UPENN. // Computational Econonmics // 714 //PS01

Rodrigo A Morales M

2nd December 2019

1 Recursive SPP:

The social planner in our economy maximizes the expected lifetime utility

$$\mathbb{E}_0 \left[\sum_{t=0}^{\infty} \beta^t \left(c_{1,t}^{\theta} c_{2,t}^{1-\theta} - \frac{(l_{1,t}+l_{2,t})^2}{2} \right) \right]$$

subject to:

$$c_{1,t} + k_{t+1} = e^{z_t} k_t^{\alpha} l_{1,t}^{1-\alpha} + (1-\delta)k_t$$
$$c_{2,t} = A_t l_{2,t}$$

The household budget constraint (along with prices) is irrelevant for the social planner. The planner chooses the amount of capital tomorrow as well as consumption and labor for each of the two goods in the economy, i.e. $\{k_{t+1}, c_{1,t}, c_{2,t}, l_{1,t}, l_{2,t}\}$.

Recursive Social Planner Problem. For the notation of our problem, let the amount of capital today and the technology shocks be our state variables, $S = (k, z, A)$.

$$\mathbb{V}(k, z, A) = \max_{k', l_1, l_2 \geq 0} \left\{ c_1^{\theta} c_2^{1-\theta} - \frac{(l_1+l_2)^2}{2} + \beta \sum_{z', A'} \pi\left(z', A' | z, A\right) \mathbb{V}\left(k', z', A'\right) \right\}$$
$$\text{s.t.} \quad c_1 + k' = e^z k^{\alpha} l_1^{1-\alpha} + (1-\delta)k$$
$$c_2 = A l_2$$

2 Steady State:

To find the steady state, we use $(z_{ss}, A_{ss}) = (0, 1)$. Also, notice that from the SPP, the relevant decision variables are $\{l_1, l_2, k\}$ (as c_1 and c_2 are determined by them), so we only need the corresponding FOCs of the Recursive Problem, and solve for them (computationally) under the restriction that $(l_1, l_2, k, k') = (l_{1,ss}, l_{2,ss}, k_{ss}, k_{ss})$.

$$[l_1]: \quad \theta(1-\alpha) k_t^{\alpha} l_{1,t}^{-\alpha} \left(k_t^{\alpha} l_{1,t}^{1-\alpha} + (1-\delta)k_t - k_{t+1} \right)^{\theta-1} l_{2,t}^{1-\theta} - (l_{1,t} + l_{2,t}) = 0$$
$$[l_2]: \quad (1-\theta) \left(k_t^{\alpha} l_{1,t}^{1-\alpha} + (1-\delta)k_t - k_{t+1} \right)^{\theta} l_{2,t}^{-\theta} - (l_{1,t} + l_{2,t}) = 0$$

$$[k']: \quad -\theta c_{1,t}^{\theta-1} l_{2,t}^{1-\theta} + \theta\beta c_{1,t+1}^{\theta-1}\left(\alpha k_{t+1}^{\alpha-1} l_{1,t+1}^{1-\alpha} + 1 - \delta\right) l_{2,t+1}^{1-\theta} = 0$$

In the steady state, $k = k' = k_{ss}$. Thus, the above system reduces to:

$$\theta(1-\alpha) k_{ss}^{\alpha} l_{1,ss}^{-\alpha}\left(k_{ss}^{\alpha} l_{1,ss}^{1-\alpha} - \delta k_{ss}\right)^{\theta-1} l_{2,ss}^{1-\theta} - (l_{1,ss} + l_{2,ss}) = 0$$

$$(1-\theta)\left(k_{ss}^{\alpha} l_{1,ss}^{1-\alpha} - \delta k_{ss}\right)^{\theta} l_{2,ss}^{-\theta} - (l_{1,ss} + l_{2,ss}) = 0$$

$$\alpha\beta k_{ss}^{\alpha-1} l_{1,ss}^{1-\alpha} + \beta(1-\delta) - 1 = 0$$

This is a system of three equations and three unknowns: $k_{ss}, l_{1,ss}$ and $l_{2,ss}$.

It is about time to let the reader know that in what follows all the programming lines and programs are written in Julia. I will not include those lines which are trivial or which would make this document too heavy to read. Similarly, many plots are the same as the Fixed Grid, so I will not show them all, only the first time computing the results, those which are asked, and the ones which are important to make a point, for instance about a tradeoff between accuracy and running time.

Calling a simple numerical solver built in Julia gives a solution for these steady state values. Using the resource constraints in each sector, one can recover the steady state values of consumption, as well as the Value function in the steady state. These values will be used in the programs that follow, specially as initial solutions or for the capital grid.

```
Julia Code for problem set 1 by RAMM began running...

Modules and packages loaded ok

Steady state

K_ss = 0.8301903102383987  L1_ss = 0.23498974340922302  L2_ss = 0.269031277
7924755  C1_ss = 0.2733757991290202 C2_ss = 0.2690312777924755

----------------------------------------------------------------------
```

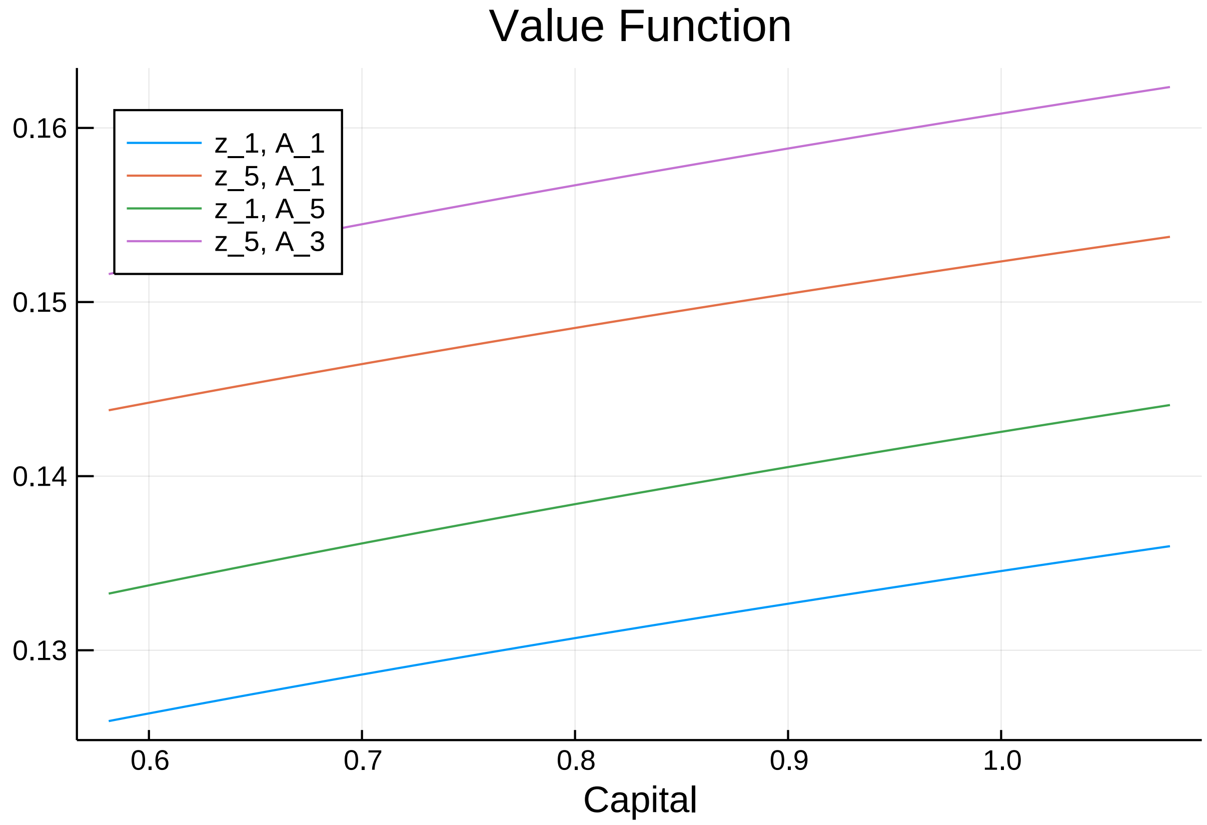
3 VFI with a Fixed Grid:

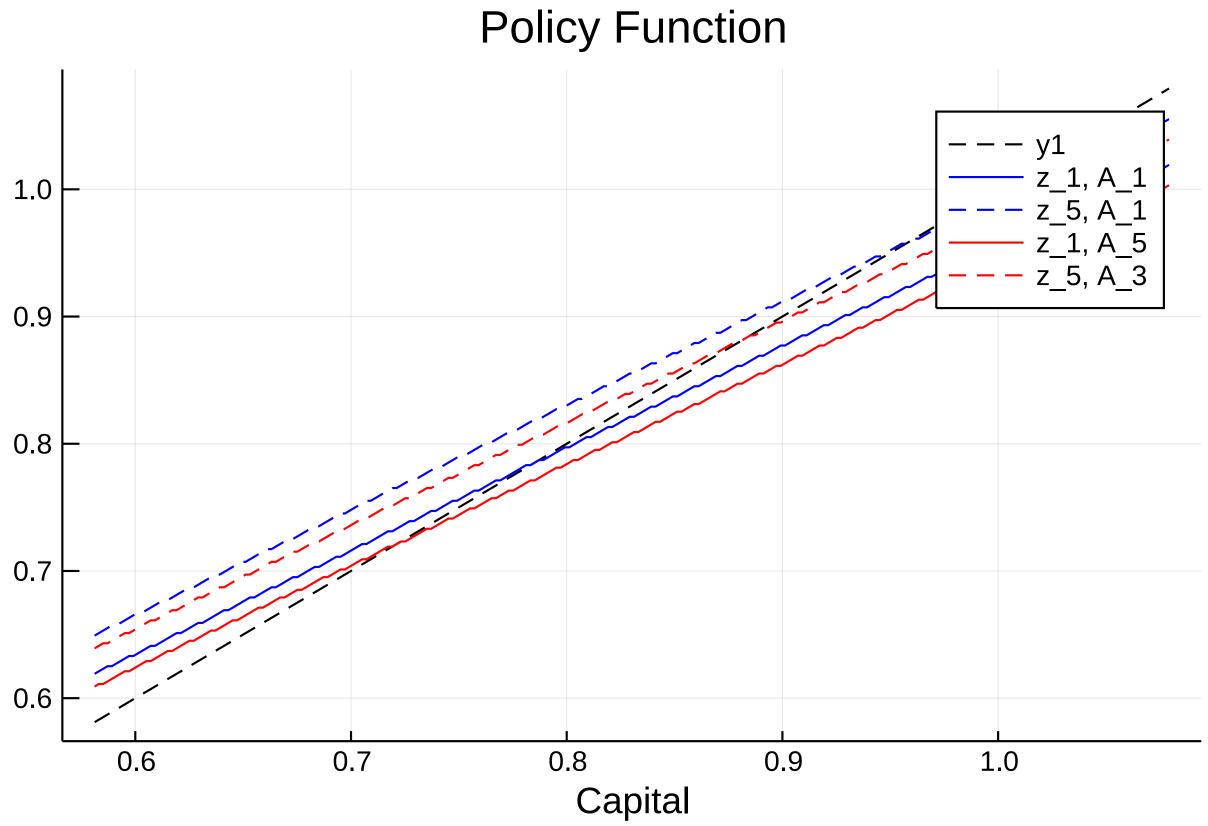
A fixed grid of 250 points around the steady state is used $(0.7k_{ss}, 1.3k_{ss})$, as well as linear interpolation. The method is based on solving for l_1 and l_2 given combinations of (k, k'), and using those solutions over a VFI to find the optimal policy (of k'), as well as the Value Function.

```
calling a_fixed_grid_optimized...

Getting tensor of labors...
 Tensor of labors computed...
 VFI starts....
 Iteration = 1 Sup Diff = 0.280655137754657
 Iteration = 10 Sup Diff = 0.01675651002137744
 Iteration = 20 Sup Diff = 0.009618491578490125
 Iteration = 30 Sup Diff = 0.005853292403856835
 Iteration = 40 Sup Diff = 0.003621157001082652
 Iteration = 50 Sup Diff = 0.0022619606296384567
 Iteration = 60 Sup Diff = 0.001424794584646469
 Iteration = 70 Sup Diff = 0.0009044363701310624
 Iteration = 80 Sup Diff = 0.0005782122477416324
 Iteration = 90 Sup Diff = 0.0003720244908486374
 Iteration = 100 Sup Diff = 0.00024071442465124594
 Iteration = 110 Sup Diff = 0.00015651236760517236
 Iteration = 120 Sup Diff = 0.00010218678692343153
 Iteration = 130 Sup Diff = 6.694976190680445e-5
 Iteration = 140 Sup Diff = 4.398977838155852e-5
 Iteration = 150 Sup Diff = 2.8971918564407816e-5
 Iteration = 160 Sup Diff = 1.9117602933869356e-5
 Iteration = 170 Sup Diff = 1.2634545649084104e-5
 Iteration = 180 Sup Diff = 8.36032518828525e-6
 Iteration = 190 Sup Diff = 5.537525089762495e-6
 Iteration = 200 Sup Diff = 3.6707050217274194e-6
 Iteration = 210 Sup Diff = 2.4347474343781313e-6
 Iteration = 220 Sup Diff = 1.6157438698394177e-6
 Iteration = 230 Sup Diff = 1.0726555427263083e-6
216.714239 seconds (6.24 G allocations: 93.098 GiB, 22.29% gc time)
```

3.1 Plots:

The plots show the Value Function for different combination of shocks. In general, given a higher shock, the Value function is higher. This is consistent with what is expected from the model. Nontheless, the Policy Function behaves a little different, because once the agent is able to save a given amount, he prefers to consume the extra amount of capital.





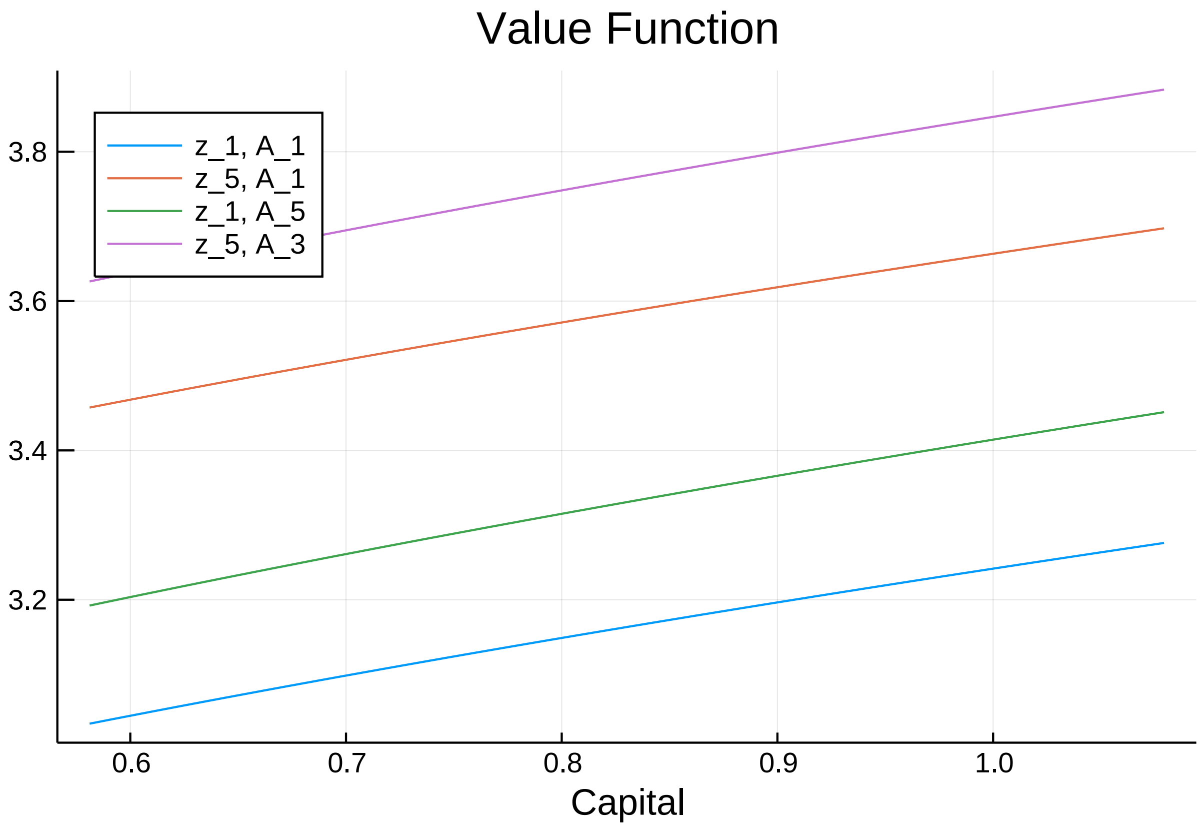
4 Endogenous grid

The endogenous grid algorithm is based on both Carroll (2006) and Barillas and FernandezVillaverde (2006). The main idea is to use a different approach for the same VFI. Basically, first a matrix for Y is computed, where Y is a function of capital and the shocks. Now, there is a correspondence between Y and the consumption through the capital. What is different from the problem at hand is that we have two consumptions and two conditions, and that is why we need to use two First Order Conditions.

This algorithm, together with Stochastic were by far the ones that required the longest to implement.

```
calling endogenous Grid...

VFI EGM fixed labor...

egm_lss finished
 Number of Iterations = 479 Sup Diff = 9.655091208601916e-8
 15.417289 seconds (4.23 M allocations: 1.355 GiB, 4.55% gc time)
```

4.1 VFI Plot of the Endogenous Grid



5 Comparison of Grids

The accuracy of both programs is good, as the same level of tolerance was used for both of them.

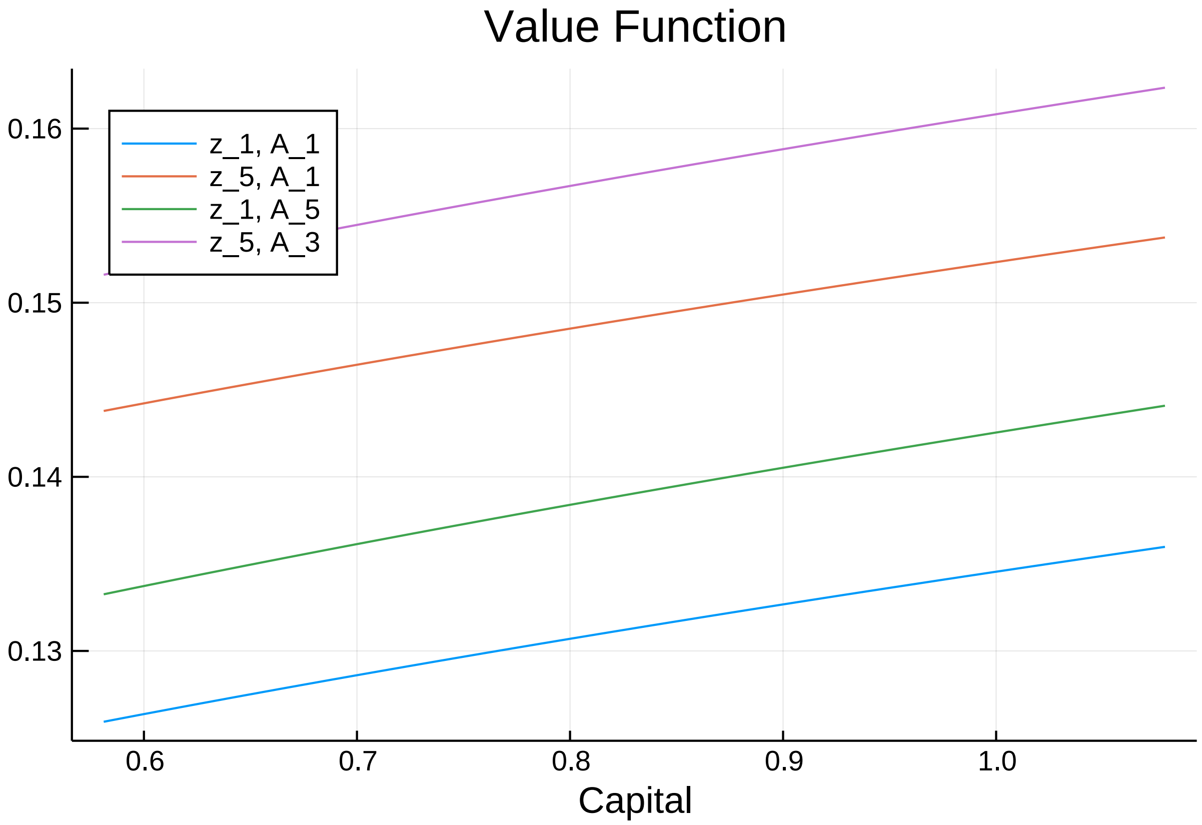
The computing time is much faster for the Endogenous Grids. Nontheless, the time it required to implement more than surpasses the benefit of its running time. If it were the case that some general equilibrium condition required to solve the problem many times, perhaps only then it would be beneficial to consider using such endevour in coding the endogenous implementation, which by the way is more prone to errors when coding, as it also requires a lot more algebra.

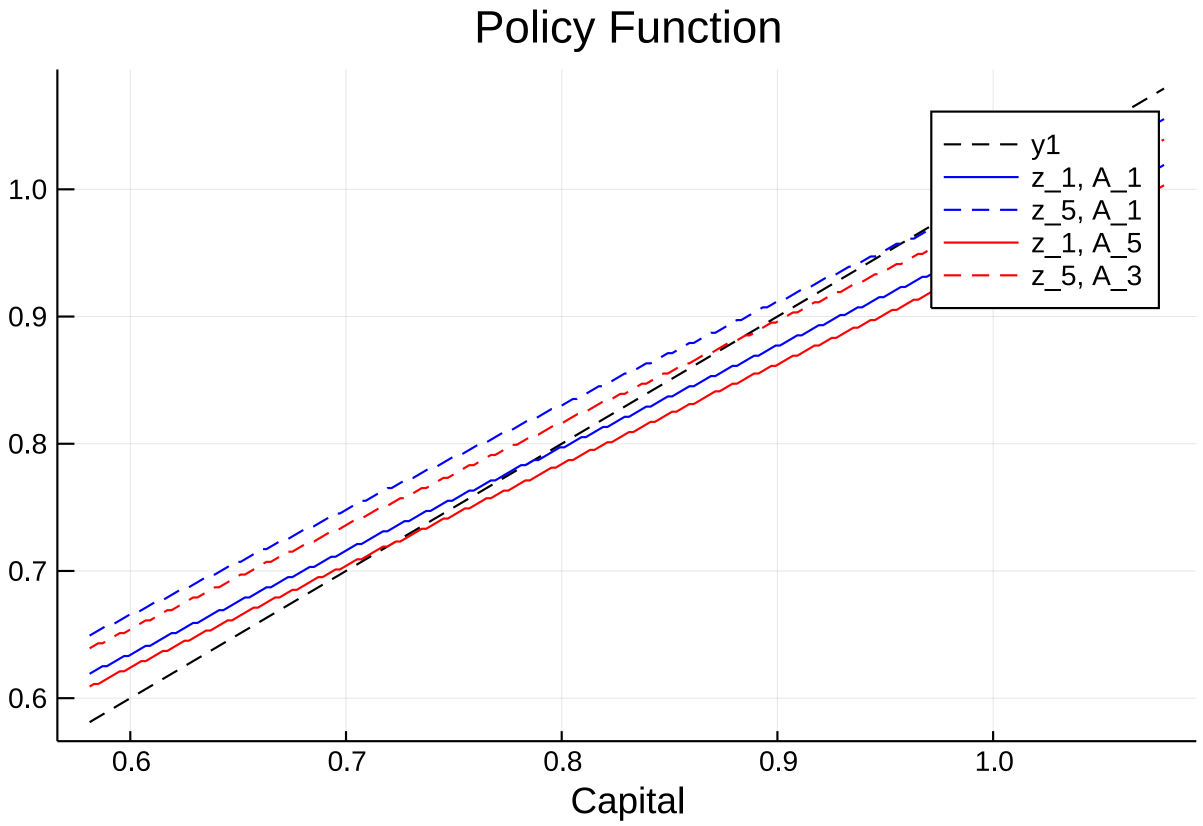
6 Accelerator:

This algorithm basically skips the Bellman maximization 9 out of 10 times. The accuracy is slightly sacrificed, not to a point in which it is significant for our problem, but there is a significant reduction in computing time. The plots show Value Functions and Policy Functions which are not distinguishable to the naked eye from the ones obtained through the simple Value Function Iteration.

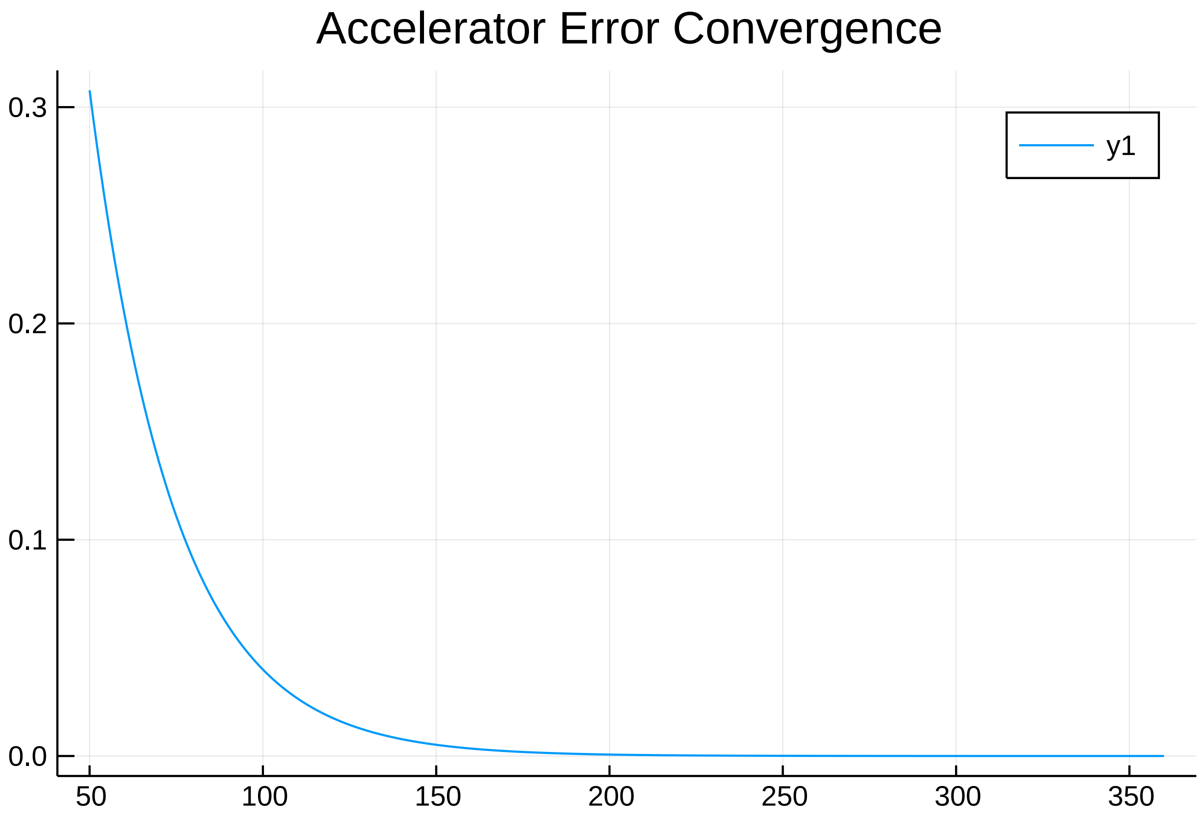
```
Calling Accelerator...

Getting tensor of labors...
 Tensor of labors computed...
 VFI starts....
Accelerator finished
 Number of Iterations = 360 Sup Diff = 9.825041917304719e-7
 35.639594 seconds (1.04 G allocations: 15.473 GiB, 22.60% gc time)
```

6.1 Plots of the Accelerator





I also plot the error grid for the iterations, to see the slight increase there is in the error over the iterations:



7 Multigrid

Following Chow and Tsitsiklis (1991), Multigrid Algorithm is used. Now, in order to exploit the fact that every iteration calculates the Value Function and the Policy Function, I compute a function which generates points in between the points already found, in a spirit similar to the Cantor set, along with dictionary which save the values for the Value Function and the labor functions. The main reason for doing this, instead of a simple interpolation is precisely because the labor functions are the result of a nl solver. Now, although it is true that labor is not very responsive to changes, and although it could also be interpolated over, I prefer to use the the exact values, as this should show more accuracy and should be faster, so there is no reason why for this particular problem it would be desirable to do otherwise.

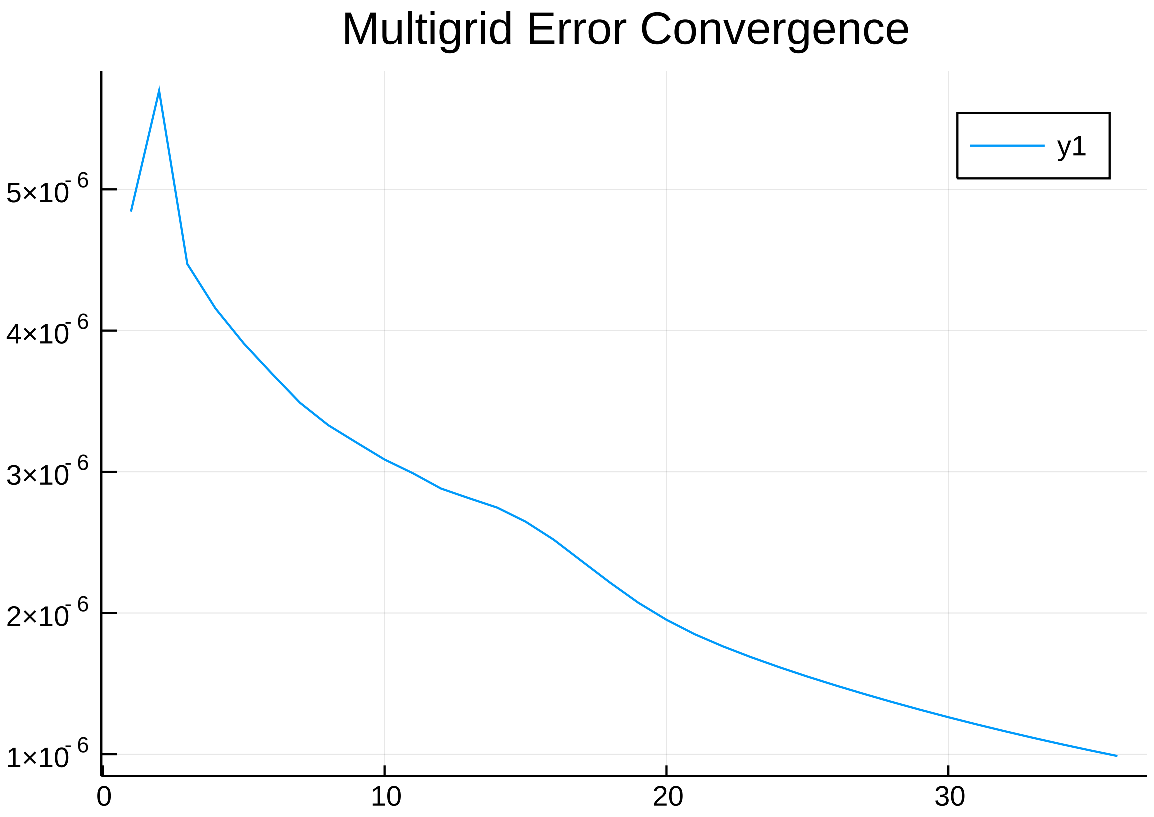
The algorithm performs first 129 capital points, then 513 and finally 4097, roughly the the (100, 500, 500) asked for.

```
Calling Multigrid...

Multigrid starts ;)....
Solving for grid number 1

Finding labour choice matrices for all k,kprime' ...

VFI ...

Multigrid for grid number 1 finished solving
 Number of Iterations = 208 Sup Diff = 9.614181311736666e-7
Solving for grid number 2

Finding labour choice matrices for all k,kprime' ...

VFI ...

Multigrid for grid number 2 finished solving
 Number of Iterations = 64 Sup Diff = 9.68985429850145e-7
Solving for grid number 3

Finding labour choice matrices for all k,kprime' ...

VFI ...

Multigrid for grid number 3 finished solving
 Number of Iterations = 36 Sup Diff = 9.873087453005005e-7
Multigrid finished
 Number of Iterations = 36 Sup Diff = 9.873087453005005e-7
653.827657 seconds (16.65 G allocations: 263.613 GiB, 27.40% gc time)
```

The plot of the policy function and value function is already good enough even for the 513 points grid (plot is omitted because it looks exactly like the acceleartor ones).

In order to compare the accuracy of the multigrid, one can see the running time of the function, I present the plot of the error as it moves along the iterations.

7.1 Plot of convergence of error:



8 Stochastic Grid

Stochastic Grid follows Rust (1997), which proposes an algorithm with some challenges for its implementation. Given that our exogenous states are only $3x5$, I decided only to make a random draw over the grid of capital. For the same reason, I use the same grid over all states, making the interpolation a bit easier. Nontheless, precisely for this reason, this algorithm is not really well suited for the problem at hand, as it does not really reduce significantly the curse of dimensionality (given that the other algorithms are already very efficient), and indeed, as one can see in the plots, the accuracy is reduced, and the time is not really much smaller than, for instance, that of the Accelerator, specially considering the amount of time required to code it.

The grid of capital is chosen for $(0.85k_{ss}, 1.25k_{ss})$, in contrast with the previous results for a 35 percent interval.

The grid was chosen of size 1000, to give room to the number of draws to actually grow into a large number, and allow for the convergence theorem (contraction mapping) to be consistent with the experiment.

Also, because the algorithm was having trouble in converging once around errors of $1e-5$, I reduced the tolerance to $1e-6$.

```
Calling Stochastic...

Getting tensor of labors...
```

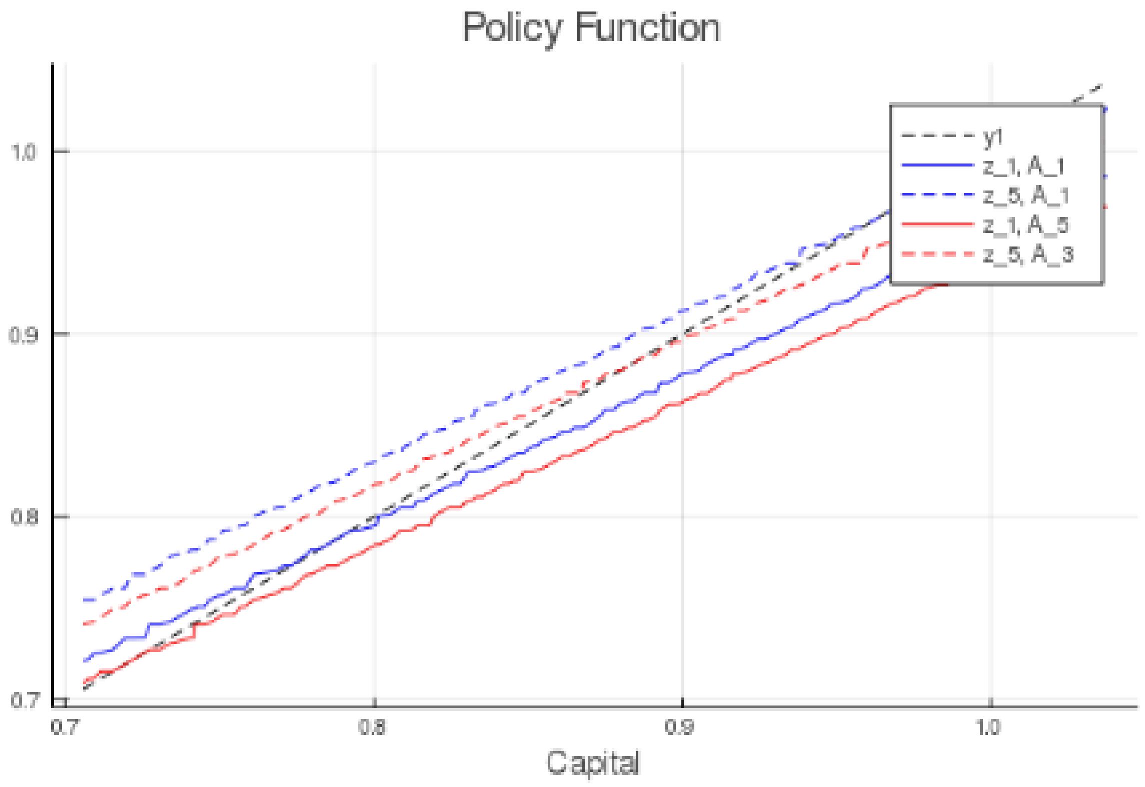


Figure 1: Policy Function for the Stochastic Grid

```
Tensor of labors computed...

VFI starts....

Accelerator finished

43.567893 seconds (1.51 G allocations: 23.395 GiB, 10.46% gc time

Number of Iterations = 330 Sup Diff = 1.5725093996388217e-6

1.861284 seconds (44.71 M allocations: 694.960 MiB, 16.63% gc time)
```

8.1 Plot of the Stochastic Grid: