Lista II - Métodos Numéricos

EPGE - 2018

Professor: Cézar Santos

Aluno: Raul Guarini Riva

Problema 1. O problema no planejador consiste em escolher recursivamente quanto consumir no presente e quanto poupar na forma de capital para o próximo período, tendo por base o estado da economia formado pelo estoque de capital atual e a produtividade atual.

A função utilidade das famílias não leva em consideração o lazer. Desta forma, ofertarão trabalho inelasticamente. Normalizando a dotação de trabalho para uma unidade, a equação funcional com a qual o planejador se defronta é a seguinte:

$$V(K,z) = \max_{c,K'}\{u(c) + \beta\mathbb{E}[V(K',z')|z]\}$$
$$\text{s.t.} \quad c + K' \leq zK^{\alpha} + (1-\delta)K$$

A hipótese de u estritamente crescente implica que a restrição de recursos será satisfeita com igualdade em todo instante do tempo. Daí, reescreve-se:

$$V(K,z) = \max_{K'}\{u(zK^{\alpha} + (1-\delta)K - K') + \beta\mathbb{E}[V(K',z')|z]\}$$

Problema 2. Supondo não haver incerteza, normaliza-se o valor da produtividade para a média incondicional do processo:

$$\log(z) = 0 \implies z = 1$$

A equação de Euler é dada por

$$\beta\mathbb{E}\left[\left(\frac{c'}{c}\right)^{-\mu}(z'\alpha K'^{\alpha-1} + 1 - \delta)\big|z\right] = 1$$

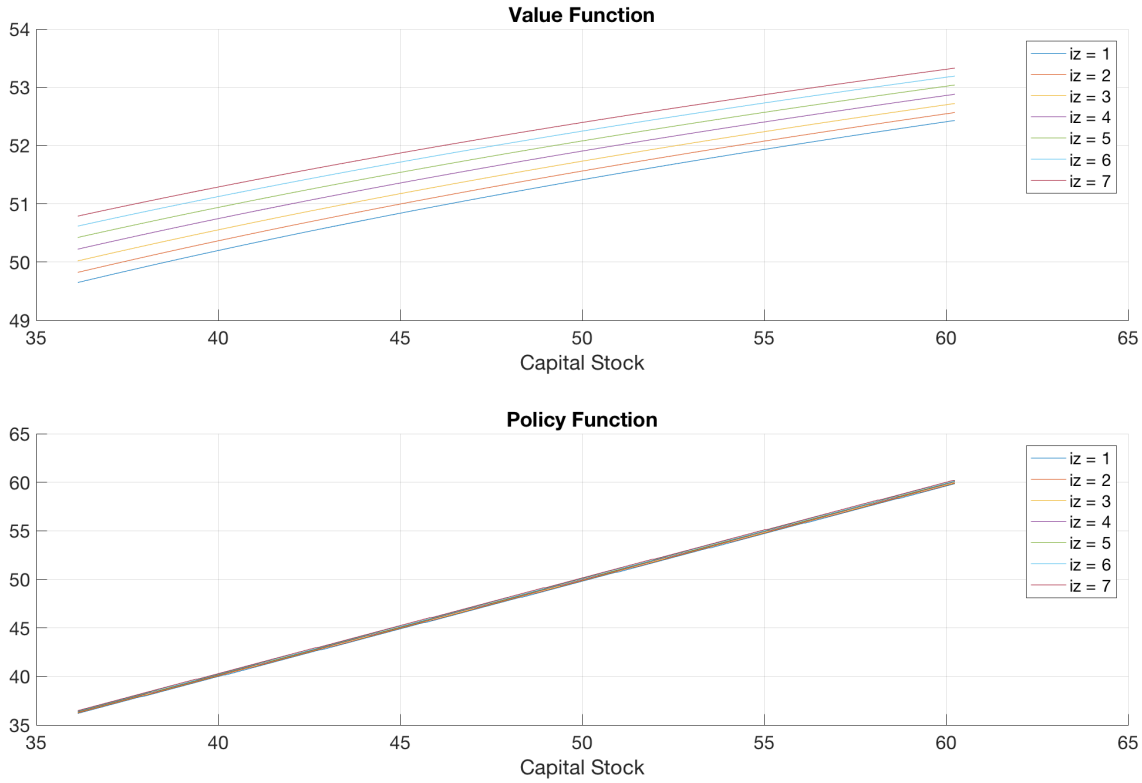
Sem incerteza e já resolvendo para o estado estacionário:

$$\beta\alpha K_{ss}^{\alpha-1} = 1 - \beta(1-\delta)$$
$$\boxed{K_{ss} = \left(\frac{\alpha\beta}{1-\beta(1-\delta)}\right)^{\frac{1}{1-\alpha}}}$$

Com a calibração sugerida na lista, temos $K_{ss} = 48.1905$.

Problema 3. O arquivo com o código principal da lista é `ps2.m`. Para esta primeira implementação, explorei a monotonicidade da função política e a concavidade do funcional sendo otimizado. O código está no formato de uma função chamada `VFinder_Iterated.m`, devidamente documentada através do próprio `help` do MATLAB.

A primeira iteração foi feita através de força bruta e as próximas seguiram explorando os aspectos teóricos do problema. Abaixo, a função valor e a função política (como esperado, crescente!!!):



Obtive o resultado, também esperado, de que a função valor, para um nível de k fixo, é crescente na TFP. A legenda iz $= 4$, por exemplo, indica que determinada curva foi computada com o valor da TFP correspondente ao quarto ponto do grid do choque $\log(z)$.

Para comparar o desempenho desta abordagem (que não utiliza nenhum tipo de vetorização), implementei o método de "força bruta vetorizada". Isto é, operar maximizações ao longo das dimensões de arrays do MATLAB evitando criar "for loops". Esta abordagem está no script `brute_force.m`.

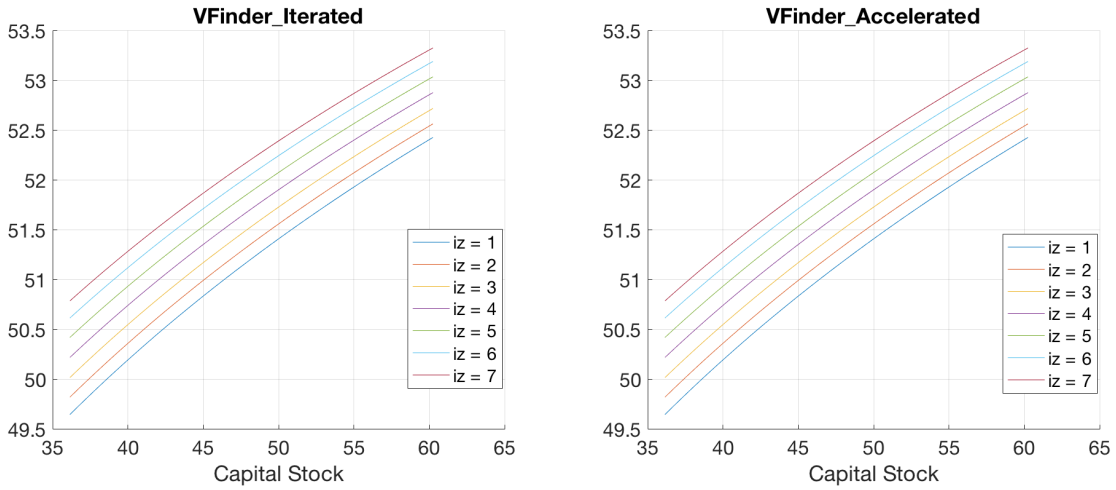
Houve um ganho de desempenho considerável. O método de força bruta vetorizada computou a função valor e a função política em 9.96 segundos. O método que utiliza a função `VFinder_Iterated.m` resolveu o mesmo problema em 7.84 segundos. Isto representa um ganho de quase 22%, o que considero expressivo.

Nota-se, entretanto, que o método de força bruta é mais geral que seu concorrente, em vista do fato que funcionaria igualmente bem para um problema em que não tivéssemos um resultado teórico garantindo a monotonicidade da função política ou a concavidade do funcional em questão. Uma performance pior é o preço a ser pago pela maior versatilidade.

Problema 4. A função `VFinder_Accelerated` implementa o mesmo algoritmo do problem anterior, se valendo da técnica do acelerador, contudo. Seu último argumento, `pct`, informa à função qual a frequência em que deve ser utilizada uma otimização completa: atualização da função política e da função valor. Por exemplo, `pct = 0.1` faz com que em apenas 10% das iterações seja utilizado um programa de otimização completa (cômputo da função política e atualização da função valor).

Nas 50 primeiras iterações do algoritmo, otimizações completas são feitas, com o objetivo de colocar o processo "na rota de convergência" (isto é controlado através do parâmetro `safety_net` no escopo da função).

Abaixo, uma comparação das funções valor encontradas. Como esperado, temos a mesma solução (a menos de erros de aritmética de ponto flutuante):



A função `VFinder_Accelerated` resolveu o modelo em 7.66 segundos, cerca de 3% mais rápido do que `VFinder_Iterated`. O ganho de velocidade não foi expressivo, em minha visão, porque o algoritmo rival já não usa, a não ser na primeira iteração, o operador max(), em vista da maneira como a concavidade do funcional foi explorada. Isto reduz o ganho de performance potencial do acelerador. Quando comparamos esta solução com a implementação de força bruta, entretanto, notamos um ganho de performance de mais de 3 segundos, mais de 30%![1]

Problema 5. O método do multigrid mostrou-se poderoso. Analisando o formato da função valor encontrada através dos outros dois métodos e em vista da facilidade/rapidez de sua implementação, optei pelo uso da interpolação linear. A curvatura de V ao redor do estado estacionário não parece tão grande.

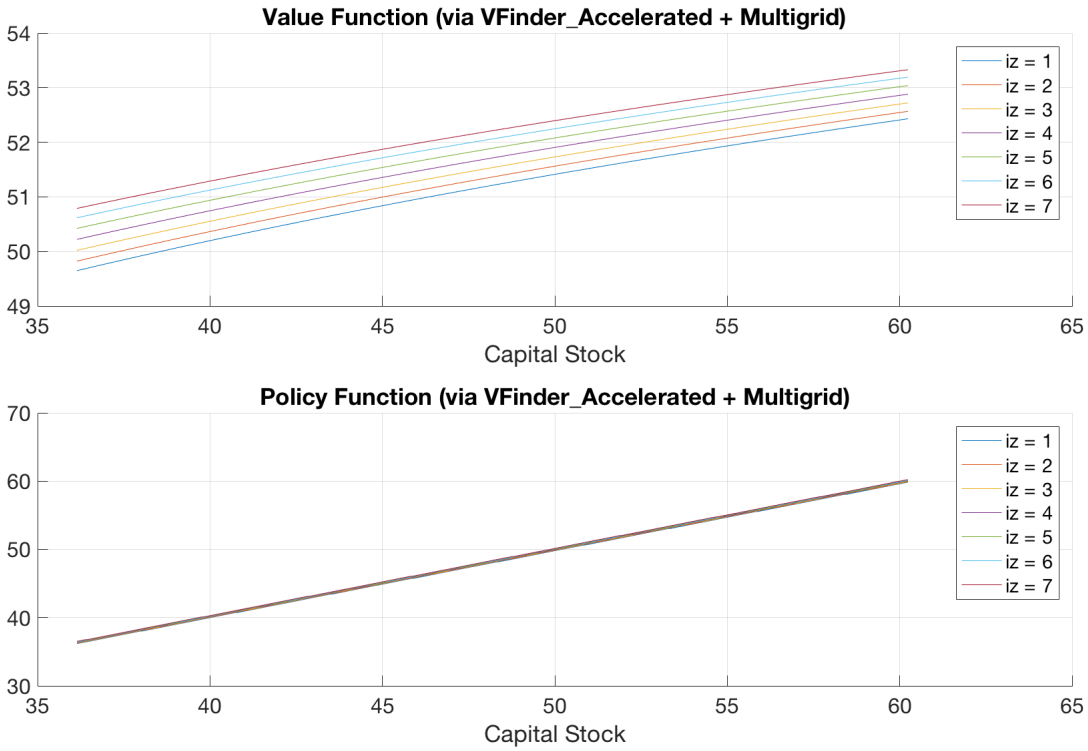
Dado um chute inicial para a função valor, resolvi o problema funcional num grid inicial com 100 pontos, interpolei os valores encontrados num grid mais fino de 500 pontos e utilizei o interpolante como um novo chute inicial para resolver o problema no grid mais fino. Esta interpolação ocorreu linearmente para cada valor de z fixo. Resolvido o problema no grid de 500 pontos, repeti o processo visando resolver a equação funcional num grid de 5000 pontos para k. O algoritmo levou ao redor de 73 segundos para convergir.

Claro, este problema é mais complicado do que o problema anterior uma vez que multiplicamos por 10 o número de pontos no grid de k. Como discretizamos também a variável de controle, aumentamos em 100 vezes o número de pontos considerados.

Para comparar o desempenho do multigrid com os dois outros métodos implementados acima, através

[1] Estes resultados foram computados com base na sugestão da lista de usar `pct` $= 0.1$.

de `VFinder_Iterated` e `VFinder_Accelerated`, resolvi o problema com uma sequência de grids de 100, 250 e, finalmente, 500 pontos. O tempo de execução foi de 0.58 segundos, isto é, mais de 10x mais rápido do que os outros dois métodos!

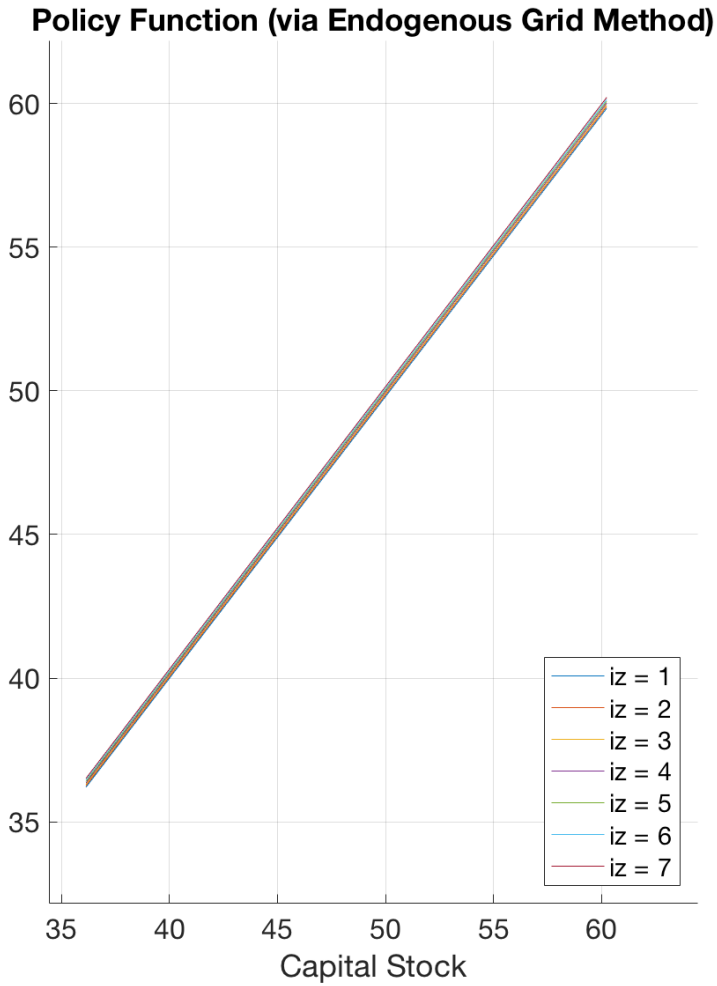


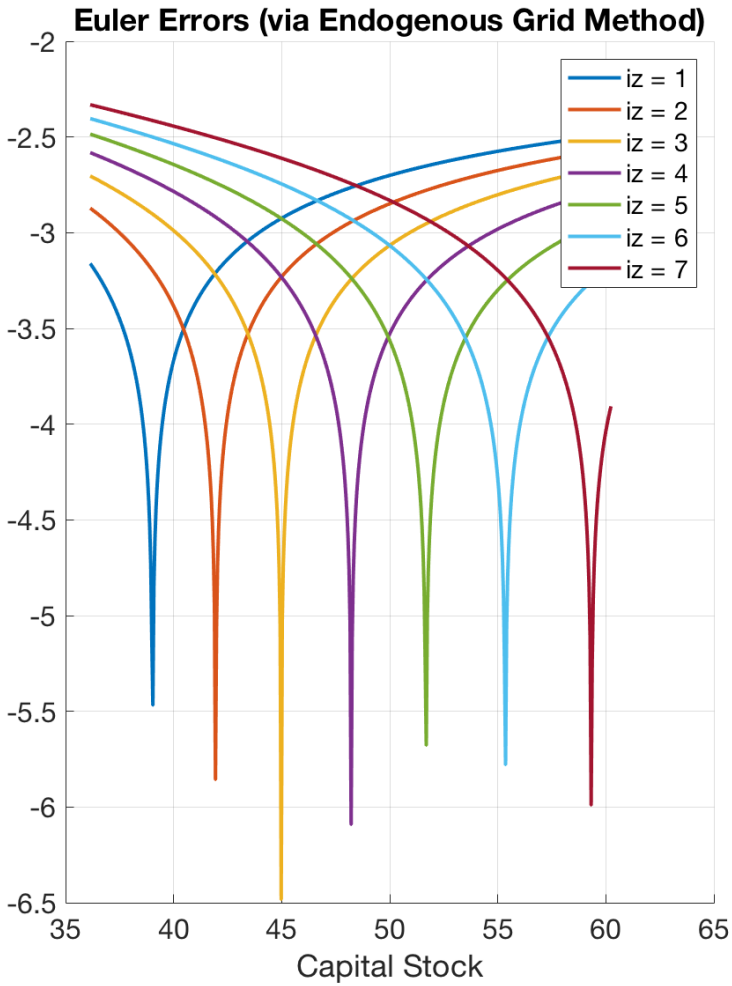
Problema 6. O método do grid endógeno pareceu promissor, ainda que tenha tido performance um pouco pior do que o multigrid. O algoritmo convergiu em 1.66 segundos, o que representa uma significativa melhora com respeito ao método do acelerador e da força bruta, no entanto.

Como pode ser visto a partir da Equação de Euler derivada no Problema 2, dado o grid exógeno para k' e um chute para a função política do consumo, não podemos inverter o lado esquerdo diretamente e encontrar o valor de k_{egm} do grid endógeno pois a função $zk^{\alpha} + (1-\delta)k$ não admite inversa analítica trivial em k, ponto a ponto em z. A saída que encontrei foi inverter $m(k) = zk^{\alpha} + (1-\delta)k$ numericamente, dado um valor de z. Fiz isto através da função `polyfit` do MATLAB, usando um polinômio de grau 4.

Uma vez que não há teoremas gerais com respeito à convergência do método do grid endógeno, computei os Euler Errors para checar a qualidade da aproximação. O erro médio foi de -3.033, o que considero satisfatório. A figura a seguir mostra a função política do capital e os Euler Errors encontrados.

A qualidade da aproximação é tanto melhor quanto mais próximo se estiver do capital de estado estacionário sem incerteza, para cada z fixo. De fato, nos pontos de maior precisão, os Euler Errors ficaram ao redor de -6, valor considerado satisfatório pelos slides. Nos pontos em que a aproximação se mostrou pior, os Euler Errors se mantiveram ao redor de -2.5. Levando em conta a escala logarítmica, a interpretação nestes casos é de que estamos errando o valor do consumo ótimo em \$1 a cada $10^{-2.5} \approx$ \$316 gastos, um erro de aproximadamente 0.32%.


Policy Function (via Endogenous Grid Method)
60
55
50
45
40
35
35
40
45
50
55
60
Capital Stock
iz = 1
iz = 2
iz = 3
iz = 4
iz = 5
iz = 6
iz = 7


Euler Errors (via Endogenous Grid Method)
-2
-2.5
-3
-3.5
-4
-4.5
-5
-5.5
-6
-6.5
35
40
45
50
55
60
65
Capital Stock
iz = 1
iz = 2
iz = 3
iz = 4
iz = 5
iz = 6
iz = 7