Advanced Topics Macro I - Assignment 2

Andrea Titton & Alessandro Zona Mattioli

November 2020

1 Question 1

We are given the $AR(1)$ process:

$$z_{t+1} = \rho z_t + (1 - \rho^2)\varepsilon_{t+1} \tag{1}$$

Given $|\rho| < 1$, and knowing $Cov(z_t, \varepsilon_t) = 0 \ \forall t$, we can derive the asymptotic variance as:

$$\begin{aligned} V(z) &= \rho^2 \cdot V(z) + (1 - \rho^2)^2 \cdot V(\varepsilon) \\ V(z) &= \frac{(1 - \rho^2)^2 \cdot \sigma_\varepsilon^2}{1 - \rho^2} \\ V(z) &= (1 - \rho^2) \cdot \sigma_\varepsilon^2 \end{aligned} \tag{2}$$

We then turn to coding. The main file that executes all required scripts is run-two.jl. To execute the first exercise we rely on the function run-markov, stored in the ar-markov.jl script. The function first defines a Process with the same moments and dynamics as the one we are working with and then calls either the function tauchen or rouwenhorst depending on the option specified. These functions are responsible for computing the discrete transition matrix given a process and the parameter N, and are stored in the markov folder in week-two.

In the tauchen case, we compute the transition matrix using the tauchen function:

```
1
2  function tauchen(proc::Process, N::Int; m::Int=3)
3      F = cdf(proc)
4
5      Ψ = m * √(var(proc))
6
7      partition = makepartition(Ψ, N)
8      d = distance(partition)
9
10
11     f(z_row) = z_col -> F((z_row + d - evol(proc, z_col)) / σ_ϵ(proc))
12     b(z_row) = z_col -> F((z_row - d - evol(proc, z_col)) / σ_ϵ(proc))
13
14     P = zeros((N, N))
15
16     P[:, 1] = f(partition[1]).(partition)
17
18     for i in 2:(N - 1)
19         P[:, i] = f(partition[i]).(partition) - b(partition[i]).(
               partition)
```

```
    end

    P[:, N] = 1 .- b(partition[N]).(partition)

    return P, partition
end
```

Notice that we define the Partition structure in such a way that $\tilde{z}_i = -\infty$ for the points below the minimum $\tilde{z}_i$ and $+\infty$ above. This simplifies formulas substantially in the computation of the elements of P. The tauchen function in the ends returns the transition matrix and the grid points.

Moving on to the Rouwehorst method, this is executed in the rouwenhorst.jl file. In this case we first define a P_N function responsible of constructing the P_N Markov chain recursively, given the $p.q, \psi$ parameters.

```
function P_N(N)
    if N == 2
        return (p, q) -> [p (1 - p); (1 - q) q]
    end

    P_p = P_N(N - 1)

    function nested_P(p, q)
        Ψ = P_p(p, q)
        n, n = size(Ψ)

        Ω = zeros((n + 1, n + 1))

        Ω[1:n, 1:n] += p * Ψ
        Ω[1:n, end - n + 1:end] += (1 - p) * Ψ
        Ω[end - n + 1:end, 1:n] += (1 - q) * Ψ
        Ω[end - n + 1:end, end - n + 1:end] += q * Ψ

        return Ω
    end


    return nested_P
end
```

Once this is defined, we allow for two different ways to solve the estimation via method of moments. One is analytical and linked to our particular problem, (rouwenhorst_analy function). The other is numerical and is more general (rouwenhorst_numerical). We end up using the analytical one for this problem as the numerical function is less stable and very sensible to the specification of N.

```
function rouwenhorst_analy(proc::Process, N::Int)

    Ez = mean(proc)
    Vz = var(proc)
```

```
    corr = autocor(proc)

    q0 = 0.8 * rand(Uniform(.9, 1.1)) # Look around a sensible q value

    if Ez == 0
        # If E[z] = 0, we can find an analytical solution
        Ψ, p, q = zero_process(Vz, corr, N)
    else
        N_scale = -1 + 1 / (N - 1)

        p(q) = corr + 1 - q
        Ψ(q) = Ez * (2 - p(q) - q) / (q - p(q))
        s(q) = (1 - p(q)) / (2 - p(q) - q)

        four_s(q) = 4 * s(q) * (1 - s(q))

        function f(q)
            return Ψ(q)^2 * (1 + four_s(q) * N_scale) - Vz
        end

        q = find_zero(f, q0)
    end

    S = ispositive(proc) ? makepositivepartition(Ψ(q), N) :
        makepartition(Ψ(q), N)

    return colnormalized(P_N(N)(p(q), q)), S
end
```

where,

```
"""
Solves the analytical rouwenhorst
process for the special case E[z] = 0
"""
function zero_process(Vz::Float64, corr::Float64, N::Int)
    q = (corr + 1) / 2

    # For p = q, s = 0.5, and 4s(1-s) = 1.
    # Write Ψ and p as a functional for consistency
    Ψ = (_) -> √(Vz * (N - 1))
    p = (_) -> q

    return Ψ, p, q

end
```

Finally, we use the mutable structure MarkovDiscrete we define in markov/discretization in the simulation.jl file to compute summary statistics and simulate $T = 2000$ periods, throwing away the first 500.

The results of the various simulations are reported below. We can see that the Tauchen method works fine for a level of ρ that is not close to 1 and for $N = 5$. On the other hand, the Rouwenhorst method, while being computationally more intensive, better performs with ρ close to 1, but also with higher N.

```

Summary for ρ = 0.7 and N = 5, with method = rouwenhorst:
μ: 0.003833415954696192
ν: 0.49694004021737326
ρ: 0.6915756224182722

Summary for ρ = 0.7 and N = 100, with method = rouwenhorst:
μ: -0.007909423944909389
ν: 0.49900591927618804
ρ: 0.6943900235840942

Summary for ρ = 0.99 and N = 5, with method = rouwenhorst:
μ: 0.026161529098169982
ν: 0.016714386560971444
ρ: 0.9883404293417892

Summary for ρ = 0.99 and N = 100, with method = rouwenhorst:
μ: -0.02464141277194133
ν: 0.019649074872591016
ρ: 0.9895114290269361

Summary for ρ = 0.7 and N = 5, with method = tauchen:
μ: -0.01352970336951598
ν: 0.405911138384325
ρ: 0.7217662525522333

Summary for ρ = 0.7 and N = 100, with method = tauchen:
μ: 0.0043618488135729145
ν: 0.2619266443035376
ρ: 0.6983377837667868

Summary for ρ = 0.99 and N = 5, with method = tauchen:
μ: 0.423202079389977
ν: 4.93089964506897e-32
ρ: 0.9998947479212714

Summary for ρ = 0.99 and N = 100, with method = tauchen:
μ: -0.002061121397838489
ν: 0.0005594971435910456
ρ: 0.9911091901971842
```

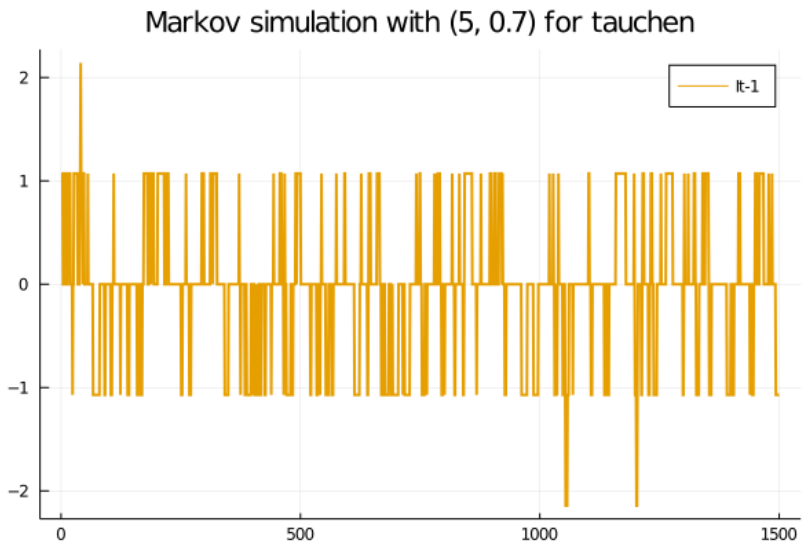

Figure 1: Simulation of z_t with Tauchen method, $N = 5$, $\rho = 0.7$

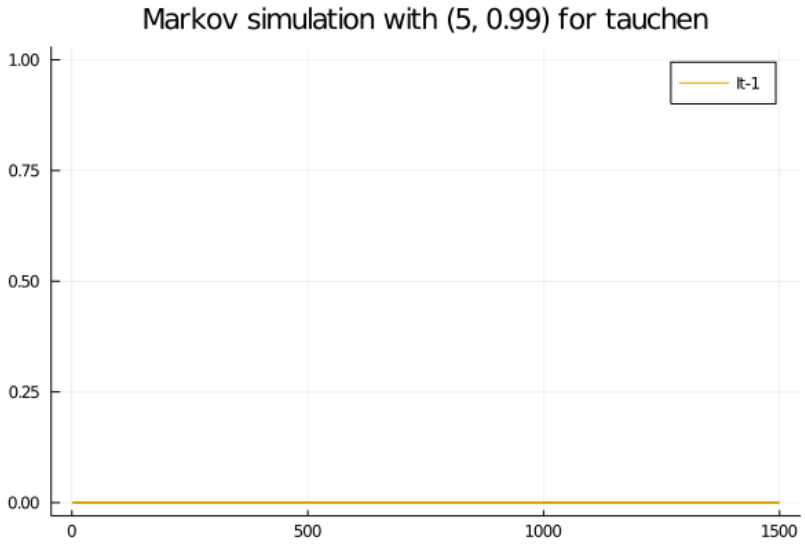

Figure 2: Simulation of z_t with Tauchen method, $N = 5$, $\rho = 0.99$

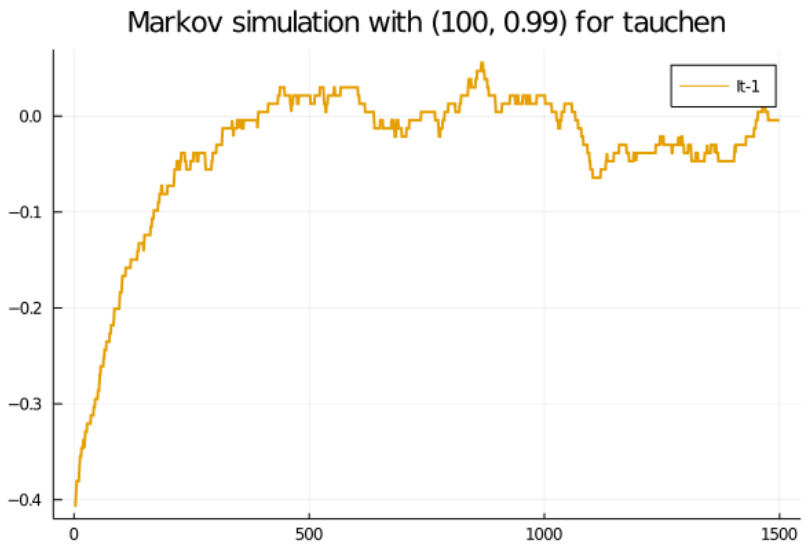

Figure 3: Simulation of z_t with Tauchen method, $N = 100$, $\rho = 0.99$

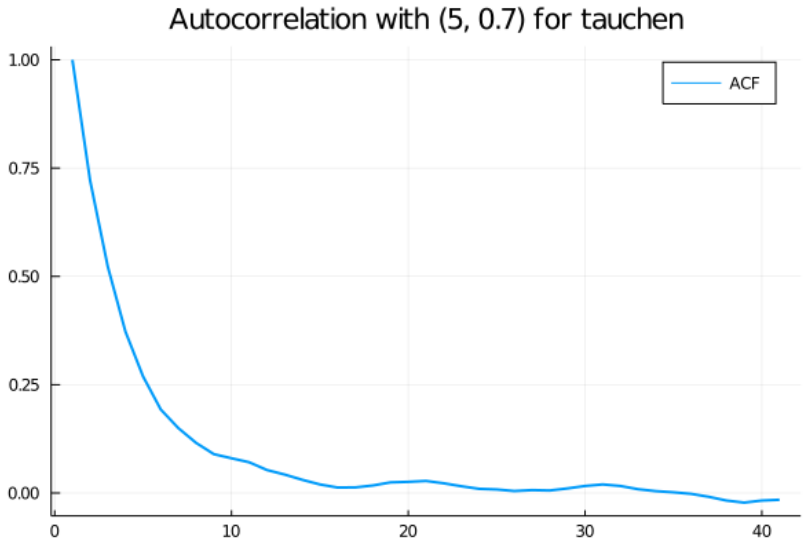

Figure 4: Autocorrelation of z_t with Tauchen method, $N = 5$, $\rho = 0.7$

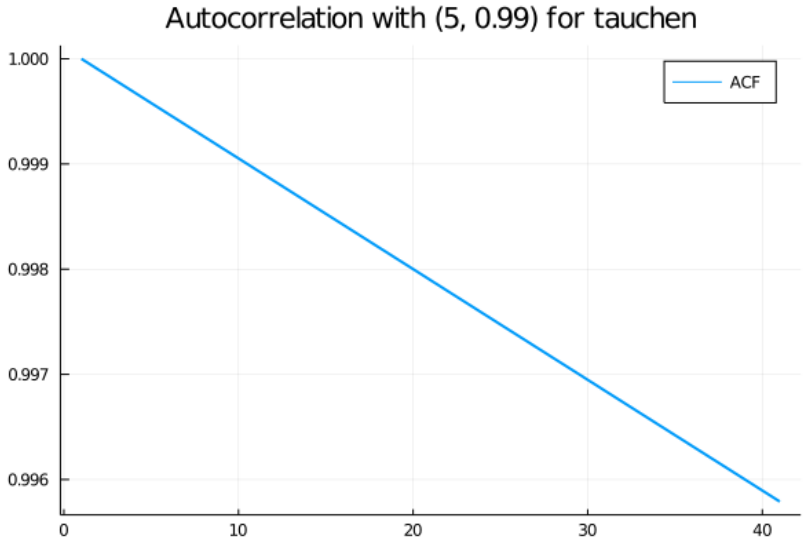

Figure 5: Autocorrelation of z_t with Tauchen method, $N = 5$, $\rho = 0.99$

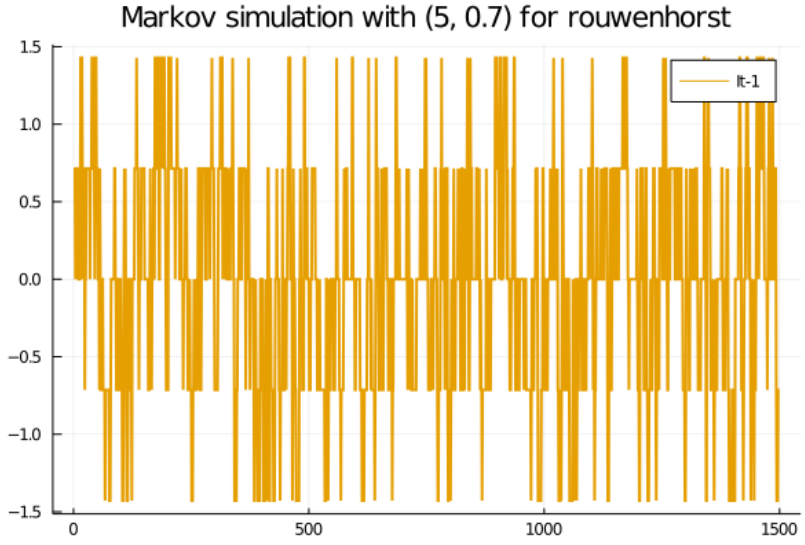

Figure 6: Simulation of z_t with Rouwenhorst method, $N = 5$, $\rho = 0.7$

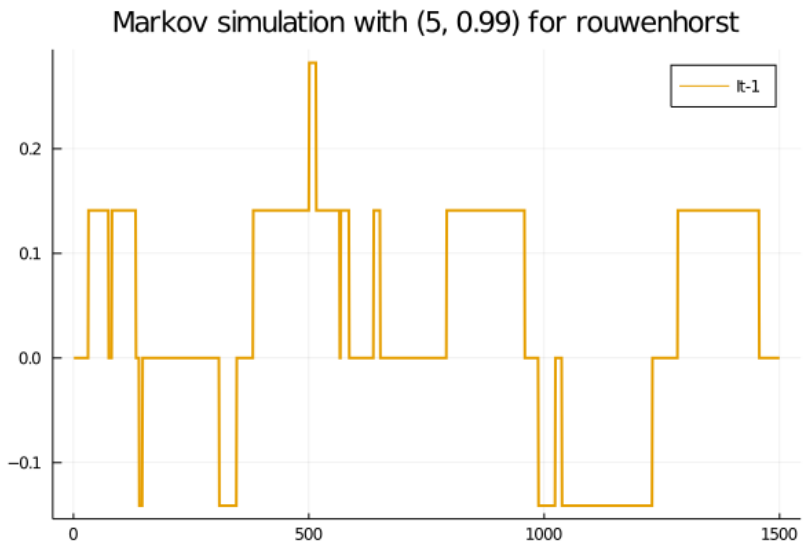

Figure 7: Simulation of z_t with Rouwenhorst method, $N = 5$, $\rho = 0.99$

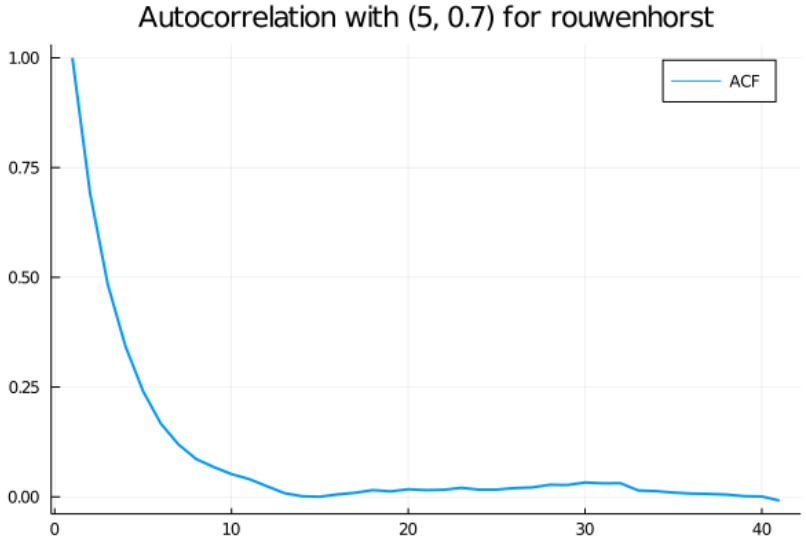

Figure 8: Autocorrelation of z_t with Rouwenhorst method, $N = 5$, $\rho = 0.7$

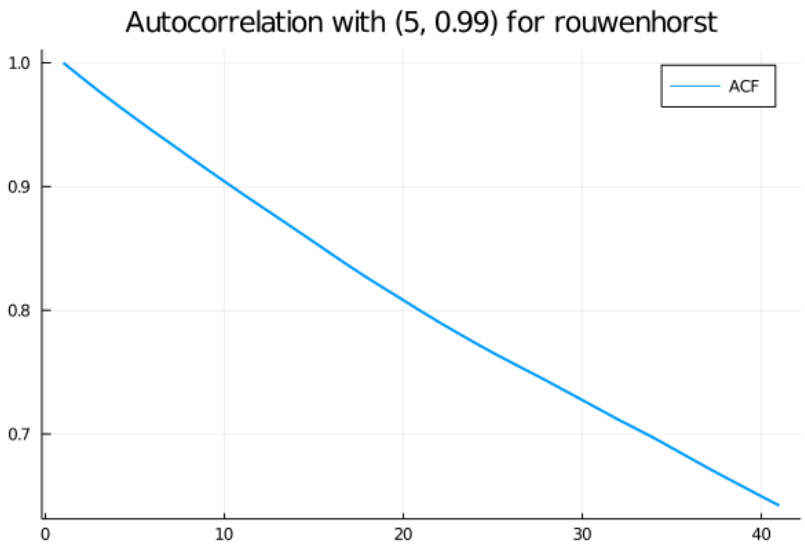

Figure 9: Autocorrelation of z_t with Rouwenhorst method, $N = 5$, $\rho = 0.99$

2 Question 2

To tackle this problem we use the run_solver function of the solvestochgrowth.jl file. We first discretize the Markov process using the Rouwenhorst method with the function make_z_proc, which discretizes process y_t and then takes the exponential.

```
function make_z_proc(μ::Float64, σ::Float64, N::Int, ρ::Float64)::
    MarkovDiscrete

    μ_y, σ_y = 0., σ / √(1 - ρ^2)

    z = Process(
        Normal(μ_y, σ_y),
        ρ,
        Normal(μ, σ)
    )

    P, S = rouwenhorst(z, N; numerical=false)
    markov = MarkovDiscrete(P, S, exp)

    return markov
end
```

With this discretized process in hands we can then move to implementing value

function and policy function iteration. We first define the Stochastic Growth Model process and then we move to retrieving the value and policy functions in valuesolve and policysolve, both stored in the ogm folder.

policysolve defines a grid for k_t and the support for z_t and then computes the Euler equation:

```
function euler_diff(k::Float64, prod::Float64, k_prime::Float64)::
    Float64
        vals = 1. .+ f_prime.(k, support_z)
        p_z = p_cond(model.z, prod)

        return inv_u_c(model.β * E(vals, p_z) * u_c(f(k, prod) -
            k_prime))
end
```

Then, for each element of the support of z, the function solves the Euler equation for each (k, k'). Once a solution has been found, it computes the Euler equation error.

```

    policy_k_matrix = zeros(grid_N, length(support_z))
    EEE = zeros(grid_N, length(support_z))

    for (h, z) in enumerate(support_z)

        policy_k = ones(grid_N)
        euler_error = ones(grid_N) * Inf

        for i in 1:max_iter
            current_policy = copy(k_space.steps)

            for (j, k_row) in enumerate(k_space)
                k_prime = current_policy[j]

                k_double_prime = current_policy[get_k(k_prime)]

                opt_c = euler_diff(k_prime, z, k_double_prime)

                opt_k = f(k_row, z) - opt_c
                current_policy[j] = opt_k

            end

            distance = maximum(abs.(policy_k - current_policy))

            if distance < tol
                if verbose print("Found policy in i iterations (|x - x
                    '| = distance)") end

                k_double_prime = current_policy[get_k.(current_policy)
                    ]
```

```
31
32              euler_eq = euler_diff.(current_policy, z,
                    k_double_prime)
33              current_c = f.(k_space, z) .- current_policy
34
35              euler_error = log10.(abs.(1. .- (euler_eq ./ current_c
                    )))
36
37              break
38          end
39
40          policy_k = current_policy
41      end
42
43      policy_k_matrix[:, h] = policy_k
44      EEE[:, h] = euler_error
45
46  end
47
48  return policy_k_matrix, EEE
```

A similar procedure is implemented for the value function iteration. We first construct a grid for (k, k') which has an additional dimension given by the support of z. This is convenient as the term $E[V(z', k')]$ in the Bellman equation can then be defined as $V_i \cdot P'$, where P transition matrix of the discretized Markov process, and V_i contains the row maximums of the (k, k') grid for each element of the support of z (in our case it is therefore a N_k x N_z matrix). We repeat the value function iteration until convergence and then compute the Euler equation error, as for the policy function iteration case:

```
1
2   support_z = support(model.z)
3   k_space = Partition(collect(range(1e-5, 5, length=grid_N)))
4
5   kz_grid = collect(Iterators.product(k_space, k_space, support_z))
6
7   V_i = ones(grid_N, length(support_z))
8
9   utility = u_k.(kz_grid)
10
11  get_k = (k::Real) -> get_closest(k_space, k)
12
13  function euler_diff(k::Float64, prod::Float64, k_prime::Float64)
        ::Float64
14
15      vals = 1. .+ f_prime.(k, support_z)
16      p_z = p_cond(model.z, prod)
17
18      return inv_u_c(model.β * E(vals, p_z) * u_c(f(k, prod) -
            k_prime))
19  end
```

```
20
21      H = zeros(size(utility))
22
23      for i in 1:max_iter
24
25          if verbose print("Iteration i / max_iter \r") end
26
27          EV = V_i * model.z.P'
28
29          for h in 1:length(support_z)
30              H[:, :, h] = utility[:, :, h] .+ model.β * EV[:, h]
31          end
32
33          values, argmax = findmax(H, dims=2)
34
35          next_V = reshape(values, size(V_i))
36
37          distance = matrix_distance(V_i, next_V)
38
39          if distance < tol
40              if verbose print("Found policy in i iterations (|x - x'| =
                    distance)") end
41
42              current_policy = reshape(kz_grid[argmax], size(V_i))
43
44              euler_error = zeros(size(V_i))
45
46              for (h, ζ) in enumerate(support_z)
47                  k_p = [tup[1] for tup in current_policy[:, h]]
48                  k_double_prime = k_p[get_k.(k_p)]
49
50                  euler_eq = euler_diff.(k_p, ζ, k_double_prime)
51                  current_c = f.(k_space, ζ) .- k_p
52
53                  euler_error[:, h] = log10.(abs.(1. .- (euler_eq ./
                        current_c)))
54              end
55
56
57              return V_i, euler_error
58          end
59
60          V_i = next_V
61
62      end
```

Once we retrieve the policy function and value function and the corresponding EEE, we can plot our results. We plot our results for the Euler equation error in Figures 14, 15, 16. We plot the EEE for the policy function, the value function and then we plot the difference in their absolute value. This allow to compare the two in terms of performance: As long as the line for the difference is above

0, this implies that the policy error is lower, while the opposite is true as soon as the line goes below 0. We can then infer, for instance, that for $z = 0.14$ (first case), the policy error is lower only for very low values of k, while for the rest of the k support the opposite is true. We can notice that computations produce an unusual discontinuity at $k = 1$ if $z = 1$. This is due to the fact that for these values $u_c(c(k)) = 1/0$

In terms of speed, the policy function takes much less time compared to the value function. This is probably due to the more complex computations required in the value function iteration, even if some gains in speed could be achieved there by exploiting monotonicity and concavity.

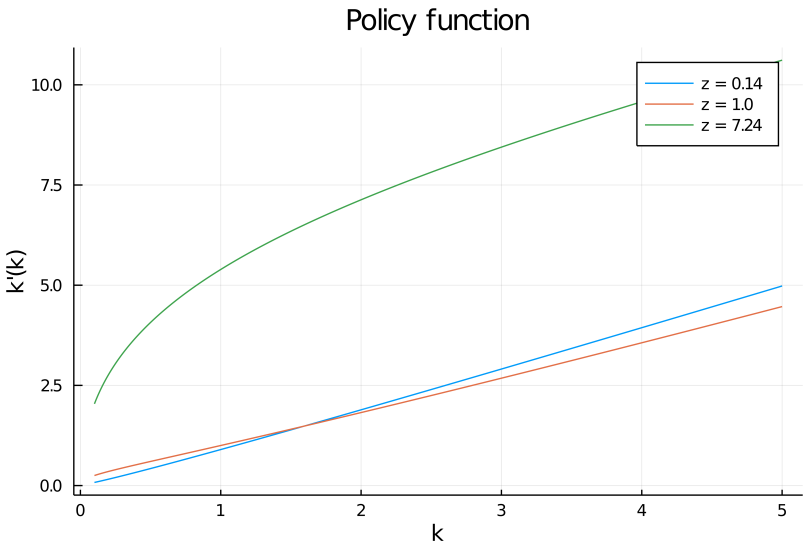

Figure 10: Policy function for $k'(k)$ at different z values.

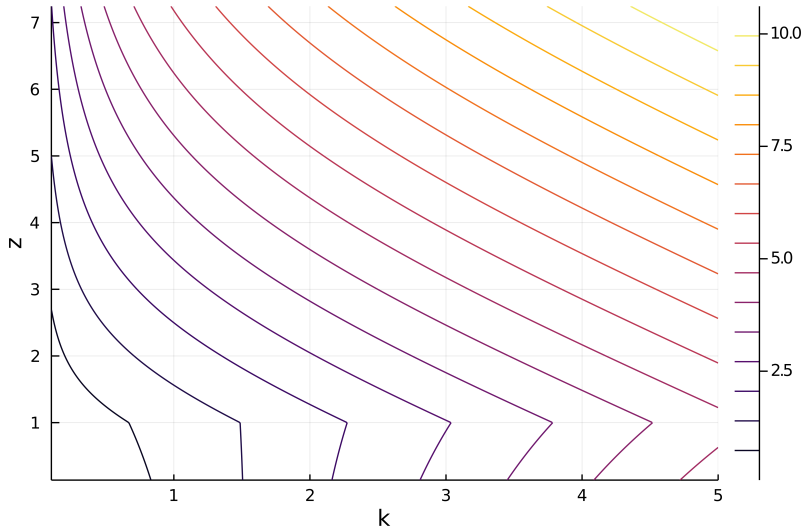

Figure 11: Policy function for $k'(k)$ at different z and k values.

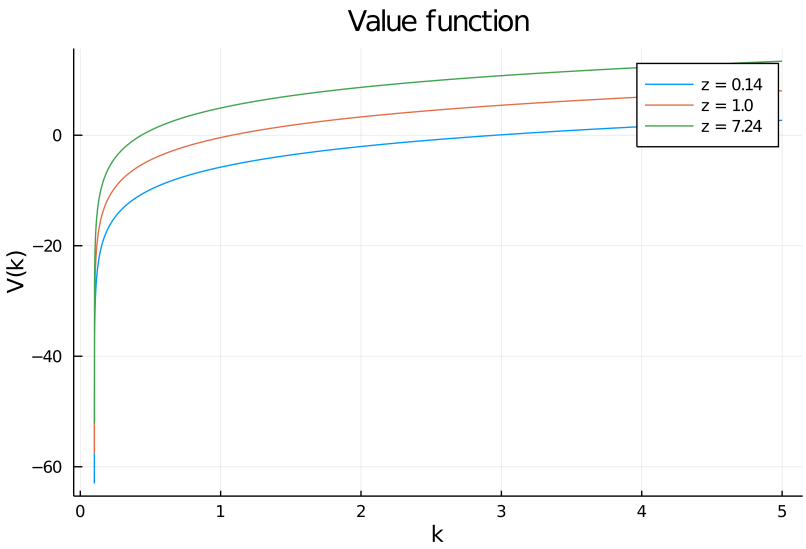

Figure 12: Value function for $k'(k)$ at different z values.

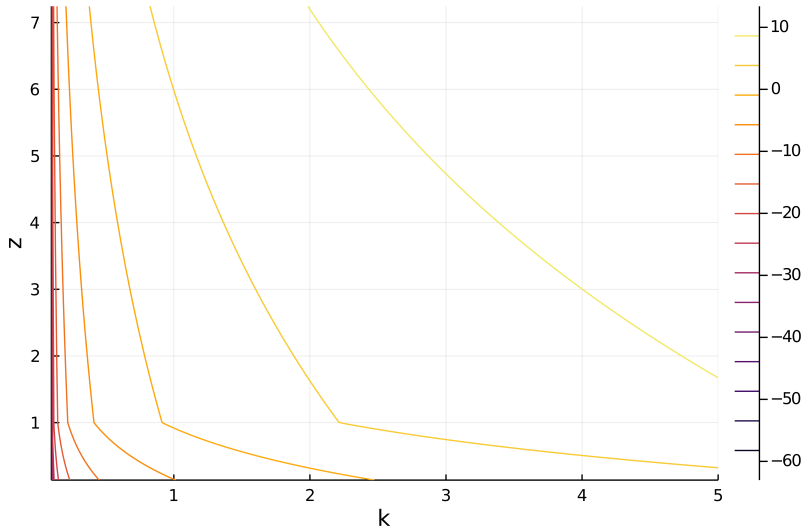

Figure 13: Value function for $k'(k)$ at different z and k values.

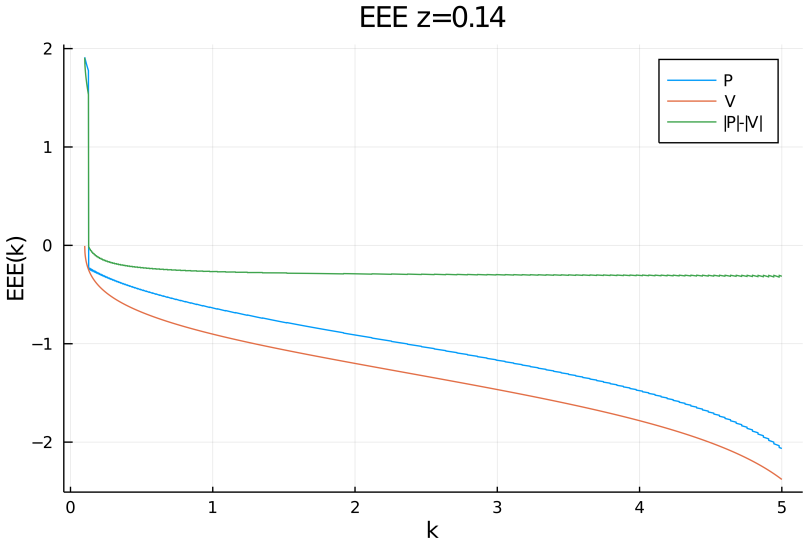

Figure 14: Euler equation error between policy and value function for the first value of z_t.

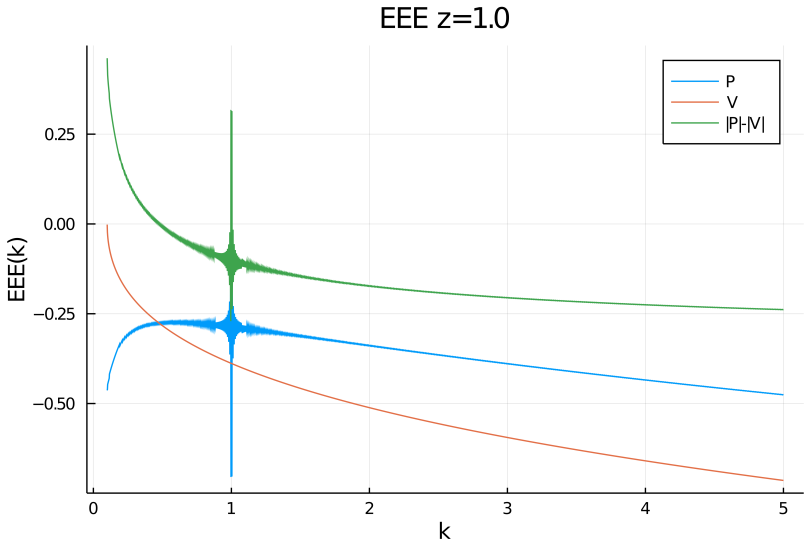

Figure 15: Euler equation error between policy and value function for the second value of z_t.

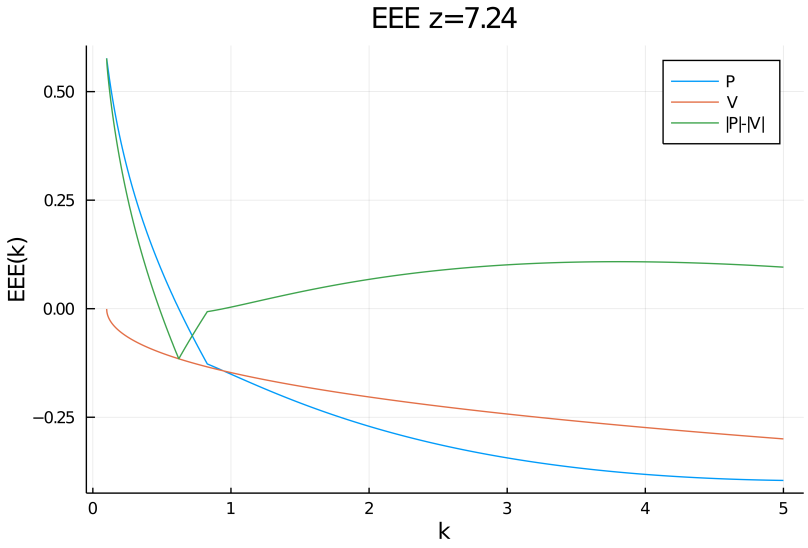

Figure 16: Euler equation error between policy and value function for the third value of z_t.