                try
                    solution = find_zero(z -> rhs(z) - lhs(z), a0)
                    next_a_prime[i, j] = max(solution, a_)
                catch e end

            end
        end

        err_distance = matrix_distance(a_prime, next_a_prime)

        verbose && print("Iteration $iter / $max_iter: $(@sprintf("
            %.4f", err_distance)) \r")

        if err_distance < tol
            verbose && print("Found policy in $iter iterations (|x - x
                '| = $(@sprintf("%.4f", err_distance))\n")
            return a′
        end

        a_prime = next\_a\_prime
        a′ = fromMtoFn(a_grid, ys, a_prime)
```

```
    end

    @warn "Could not find policy in $max_iter iterations with
        tolerance $tol"

    return policy
end
```

Unfortunately, the second method does not deliver satisfactory results, but the endogenous grid method results in the following figure:

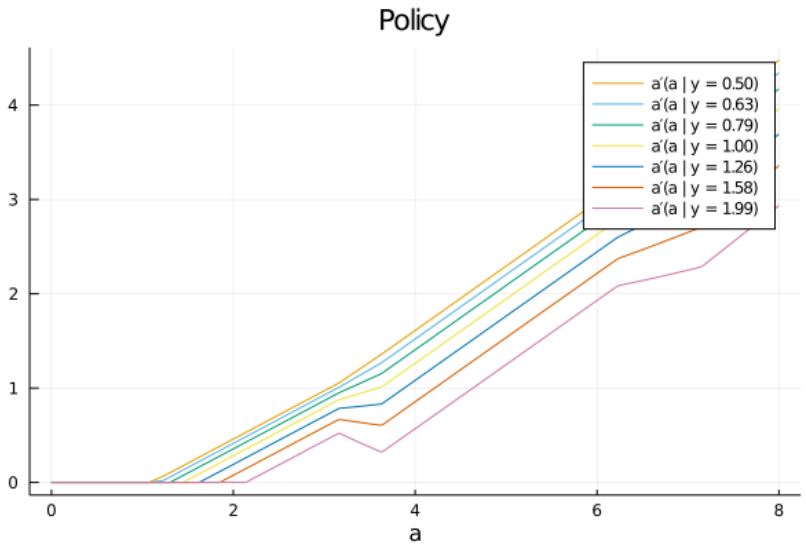

Figure 1: Policy function for different values of y.

We then move on to the steady state of the economy. We compute the steady state using an approximation to the density and the Monte-Carlo method. We deploy the two methods in the distribution_pdf function, which calls distribution_mc for the Monte-Carlo version and distribution_eigenvector for the density approximation. The former is stored in week-four/algos/stationary/montecarlo.jl and proceeds as following: it picks an initial grid for y_0, a_0 and iteratively draws the next step for y and applies the policy function a′. Then the function computes the distance between a set of moments of the newly computed a' distribution and of the previous a. If the distance is small enough we have achieved the stationary distribution, otherwise the iteration goes forward.

```

function distribution_mc(
    a′, a_grid::Vector{Float64}, model::Aiyagari;
    inits=1_000, verbose=false, max_iter=500, tol=1e-2)
```

```
5
6       markov = model.y
7       sim = (y0) -> discrete_sim(markov, T=2, drop=1, y0=y0)[2]
8
9       y0s = sample(markov.S, inits)
10      a0s = sample(a_grid, inits)
11
12      min_err = Inf
13
14      for i in 1:max_iter
15
16          y_next = sim.(y0s)
17          a_next = a′.(a0s, y0s)
18
19          err = norm(moments(a_next) - moments(a0s))
20
21          if verbose
22              min_err = min(err, min_err)
23              print(" $i / $max_iter : $err - $min_err \r")
24          end
25
26          if err < tol
27              print("Found stationary!\n\n")
28              return kde(a_next)
29          end
30
31      end
32
33      @warn "Algorithm did non converge with tol= $tol in $max_iter
            iterations"
34
35      return kde(a0s)
36
37
38  end
```

Moving on to the other approach, we use the a_todensity function to compute the transition matrix $Q_a(a'; a, y)$, using the "lottery" approach we adopted also for density discretization:

```
1
2   function a_todensity(
3       Q_aprime::Array{Float64,2},
4       a_grid::Array{Float64,1})
5
6       function populate(a::Float64)
7           dens_vec = spzeros(length(a_grid))
8           for (k, ak) in enumerate(a_grid)
9               if a < ak
10                  if k == 1
11                      dens_vec[k] = 1
12                      return dens_vec
```

```
                else
                    ak_1 = a_grid[k - 1]
                    dens_vec[k - 1] = (ak - a) / (ak - ak_1)
                    dens_vec[k] = (a - ak_1) / (ak - ak_1)

                    return dens_vec
                end
            end
        end

        dens_vec[end] = 1
        return dens_vec
    end

    return hcat(populate.(Q_aprime)...)'
end
```

Then we construct the $Q(a', y', a, y)$ by iteratively Kroenecker-multiplying each block of Q_a by the Γ transition matrix in computeQ:

```

function computeQ(
        a′::Function, a_grid::Array{Float64}, ai::Aiyagari;
        fact_finer::Int=2
    )
    ys = ai.y.S
    Γ = ai.y.P

    finer_grid = collect(
        range(a_grid[1], a_grid[end], length=length(a_grid) *
            fact_finer)
    )

    double_grid = collect.(Iterators.product(finer_grid, ys))
    N, T = size(double_grid)

    Q_aprime = (v -> a′(v...)).(double_grid)

    Q_a = a_todensity(Q_aprime, finer_grid)

    Q = spzeros(N * T, N * T)

    for j in 1:T
        st = (j - 1) * N + 1
        en = j * N

        Q[st:en, :] = Q_a[st:en, :] ⊗ Γ[j, :]`
    end

    droptol!(Q, 1e-10)

    return Q, (N, T), finer_grid
```

```
end
```

The resulting stationary distributions according to the density approximation method is presented in Figure 2:

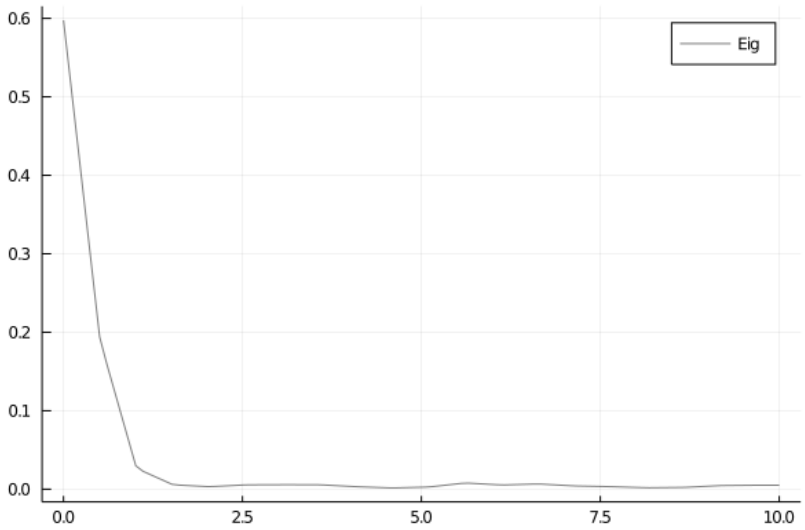

Figure 2: Staionary distribution of a according to the density approximation method.

Question 2

We solve the general equilibrium model in week-four/general.jl. We proceed by extracting the parameters and the required functions from the model and by setting up a support for the equilibrium interest rate, as suggested in the slides. We then compute $K(r_0)$ and $w(r_0)$ using the first guess of r. Moving forward, we define the function A which calls the partial equilibrium solvers we described in the previous point to derive the stationary distribution of a given r, and computes the aggregate demand for $A(r)$. As a last step, solve_general finds the equilibrium r that solves the equation $K(r) = A(r)$.

```
function solve_general(
    model::Aiyagari; verbose=false, n_steps=200)

    @unpack δ, β, a_ = model

    bounds = [ε - δ, 1 / β - (1 + ε)]
```

```
8
9       F, F_k, F_l, invF_k = make_F(model)
10
11      K = (r) -> invF_k(r + δ)
12      w = (r) -> F_l(K(r))
13
14      function A(r::Float64)
15          λ, a′, a_grid = solvepartial(
16              model, r, w(r),
17              verbose=verbose, n_steps=n_steps, mc=false, end_grid=true)
18
19          return quadgk(a -> a * λ(a), model.a_, a_grid[end])[1]
20      end
21
22      clearprices(r) = A(r) - K(r)
23
24      @time interest = find_zero(clearprices, mean(bounds), verbose=
            verbose)
25
26      return interest
27  end
```

We plot $A(r)$ and $K(r)$ and the resulting equilibrium in the figure below:

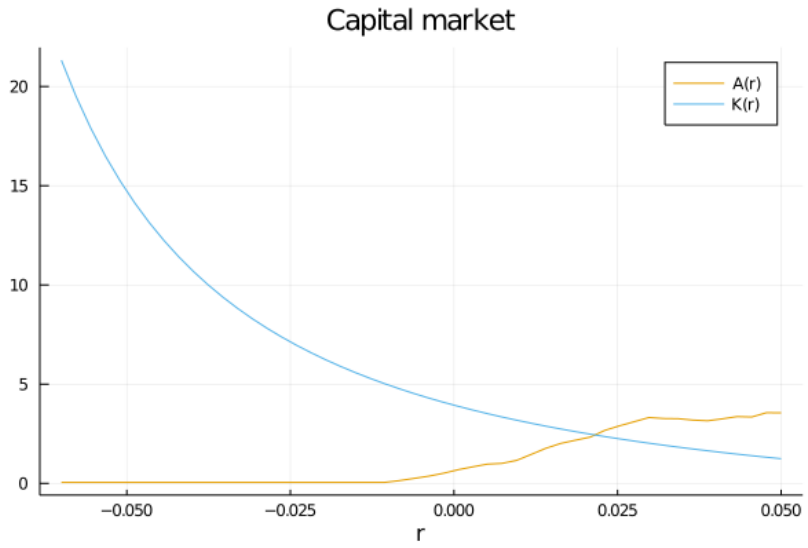

Figure 3: General equilibrium, given r.

According to our solution, equilibrium r and w are respectively 0.02 and 0.9, approximately.