Advanced Topics in Macro 1: Assignment 1

Due on 3 November 2020 12:00 am

Andrea Titton, Alessandro Zona Mattioli

Problem 1

We start the problem by defining a set of tools that will be repeatedly used in all problems. In the **utils** folder we store a set of functions and types, which are then called to compute maximization, interpolation and other standard operations. We then store the main components of the deterministic growth model in the **model** folder. Here we source the functions used to compute dynamics and equilibria of the specific model at hand.
We handle the value function iteration in the vfi.jl file. We first compute a grid of (k,k') and compute the first step of the iteration as:

```
k_star = equil_k(model)
k = collect(range(0.01, k_star + 1, length=model.k_size))
k_grid = collect(Iterators.product(k, k))

V_0 = utility(k_star, k_star) / (1 - model.) * ones(model.k_size)
u_matrix = utility.(k_grid)

V_i = copy(V_0)
policy_vec = -1 * ones(model.k_size)
```

At this point the algorithm takes a different iteration function according to our choice of the property we want to exploit. Indeed, in interalgos.jl are stored the different iteration functions that will thereafter be selected, where the naive_iteration function takes the basic iteration and the one exploiting concavity, using as maximizers the functions stored in the utils folder:

```
maximizer = concave ? concave_max : naive_max

function iterate(V_i::RealArray)::RealArray
    V_iter = copy(V_i)

    H = u_matrix .+ model. * V_i'

    for (k_idx, h_row) in enumerate(eachrow(H))

        k_prime, v_max = maximizer(h_row)


        V_iter[k_idx] = v_max
        policy_vec[k_idx] = k_prime
    end

    return V_iter
end
```

and the monotone_iteration function will in turn implement the iteration exploiting monotonicity:

```
function iterate(V_i::RealArray)::RealArray
    V_iter = copy(V_i)

    prev = 1
```

```
        H = u_matrix .+ model. * V_i'

        for (k_idx, h_row) in enumerate(eachrow(H))

            k_prime, v_max = maximizer(h_row[prev:end])

            V_iter[k_idx] = v_max

            prev = asint(k_prime + prev - 1)

            policy_vec[k_idx] = prev
        end

        return V_iter
    end
```

Finally, the Howard policy iteration is stored within the solve_value_function itself under an if condition. It follows the approach described in Heer Maussner (2011):

```
        if howard

        u = [u_matrix[i, asint(policy_vec[i])] for i in 1:model.k_size]

        Q = zeros((model.k_size, model.k_size))

        for (i, j) in enumerate(policy_vec)
            Q[i, asint(j)] = 1.
        end

        V_iter = inv(I - model. * Q) * u
```

This approach is quite flexible, as the different functions can be implemented in combination with each other in an easy way. We can see noticeable improvement in performance using both the monotonicity and concavity algorithms, while Howard requires an allocation of memory that given our parameters does not justify the gain given by faster computations.
The policy function and value function we obtain as a result of the iterations are reported in Figure 1 and Figure 2, respectively.

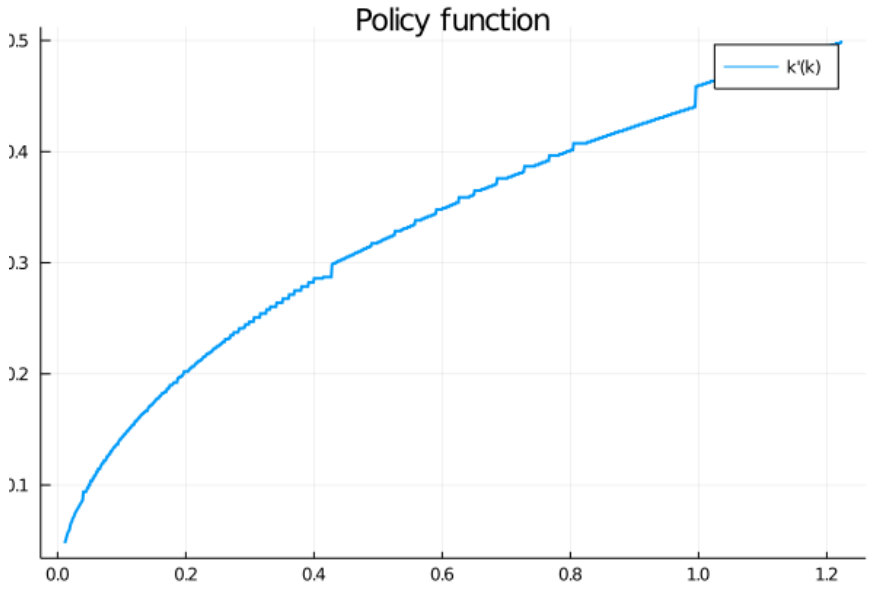


Figure 1: Policy function.

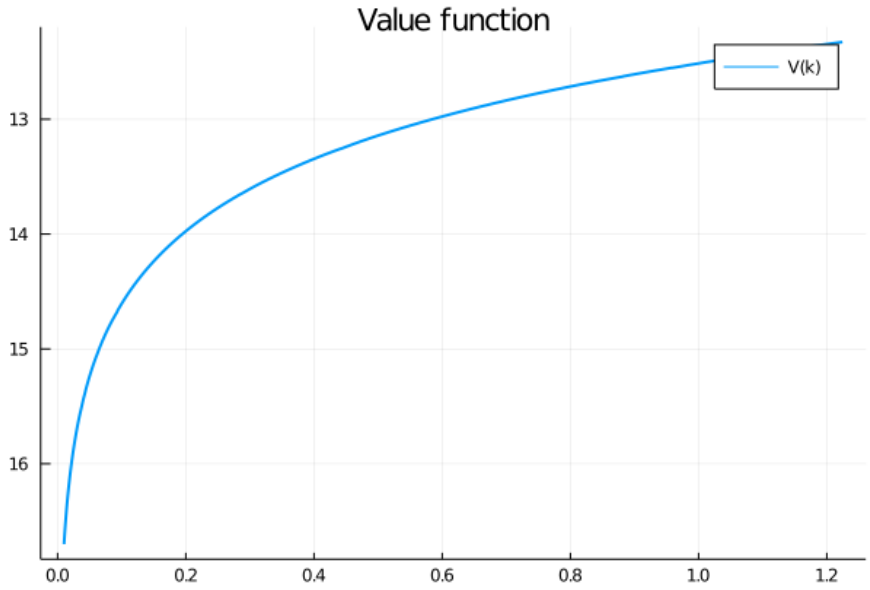


Figure 2: Value function function.

Problem 2

We start by implementing the shooting algorithm first. This is coded in the transition.jl file, through the function shooting. The function takes the start and end point, together with the function regulating the transition and a series of additional parameters. We use as function for the dynamics the Euler equation of slide 37/68. Starting from the initial level for capital, the function will then guess a new value for the next period and iterate forward the policy function for 200 periods. If at the end of the process the distance with

the end point is sufficiently small, the algorithm will stop, otherwise it will guess another second step and start again:

```
k_path = zeros(steps + 1)
k_path[1] = k_start

k_grid = range(max(0, k_start - pert), k_start + pert, length=size_grid)
print("Searching from $k_start to $k_end, around $k_grid\n")

for k_t1 in k_grid

    k_path[2] = k_t1

    for i in 3:length(k_path)

        k_t2 = fn_k2(k_path[i - 2], k_path[i - 1])

        if k_t2 < 0 break end

        k_path[i] = k_t2
    end

    if abs(k_path[end] - k_end) <
        return k_path
    end
```

The shooting function is then actually implemented by the compute_transition function, which is designed to also incorporate the the inverse shooting method, i.e. guessing capital in the period 200 until, by moving backwards, we find the right initial capital.
IN carrying on this analysis we need to bear in mind the following caveat. After the productivity shock, in our model it is not given whether the system will have a unique new steady state, as we might have bifurcations depending on the parameter values we choose for the model. In our case, for instance, we try with $\alpha = 0.5$ and $\beta = 0.9$ and after the productivity shock ($z = 2$), the model has two equilibria, one locally stable at approximately $k' = 4$ and one unstable at $k' = 0.8$. The system will then immediately converge to the second equilibrium in case the shock pushes capital instantly at the new equilibrium level, but will converge to the first equilibrium or diverge to 0 otherwise. Simply defining the end point as a solution to $k' = \alpha\beta z k^{\alpha}$ is therefore not always correct and a more complex analysis is required.
The resulting transition path is reported in Figure 3. We can see that the transition takes actually less than 200 periods and capital quickly converges to the new higher steady state.

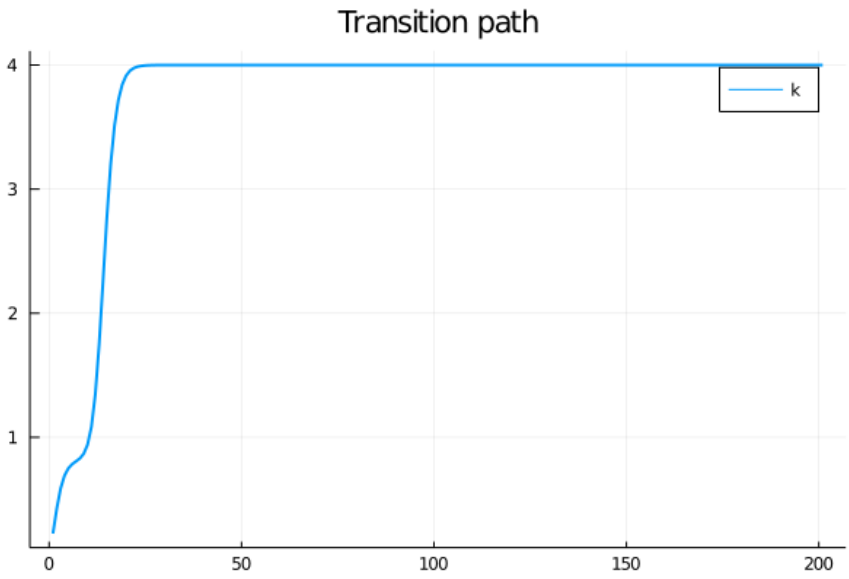


Figure 3: Transition path between two equilibria after productivity shock, according to shooting algorithm.

We give an attempt to solving the system of 200 equations in the system function in transition.jl, but at the moment the solver algorithm cannot find a realistic path for capital (see Figure 4). We also try the backward shooting algorithm in the compute_transition function, exploiting the same shooting algorithm but with an inverse function for the transition dynamics. Unfortunately, in this case the algorithm cannot reach the right starting point for capital, but diverges to 0 (see Figure 5).

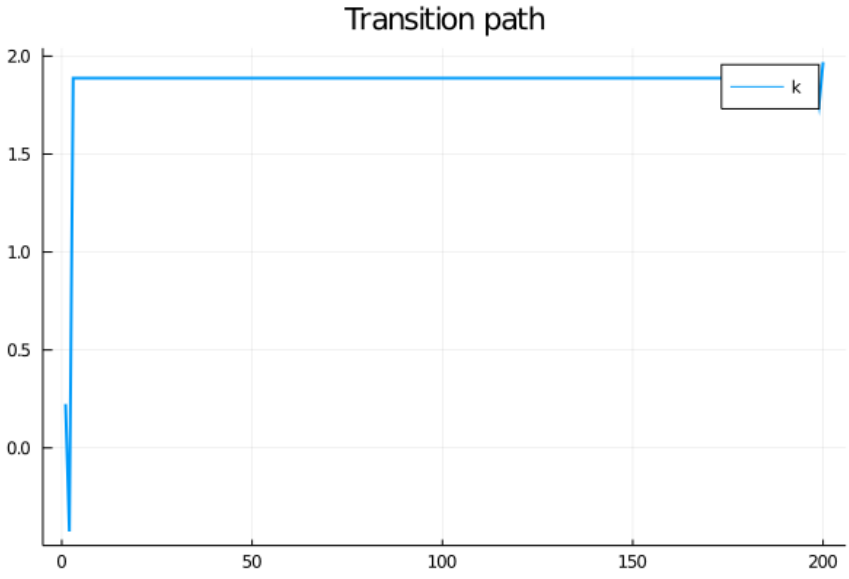
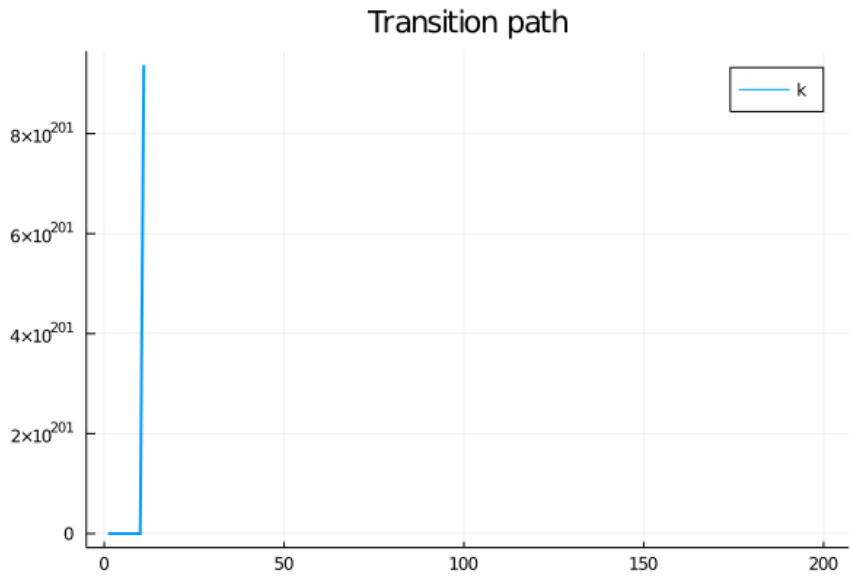


Figure 4: Transition path between two equilibria after productivity shock according to system algorithm.



shooting.png

Figure 5: Transition path between two equilibria after productivity shock according to inverse shooting algorithm.

Problem 4

Let $c_n = \frac{b_n - a_n}{2}$ be the middle point in the i-th iteration of the bisection method. Now consider the absolute error. By construction in needs to be bounded above by the i-th splitting of the interval, hence,

$$|c_n - c| \leq \frac{|b - a|}{2^n} \tag{1}$$

where c is the root of the function f. It is straight forward to see then that $|c_n - c| \to 0$ linearly at rate 0.5.