Numerical Methods **Problem Set 2**
Professor: Cezar Santos **February 27, 2023**
TA: Rafael Vetromille **Student:** Luan Borelli

Preliminary comments. For all results presented in the next pages I considered a tolerance of 10^{-5} and a maximum number of 10000 iterations (that was never reached, though). For parallelization, I considered 8 threads on an Intel(R) Core(TM) i7-9700 CPU @ 3.00GHz processor, with 16 GB of RAM. To facilitate the replication of results, the attached codes were separated into several files, one for each method and specification considered in the exercises.

---

1. *Write the planner's problem in recursive form.*

Solution. Since agents do not derive utility from leisure (i.e. disutility from work), we know that labor supply is perfectly inelastic so, without loss of generality, we can impose $N_t = 1$ for all t. In this way, the capital per unit of labor $k_t \equiv K_t/N_t$ becomes exactly equal to K_t. Also, the product per unit of labor, $y_t \equiv Y_t/N_t = z_t k_t^\alpha$, becomes exactly equal to Y_t. The planner's problem in sequential formulation can then be stated as

$$\max_{\{c_t\}_{t=0}^\infty, \{k_t\}_{t=0}^\infty} \mathbb{E}_0 \sum_{t=0}^\infty \beta^t u(c_t) \quad \text{subject to } c_t + k_{t+1} = z_t y_t + (1-\delta)k_t \quad \forall t,$$

$$\text{or} \quad \max_{\{c_t\}_{t=0}^\infty, \{k_t\}_{t=0}^\infty} \mathbb{E}_0 \sum_{t=0}^\infty \beta^t \frac{c_t^{1-\mu} - 1}{1-\mu} \quad \text{subject to } c_t + k_{t+1} = z_t k_t^\alpha + (1-\delta)k_t \quad \forall t.$$

Therefore, the recursive formulation can be stated as

$$V(k,z) = \max_{c,k'} \frac{c^{1-\mu}-1}{1-\mu} + \beta \mathbb{E}_{z'}[V(k',z')] \quad \text{subject to } c + k' = zk^\alpha + (1-\delta)k, \tag{1}$$

where $\mathbb{E}_{z'}[\cdot]$ indicates that the expectation is being taken with respect to z'.

It is (computationally) convenient to simplify this expression even further by replacing the constraint in the objective function, so that the problem becomes a single control problem:

$$V(k,z) = \max_{k'} \frac{[zk^\alpha + (1-\delta)k - k']^{1-\mu} - 1}{1-\mu} + \beta \mathbb{E}_{z'}[V(k',z')]. \tag{2}$$

□

---

2. *For now, assume no uncertainty, i.e., $\sigma = 0$. Derive the Euler equation and find the steady state capital stock k_{ss}.*

Solution. Since the variance of $\log z_t$ is given by $\sigma^2/(1-\rho^2)$, with $\sigma = 0$ the process of $\log z_t$ itself has zero variance and, since $\log z_t$ is mean-zero, it follows that we must have $\log z_t = 0$ for all t. Therefore we must have $z_t = 1$ for all t, or (in recursive notation) $z = z' = 1$ for all z. Denoting $V(k) \equiv V(k, 1)$, problem (2) becomes fully deterministic:

$$V(k) = \max_{k'} \frac{[k^\alpha + (1-\delta)k - k']^{1-\mu} - 1}{1-\mu} + \beta V(k'). \tag{3}$$

The first order condition for k' is given by

$$[k'] : \qquad -[k^\alpha + (1-\delta)k - k']^{-\mu} + \beta V_1(k') = 0 \tag{4}$$

By Benveniste-Scheinkman Theorem,

$$V_1(k') = [k'^\alpha + (1-\delta)k' - k'']^{-\mu}(\alpha k'^{\alpha-1} + 1 - \delta).$$

Plugging it back into (4) we obtain

$$-[zk^\alpha + (1-\delta)k - k']^{-\mu} + \beta[zk'^\alpha + (1-\delta)k' - k'']^{-\mu}(\alpha z k'^{\alpha-1} + 1 - \delta) = 0. \tag{5}$$

In the steady state we must have $k'' = k' = k$, so

$$-[k^\alpha+(1-\delta)k-k]^{-\mu}+\beta[k^\alpha+(1-\delta)k-k]^{-\mu}(\alpha k^{\alpha-1}+1-\delta) = 0 \implies -1+\beta(\alpha k^{\alpha-1}+1-\delta) = 0.$$

Solving for k finally yields

$$k_{ss} = \left(\frac{1}{\alpha\beta} - \frac{1-\delta}{\alpha}\right)^{\frac{1}{\alpha-1}} \tag{6}$$

$$= \left(\frac{1-\beta(1-\delta)}{\alpha\beta}\right)^{\frac{1}{\alpha-1}}. \tag{7}$$

□

3. *From now on, use the full model with uncertainty. Solve this problem in the computer using value function iteration (VFI). For this, you will need to discretize your state variables. For the TFP shock, use Tauchen's (1986) method with 7 grid points. For the grid for the capital stock, use 500 linearly spaced points in $[0.75k_{ss}, 1.25k_{ss}]$. I strongly recommend that you do not use the brute force grid search method to find the policy function. For this and the following exercises, provide evidence for your solution: figures for the value and/or policy function, running time, Euler errors, etc.*

Solution. In this exercise, I explore and compare results considering several different value function iteration (VFI) algorithms and specifications. Time performance results are reported in Table 1. The first two rows explore how time performance can be improved by simply adopting a "wise" initial guess for the value function, instead of a naive "zeros" guess. For the "wise" initial guess I consider Ben Moll's suggestion.[1] Notice that these first two rows do *not* consider parallelization. Parallelization starts to be considered only from the third row on (while keeping Moll's initial guess). The third row simply adds parallelization to the results of the second row, while from the fourth row on more sophisticated tricks began to be explored: vectorization, monotonicity, and concavity (while still keeping both Moll's initial guess *and* parallelization).[2] Monotonicity and concavity tricks closely follow what was explained in classes and in the course slides. Panels 1a to 1f in Figure 1 show the results for the value function and policy functions for capital and consumption, respectively. Panel 1g shows the Euler equation errors.[3]

Table 1: Time performance for different VFI algorithms and specifications.

Method			**Time**	**Iters**	**RP**
Brute force grid search (BFGS) VFI	*W/o parallelizing*	*Zeros' initial guess*	1933s	849	$1\times$
		Moll's initial guess	539s	235	$3.59\times$
	Parallelizing (with Moll's initial guess)		112s	235	$17.26\times$
BFGS VFI + Concavity + Parallelization + Moll's initial guess			109s	235	$17.73\times$
BFGS VFI + Monotonicity + Parallelization + Moll's initial guess			52.2s	235	$37.03\times$
Vectorized VFI + Parallelization + Moll's initial guess			4.78s	235	$404.39\times$
BFGS VFI + Concavity + Monotonicity + Parallelization + Moll's initial guess			0.94s	235	$2056.38\times$

Note: RP stands for "relative performance"; i.e., the relative time performance of each method with respect to the benchmark method (BFGS VFI without parallelizing, with zeros' initial guess).

As can be seen in Table 1, without parallelization and with a naive initial guess (the first row — our benchmark) the performance is very bad: the algorithm takes 849 iterations to converge and the code takes more than 30 minutes to solve the model. In the second row, it is possible to observe that by simply considering a "smarter" guess, there is a significant

[1]That consists of $V(k,z) = \dfrac{\tilde{c}(k,z)^{1-\mu} - 1}{(1-\mu)(1-\beta)}, \quad \forall (k,z) \in K_{\text{grid}} \times Z_{\text{grid}}$, where $\tilde{c}(k,z) \equiv zk^{\alpha} + (1-\delta)k - k$.

[2]For a brief explanation and discussion on vectorization, see Appendix A.

[3]Since the graphical results for all methods reported in Table 1 are numerically identical, I report these graphs only once.

performance gain: only 235 iterations are needed until convergence, and the time is reduced by about 3.6 times when compared to the benchmark. By including parallelization, the time is further reduced by an incredible 17.26 times over the benchmark, for the same number of 235 iterations.

From the fourth row onwards, more sophisticated tricks come into play. In the fourth row, the model is solved by exploiting (only) concavity, while maintaining Moll's initial guess and parallelization. It is possible to see that there is a slight but practically insignificant gain in time.[4] In the fifth row, we consider (only) the monotonicity and there is a considerable performance gain: the model is solved 37.03 times faster than the benchmark. The most relevant and even more impressive gain, however, lies in the seventh row: joining both concavity *and* monotonicity, the model is solved in an incredible time of 0.94 seconds (2056.38 times faster than the benchmark!), which shows that the monotonicity and concavity tricks offer enormous complementarity gains.

Finally, with vectorized VFI — sixth row — there is also a very expressive performance gain. The model is solved 404.39 times faster than the benchmark. It is interesting to note that this gain comes purely from computational efficiency gains (i.e., we are not exploiting any properties of the model structure itself to "cut corners" in maximization, as in the case of monotonicity and concavity; we are purely exploring computational aspects) — this highlights the power of vectorization.

Lastly, the Euler equation errors' statistics are reported in Table 2. Focusing on the mean, we can interpret that, on average, a \$1 mistake is made for every $\$10^{3.09} \approx \$1230$ spent. This result will be useful as a benchmark to compare the relative accuracy of other methods to be presented in the following exercises.

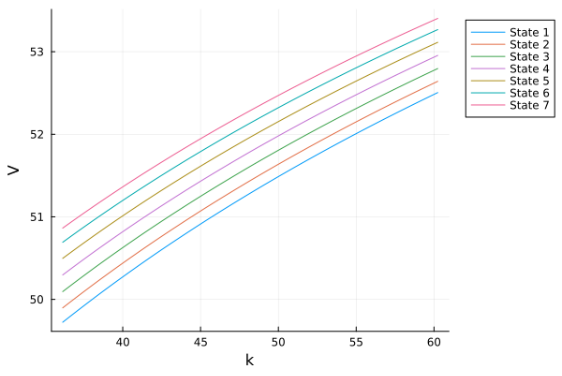
Table 2: Euler Equation Errors (EEEs) statistics — BFGS VFI.

Euler Equation Errors' Statistics	
Mean	-3.09447775759327
Median	-3.042207112577707
Maximum	-1.7476141199326618
Minimum	-6.132252761282904

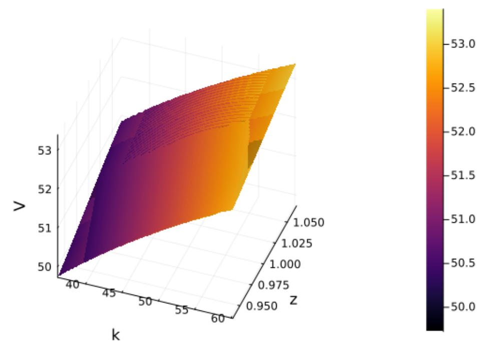
□

[4] After thinking for a while about the reasons for this very small performance gain, I came to the conclusion that the explanation lies in parallelization. It turns out that, at least the way I programmed concavity, the algorithm offers fewer "parallelization opportunities" than the basic BFGS VFI algorithm — it has one less for-loop, due to the lack of maximization. Thus, the gains from parallelization in the basic BFGS VFI algorithm end up "overshadowing" the performance gains of the concavity algorithm, which is less able to exploit parallelization. To verify this hypothesis, I ran concavity *without* parallelization: 499.82 seconds. Comparing this result with BFGS VFI *without* parallelization — 539 seconds — it is possible to observe that the relative performance of the concavity algorithm becomes more expressive, as expected.

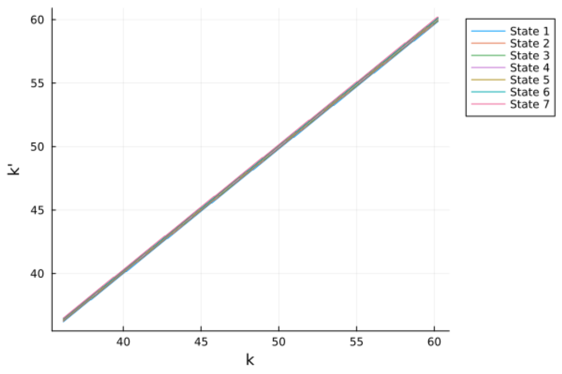
Figure 1: VFI results — Value function, policy functions and Euler Equation Errors.



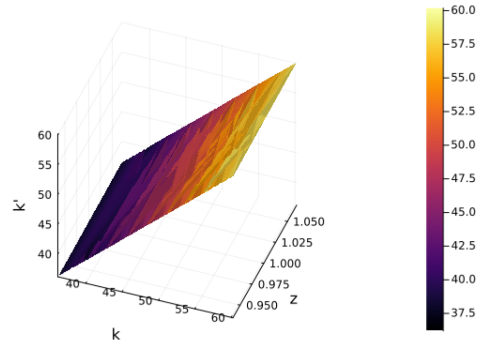
(a) Value function — 2D plot.



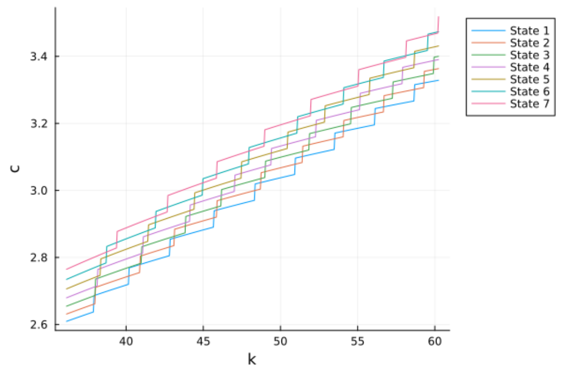
(b) Value function — 3D plot.



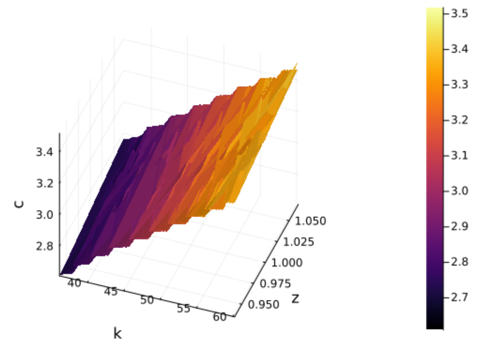
(c) Capital policy function — 2D plot.



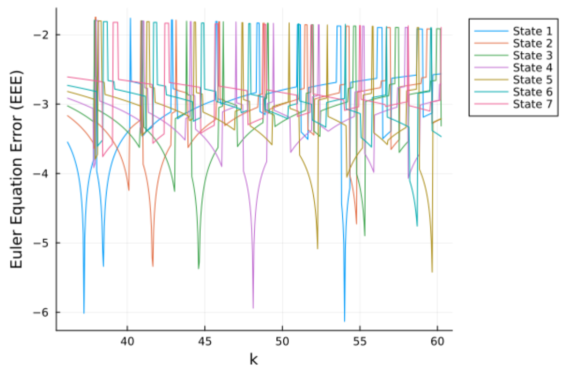
(d) Capital policy function — 3D plot.



(e) Consumption policy function — 2D plot.



(f) Consumption policy function — 3D plot.



(g) Euler Equation Errors.

4. *For this exercise, redo the previous item using the accelerator. That is, only perform the maximization part for a few iterations (10% of them, for example). Compare these results with the previous ones.*

Solution. The underlying idea of the accelerator method is simple: for each maximization in the basic VFI algorithm, we consider the approximate policy function just obtained after this particular maximization and iterate an additional n times the value function, but now considering this approximate policy function — i.e., for each maximization, we iterate n additional times *without* maximizing, using the just-obtained policy function. After these additional n iterations, we repeat the process: we perform one more maximization, obtain a new approximate policy function, and iterate the value function n more times. We continue with this until convergence.

Since the "additional" iterations will be very fast — as they do not involve costly brute force maximizations — it is to be expected that such a procedure will substantially speed up the convergence time of the original algorithm. It is important to note that the number n of "additional" iterations after each maximization must be set arbitrarily. I set $n = 10$, inspired by the exercise's suggestion, since setting $n = 10$ can be *approximately* understood as "only performing the maximization part for 10% of the iterations".[5] The results are reported in Table 3 and in the panels of Figure 2.

Table 3: Time performance for the VFI algorithm — Accelerator.

Method *(Parallelizing + Moll's initial guess)*	**Time**	**Iters**	**Acceleration**
BFGS VFI + Accelerator (10%)	9.98s	242	11.22×
Vectorized VFI + Accelerator (10%)	0.82s	242	5.82×
BFGS VFI + Accelerator (10%) + Concavity + Monotonicity	0.38s	242	2.47×

Note: "Acceleration" indicates how many times faster the result is in relation to its equivalent *without* considering the accelerator method.

As can be seen, the adoption of the acceleration method in the standard BFGS VFI method was able to accelerate convergence by 11.22 times; the time has been reduced from 112 seconds to just 9.98 seconds. For the vectorized case, the acceleration reached 5.82 times, reducing the time from the previous 4.78 seconds to just 0.82 seconds.

In the last row, we again consider the BFGS VFI, but now exploring both concavity and monotonicity, in addition to including the accelerator method. Note that even in this case (which was already extremely fast) the method managed to speed up the convergence time by 2.47 times: without the accelerator method, convergence occurred in 0.94 seconds; including the acceleration method, it converged in just 0.38 seconds!

[5]Actually, with $n = 10$ we have one maximization every 11 iterations; i.e., approximately $1/11 \approx 9.1\%$ of the iterations are maximizations. I considered this case just to consider a "round" value for n. However, the results with $n = 9$ (which, then, would represent 10%) would be virtually identical.

Finally, the Euler equation errors' statistics are reported in Table 4. By the inspection of the Euler equation errors and plots, it is interesting to notice that the numerical results are exactly the same as the results obtained *without* the accelerator method.
Again, focusing on the mean, we can interpret that, on average, a \$1 mistake is made for every $\$10^{3.09} \approx \$1230$ spent.

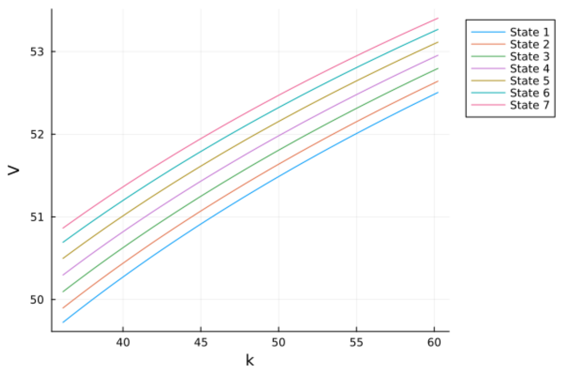
Table 4: Euler Equation Errors (EEEs) statistics —— Accelerator.

Euler Equation Errors' Statistics	
Mean	-3.09447775759327
Median	-3.042207112577707
Maximum	-1.7476141199326618
Minimum	-6.132252761282904

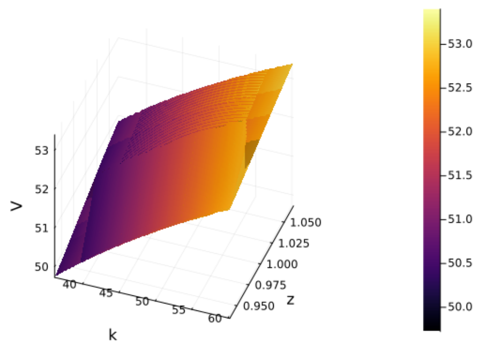
An additional important note regarding the accelerator method: in my codes, I considered "additional iterations" of the accelerator method after *all* maximization iterations — starting with the *first* one. This wasn't an issue in my case, but it could be and it's important to be aware of this. Why in my case this was not a problem? Because I already considered a good enough initial guess from the beginning — Moll's initial guess. If, however, I had taken a bad initial guess (e.g., zeros), convergence could have been seriously compromised. This is because after the first maximization, the approximation of the policy function would be very bad (due to the very bad initial guess) and the accelerator method would perform a series of additional iterations using this very poorly approximated policy function, consequently obtaining a value function very poorly approximated. This would repeat itself as the algorithm proceeded with the iterations and would seriously compromise the convergence of the solution. A simple way to solve this problem would be to force the algorithm to brute force maximize, say, in the first 100 iterations and only then start the acceleration process. Again, thanks to Moll's initial guess, that wasn't necessary here.

□

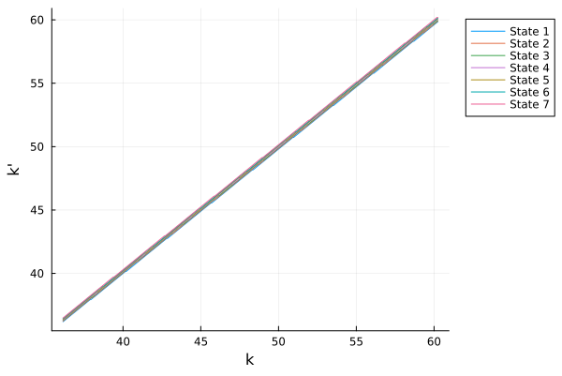
Figure 2: Accelerator results — Value function, policy functions and Euler Equation Errors.



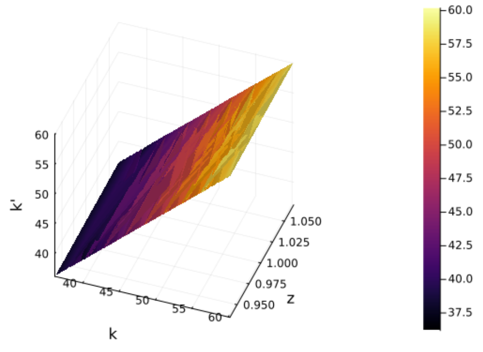
(a) Value function — 2D plot.



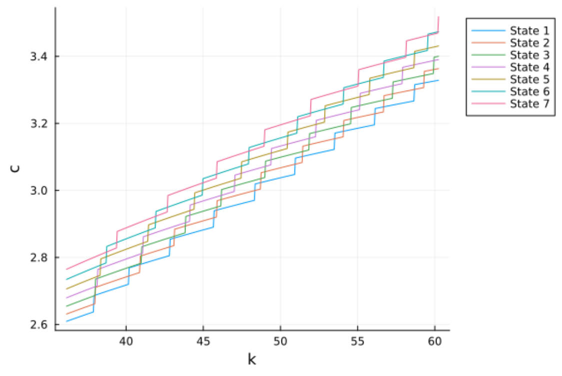
(b) Value function — 3D plot.



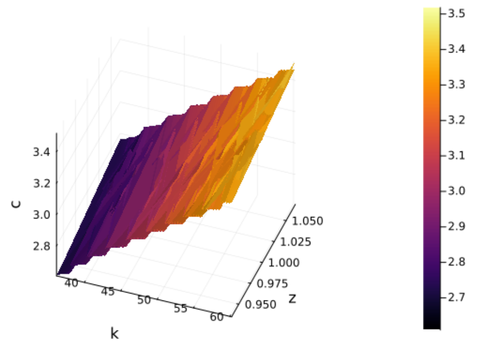
(c) Capital policy function — 2D plot.



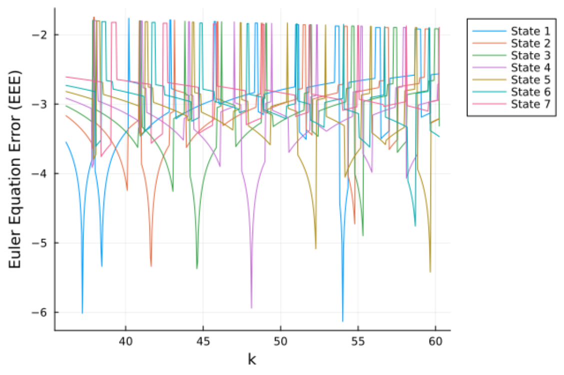
(d) Capital policy function — 3D plot.



(e) Consumption policy function — 2D plot.



(f) Consumption policy function — 3D plot.



(g) Euler Equation Errors.

5. *Now, redo everything using multigrid. First, solve the problem using a grid with 100 points, then 500 and, finally, 5000. For each successive grid, use the previous solution as your initial guess (you will need to interpolate). Compare your results.*

Solution. The idea of the multigrid method is also quite simple. Suppose we want to solve the model for a large capital grid; say 5000 grid points. Solving directly for that grid size using conventional methods would probably take too long — recall the curse of dimensionality. So, the idea is to solve the model for smaller grid sizes first (say, first 100, then 200...) and sequentially use the solution associated with each of those smaller grids as initial guesses for the next grid — until eventually reaching the objective case of a grid size of 5000. In this way, we are able to "cut" a substantial part of the iterative process before reaching the 5000-sized grid: we will then already start the process with an initial guess much closer to the "true" value function, so fewer iterations will be needed to solve the model.

The only technical complication related to this method is that after each resolution (for each grid size) it is necessary to interpolate the value function obtained, in order to convert it to the dimension of the next grid size and thus be able to use it as an initial guess. In Julia, one can use the `Interpolations.jl` package for this.[6] The results are reported in Table 5, as well as in the panels of Figure 3.

Table 5: Time performance for the VFI algorithm — Multigrid, [100, 200, 5000] grid points.

Method *(Parallelizing + Moll's initial guess)*	**Time**	**Iters**
BFGS VFI + Multigrid	3164s	455
Vectorized VFI + Multigrid	469s	455
BFGS VFI + Multigrid + Concavity + Monotonicity	3.6s	455
BFGS VFI + Multigrid + Accelerator (10%) + Concavity + Monotonicity	2.84s	539

As we can see, due to the huge size of the considered grid, solving the model using the most basic brute force grid search method is quite time consuming, even considering the multigrid trick. It took 3164 seconds — roughly 53 minutes. We can imagine how time consuming it would take without the multigrid — it would probably take many (really MANY) hours to solve the model.

Notwithstanding, when considering more sophisticated methods, the time performances become quite satisfactory. Solving the model by vectorization, disregarding any "structural tricks" such as concavity or monotonicity, we reach convergence in 469 seconds — approximately 8 minutes. If we also consider the concavity and monotonicity tricks, convergence is obtained in just 3.6 seconds. Adding in the accelerator method further, the time drops to 2.84 seconds, which is incredibly fast for 5000 grid points. Note that in the end, when

[6] For the results reported here, I considered *linear* interpolations. However, one could consider more sophisticated interpolation methods such as higher-order splines. This could even improve the algorithm's time performance.

compared with the results obtained in the previous exercises, we are solving the model for a grid 10 times *larger*, in a time as satisfactory as the time needed to solve the model for a grid 10 times *smaller*.

At this point, one might ask: why consider a much larger grid? The reason is simple and is evidenced by the Euler equation error statistics reported in Table 6: *accuracy*. As can be seen, compared with the EEEs statistics of the previous exercises, when we consider 5000 grid points (instead of 500) we end up with a much smaller mean for the EEEs: now, on average, a \$1 mistake is made for every $\$10^{3.48} \approx \$3020$ spent, which is significantly better than the \$1 mistake for every \$1230 that we got for the case of a grid size of 500 grid points.

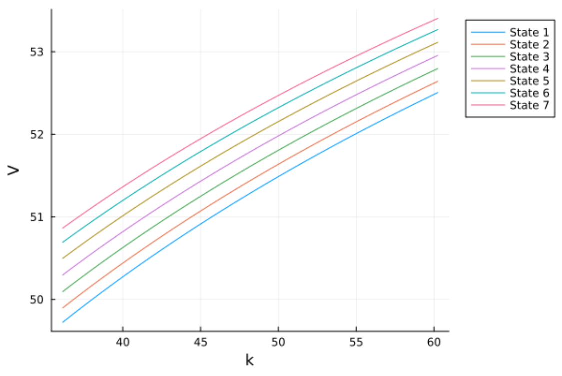
Table 6: Euler Equation Errors (EEEs) statistics — Multigrid.

Euler Equation Errors' Statistics	
Mean	-3.48109673226224
Median	-3.042207112577707
Maximum	-2.7646312478224084
Minimum	-7.976332749269417

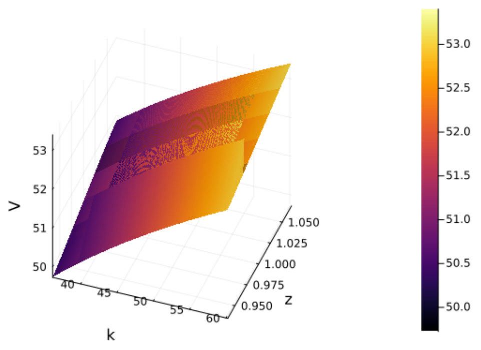
This improvement in the accuracy of the model can also be clearly visualized in the consumption policy function plots, in the panels 3e and 3f. As can be seen, the policy functions for each state are now much smoother than in the previous exercises (compare them with 2e and 2f, for instance). The function is still an "ascending ladder", but now the "rungs" of the ladder are much smaller.

□

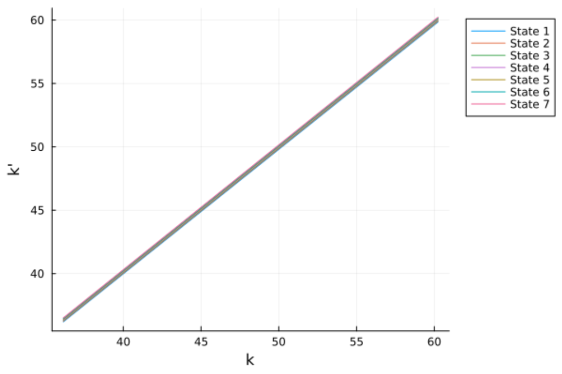
Figure 3: Multigrid results — Value function, policy functions and Euler Equation Errors.



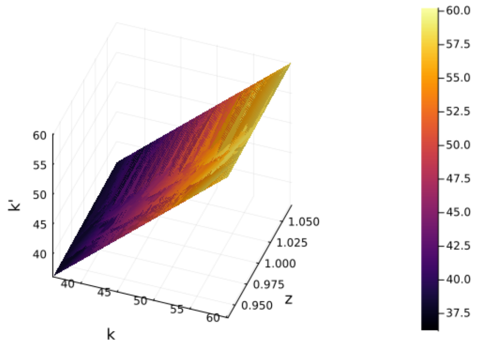
(a) Value function — 2D plot.



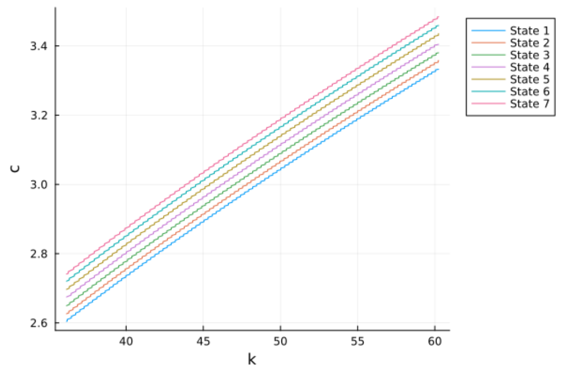
(b) Value function — 3D plot.



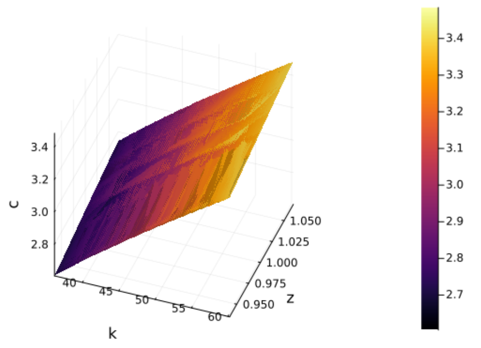
(c) Capital policy function — 2D plot.



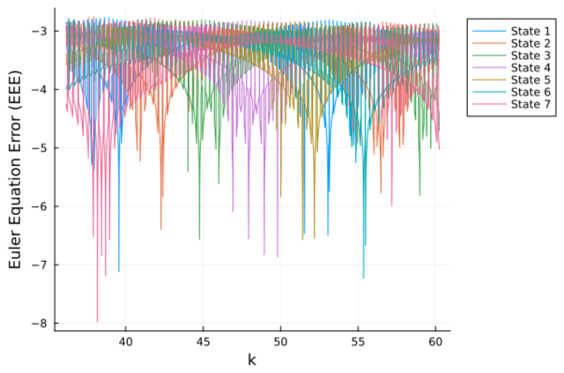
(d) Capital policy function — 3D plot.



(e) Consumption policy function — 2D plot.



(f) Consumption policy function — 3D plot.



(g) Euler Equation Errors.

6. *Now, solve the problem using the endogenous grid method. Compare your results.*

Solution. Finally, we consider the Endogenous Grid Method (EGM). Of all the methods presented so far, this one is the most "different" (and sophisticated). The idea can be briefly explained as follows: instead of considering an exogenous grid for capital, we consider an *endogenous* grid (hence the name of the method). This endogenous grid is obtained by solving for k the FOC of the problem,

$$zk^{\alpha} + (1-\delta)k - \beta^{-1/\mu}\mathbb{E}_{z'}^{-1/\mu}[c(k',z')^{-\mu}(\alpha z' k'^{\alpha-1} + 1 - \delta)] - k' = 0, \tag{8}$$

given a fixed z, an arbitrary set of points k''s and a guessed function $c(k',z')$.[7]

For instance, given a fixed z, a set $\{k'_i\}_{i=1}^{500}$ and a function $c(k',z)$, we solve (8) for k, for each triple $\{(z, k'_i, c_i(k'_i, z'))\}_{i=1}^{500}$ and thus we obtain 500 k's, which form the *endogenous grid* $\{k_i\}_{i=1}^{500}$. Observe that from this we get pairs $\{(k_i, k'_i(k_i, z))\}_{i=1}^{500}$ that *by construction* satisfy (8) — that is: we have 500 "*true*" points of the capital policy function $k'(k,z)$. The idea is then to use these inferred "true" points from the *endogenous* grid to approximate the policy functions in an arbitrary *exogenous* grid of our choice.

Suppose then that we now want to approximate the capital policy function on an *arbitrary* exogenous grid for capital — say, $\{\bar{k}_i\}_{i=1}^{500}$.[8] We can then simply interpolate $\{(k_i, k'_i(k_i, z))\}_{i=1}^{500}$ on $\{\bar{k}\}_{i=1}^{500}$ and get an approximate policy function $\{k'_i(\bar{k}_i, z)\}_{i=1}^{500}$ for the capital on $\{\bar{k}\}_{i=1}^{500}$.[9] From this, we can also trivially obtain an associated approximate policy function $\{c_i(\bar{k}_i, z)\}_{i=1}^{500}$ for consumption, using the constraint of the problem. Note, however, that the procedure described so far is assuming a certain fixed state value for z. In practice, to solve the model completely, we need to repeat this procedure $\#Z_{\text{grid}}$ times; that is, we must repeat the procedure separately for each $z \in Z_{\text{grid}}$. In particular, for the specification of this exercise, we repeat the procedure 7 times, once for each possible value of z — which means that at the end we calculated and interpolated on 7 different endogenous grids, one for each state. This will finally yield the complete approximate policy functions $\{k'_i(\bar{k}_i, z_j)\}_{i,j=1}^{500,7}$ and $\{c_i(\bar{k}_i, z_j)\}_{i,j=1}^{500,7}$ on $\{\bar{k}_i\}_{i=1}^{500}$.

These, however, are just approximations of the solutions — and they can be poor approximations. How can we improve them? Well, we can just repeat the same process again, but now using the approximate consumption policy function just obtained as the initial guess for $c(k',z')$! We repeat this process until we reach convergence of the consumer policy function,[10] given a predefined tolerance level.[11]

[7] Initially, the k''s and the function $c(k',z')$ must be arbitrarily guessed. I guessed a linearly spaced grid (as defined in Exercise 3) for the k''s and calculated the initial guess for the policy function from it, as $c(k',z') = z'k'^{\alpha} - k' + (1-\delta)k', \quad \forall z' \in Z_{\text{grid}}$, where Z_{grid} comes from Tauchen's method, as usual.

[8] That is not necessarily (and will most likely *not* be) equal to the endogenous grid, $\{k_i\}_{i=1}^{500}$. I consider, for simplicity, $\{\bar{k}_i\}_{i=1}^{500}$ to be the grid defined in Exercise 3.

[9] Again, I consider *linear* interpolations. However, as already discussed, one could consider more sophisticated interpolation methods such as higher-order splines and this could even improve the algorithm's time performance and accuracy.

[10] The theory guarantees that the approximate sequence of consumer policy functions will converge.

[11] I consider, as usual, a tolerance level of 10^{-5}.

Results are reported in Table 7 and in the panels of Figure 4.

Table 7: Time performance — Endogenous Grid Method.

Method *(Parallelizing + Moll's initial guess)*	**Time**	**Iters**
Endogenous Grid Method	23.1s	175

As we can see, convergence occurs reasonably quickly — 23.1 seconds — with a few iterations — 175. The most interesting result about the Endogenous Grid Method, however, has nothing to do with time performance; but instead with *accuracy*. Euler equation errors' statistics are reported in Table 8. As we can see, the mean of EEEs for the EGM is significantly lower than the mean of EEEs observed in all previous exercises. This indicates that the EGM provides a much more accurate solution than the other methods. Just to give you an idea of the accuracy of the EGM solution, we now have that, on average, a \$1 mistake is made for every $\$10^{4.06} \approx \$11481$ spent, which is *much* better than the \$1 mistake for every \$1230 or \$3020 that we obtained in previous exercises, for other methods.

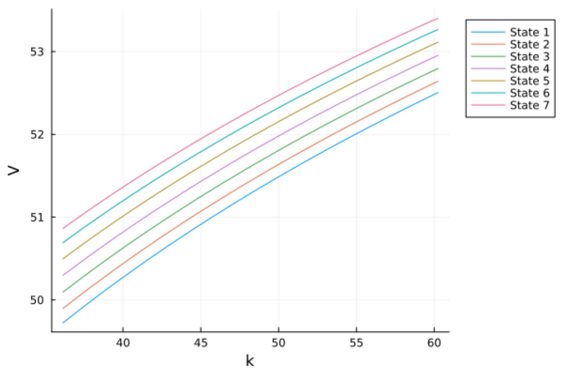
Table 8: Euler Equation Errors (EEEs) statistics — Endogenous Grid Method.

Euler Equation Errors' Statistics	
Mean	-4.062291509221267
Median	-3.9320265015282665
Maximum	-3.518341888782513
Minimum	-7.517329953580737

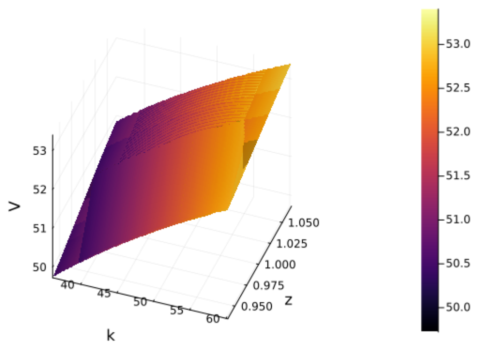
Again, this improvement in the accuracy of the model can also be clearly visualized in the consumption policy function plots, in the panels 4e and 4f. As can be seen, the policy functions for each state are now *super* smooth — much smoother than in the previous exercises (compare them with 2e and 2f, and also 3e and 3f, for instance). If you look very closely (by super zooming in), you'll see that the function is still a "ascending ladder", but now the "rungs" of the ladder are super smaller, so we can barely see them.

□

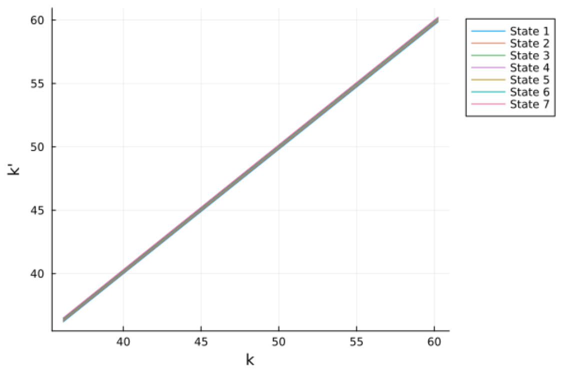
Figure 4: Endogenous Grid Method (EGM) results — Value function, policy functions and Euler Equation Errors.



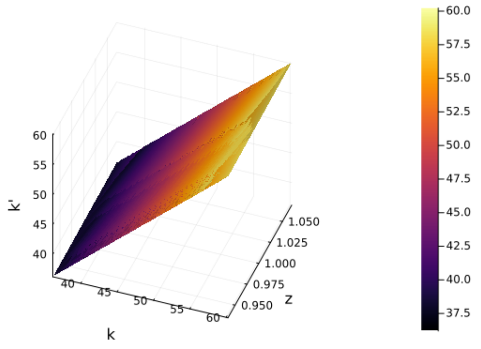
(a) Value function — 2D plot.



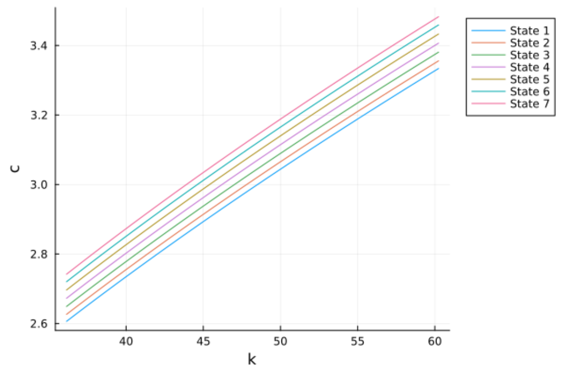
(b) Value function — 3D plot.



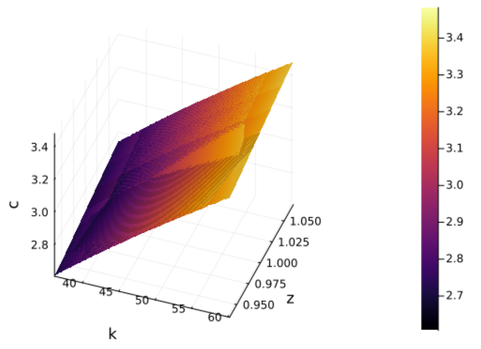
(c) Capital policy function — 2D plot.



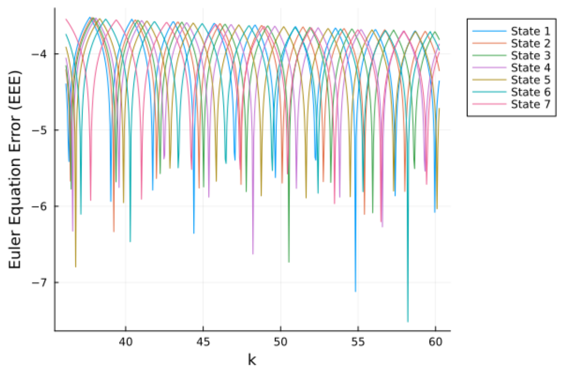
(d) Capital policy function — 3D plot.



(e) Consumption policy function — 2D plot.



(f) Consumption policy function — 3D plot.



(g) Euler Equation Errors.

Appendix A The "vectorization" method

The idea underlying vectorization is to represent the economy as a whole (all possible values of the Bellman equation, more precisely) in a gigantic array generated by efficient vector operations and then obtain the value function (capital policy function, respectively) simply by applying operators of maximum (argmax) on certain vectors that make up this array. For the case of the model considered in the exercises of this problem set, this array will be a three-dimensional array, with dimension $\#K_{\text{grid}} \times \#Z_{\text{grid}} \times \#K_{\text{grid}}$. In particular, for Exercise 3, for example, an array of dimension $500 \times 7 \times 500$ was considered; that is, an array of 1,750,000 entries. The idea is that the first two coordinates of the array represent the states k and z, respectively, while the third coordinate represents the variable of choice, k'. So, with this array in hand, to find the value function (capital policy function, respectively) it is enough, for each pair (k, z), to take the max (argmax, respectively) in relation to the third coordinate of the array.

Why (or how) is solving the model this way much more efficient? What happens is that this way we avoid the brute force grid search necessary in the other methods to find the optimal k''s (that is, to maximize the bellman equation). It turns out that generating the giant array containing all the information in the economy by means of vector operations is, in general, much faster than calculating and storing the values of the Bellman equation one by one, without vector operations, for each possible combination of (k, z, k')'s — as is done in other methods to then identify the maximum.

And what are the limitations of this method? Unfortunately, the curse of dimensionality once again haunts us. For very large grid sizes and/or a very large number of variables, this array can take on such large proportions that we can have serious computational memory allocation problems. For example, in Exercise 5, solving by vectorization, when we reach the grid of size 5000 we need to generate an array of dimension $5000 \times 7 \times 5000$ — that is, an array of 175 million entries. Note the curse of dimensionality in action: we increased the capital grid by only ten times, but the "complexity" of the economy increased by a hundredfold: we went from an array of 1.75 million entries to an array of 175 million entries.

Even for this model, which is very simple, generating the grid of size 5000 from Exercise 5 was computationally quite expensive and consumed a significant amount of memory. It is, therefore, important to keep in mind that for more complicated models, with more control and state variables and larger size grids, storing an array like this may simply be computationally impossible — and this is the main limitation of this "method".