An Advanced Course On The Science and Art of DSGE Modelling

Day 3: Classical Heterogeneous Agent Models

Szabolcs Deák

Kirill Shakhnov

Hyungseok Joo

September 2021

Aggregate Uncertainty

Introducing Aggregate Uncertainty

- We allow for aggregate uncertainty: this is the setup studied by Krusell and Smith (1998).
- Aggregate uncertainty:
  - Idiosyncratic risk is *not* the only source of uncertainty
  - Aggregate variables (including the wealth distribution!) are *not* constant
- Wealth distribution not only matters for aggregate demand, but it is a time-varying argument now.

Model Summary

The model consists of the household's maximization problem

$$V(k,\varepsilon,\lambda,A) = \max_{k'} U(Rk + \varepsilon W - k') + \beta E\{V(k',\varepsilon',\lambda',A')|\varepsilon,A\} \tag{14}$$

$$\text{s.t.} \quad \lambda' = \Gamma(\lambda, A, A') \tag{15}$$

$$k' \geq \underline{k}, \tag{16}$$

the firm's first order conditions

$$W = \alpha A^{\alpha} \left(\frac{K}{H}\right)^{1-\alpha} \tag{17}$$

$$R = (1-\alpha) A^{\alpha} \left(\frac{K}{H}\right)^{-\alpha} + 1 - \delta, \tag{18}$$

and the market clearing conditions

$$K' = \int k' d\lambda'(k',\varepsilon) \tag{19}$$

$$H = E\varepsilon \tag{20}$$

A Generic Solution Algorithm

- We need to operationalize the steps from our generic algorithm:

  1. Guess the law of motion Γ.
  2. Solve for the optimal individual decision rules given the law of motion.
  3. Aggregate the individual decisions in order to determine the aggregate demand functions.
  4. If the resulting aggregate demand functions are consistent with our guess about Γ, then we have found the solution.
  5. Otherwise update our guess. Rinse and repeat.

Guessing the Law of Motion

Guessing the Law of Motion

- Can we approach the problem the same way as we did the Aiyagari model?
- Suppose we try to solve the Aiyagari model, but with a *time varying* wealth distribution.
- The model's state variables would be: k, ε, and $\lambda(k, \varepsilon)$.
- Recall from Day 1: every time-varying state variable requires a grid
- How many gridpoints do we need?

Guessing the Law of Motion

- How many gridpoints do we need?
  - The shock ε can take 2 values: 2 gridpoints
  - Individual capital k can take n_k values: $2 * n_k$ gridpoints
  - We have $n_k \times 2$ household types: $\lambda(k_1, \varepsilon_1)$, ..., $\lambda(k_n, \varepsilon_2)$
  - Suppose $\lambda(k_1, \varepsilon_1)$ can take n_λ values: $2 * n_k * n_\lambda$ gridpoints
  - Suppose $\lambda(k_2, \varepsilon_1)$ can take n_λ values: $2 * n_k * n_\lambda * n_\lambda$ gridpoints

    $\vdots$
  - The size of the state space: $2 * n_k * \underbrace{n_\lambda * \ldots * n_\lambda}_{2n_k - 1 \text{ times}} = 4 n_k n_\lambda^{2n_k - 1}$

Guessing the Law of Motion

- How many gridpoints do we need?
  - The size of the state space: $2 * n_k * \underbrace{n_\lambda * \ldots * n_\lambda}_{2n_k - 1 \text{ times}} = 4 n_k n_\lambda^{2n_k - 1}$
  - The code that solves the Aiyagari model evaluates $340,249.5$ gridpoints each second on my laptop.
  - With $n_k = n_\lambda = 10$ we can evaluate the model once (i.e. one iteration!) every 18,639,141.5 years on my laptop.
  - And having $n_k = n_\lambda = 100$ would still be a very coarse grid...
  - Note: normally k has a continuous domain so $n_k \to \infty$. Not only the number of gridpoints is infinite, but also the dimensionality of the problem.

Guessing the Law of Motion

- We need to reduce the dimensionality of the problem.
- KS assume that agents do not use the entire distribution when predicting future prices. They only use a limited set of statistics.
- The agents' perceived law of motion takes only the first n_m moments as arguments instead of the whole distribution:

$$\lambda' = \Gamma(\mathbf{m}, A, A') \tag{39}$$

where $\mathbf{m} = \{\mathbf{m}_1, \mathbf{m}_2, \ldots, \mathbf{m}_{n_m}\}$ denotes the set of the first n_m moments of the wealth distribution.

Guessing the Law of Motion

- Example:
  - Suppose that λ is normal with mean μ_t and variance σ_t^K.
  - Assume that we know the exact dynamics of these two (conditional) moments.
  - Then we could characterize the dynamics of the entire distribution using the dynamics of the conditional mean and the conditional variance only.
  - This would correspond to $n_m = 2$, $\mathbf{m} = \{\mu_{t-1}, \sigma_{t-1}^K\}$ and $\lambda' = \Gamma(\mu_{t-1}, \sigma_{t-1}^K, A, A')$.

Guessing the Law of Motion

- So the state vector contains k, ε, A, and the first n_m moments.

- Note: the aggregate capital stock, the first-moment of the wealth distribution, is always among the state variables which allows us to compute factor prices R and W.

- KS experiment with different functional forms and values for n_m in their paper. They find that

$$\log K' = a_0(A) + a_1(A) \log K$$

  provide an extremely good fit despite of its simplicity.

- Both a_0 and a_1 depend on the business cycle.

Guessing the Law of Motion

- Important to emphasize: one moment is not always enough
  - KS find that it is enough in their model, but they check other functional forms as well.
  - Reiter (2010a) finds that he needs 8 moments in his model:
    - the tax rate on capital is a source of aggregate uncertainty
    - the tax rate itself is common to all agents
    - taxation is a random redistribution of wealth from the rich to the poor

A Generic Solution Algorithm

- We need to operationalize the steps from our generic algorithm:

  1. Guess the law of motion Γ.
  2. Solve for the optimal individual decision rules given the law of motion.
  3. Aggregate the individual decisions in order to determine the aggregate demand functions.
  4. If the resulting aggregate demand functions are consistent with our guess about Γ, then we have found the solution.
  5. Otherwise update our guess. Rinse and repeat.

Optimal Individual Decision Rules

Optimal Individual Decision Rules

- The maximization problem we need to solve:

$$
\begin{aligned}
V(k,\varepsilon,\lambda,A) = \max_{k'} U(Rk + \varepsilon W - k') + \\
\beta E\{V(k',\varepsilon',\lambda',A')|\varepsilon, A\} \\
\text{s.t.} \quad \log K' = a_0(A) + a_1(A)\log K \\
k' \geq \underline{k}
\end{aligned}
$$

- Let us specify the exogenous processes.

Optimal Individual Decision Rules

- Assume aggregate productivity is a 2-state Markov chain:
  - It can take only two values: $A_2 > A_1$
  - We will refer to these two states as: *good* times and *bad* and denote them by g and b
  - The transition probabilities of these two realizations are defined by the 2×2 matrix $\mathcal{P}^A$.

Optimal Individual Decision Rules

- Assume individual productivity is a 2-state Markov chain:
  - It can take only two values: $\varepsilon_2 > \varepsilon_1$
  - We will refer to agents in these states as: *employed* and *unemployed* and denote them by 1 and 0
  - The transition probabilities of these two realizations are defined by the 2×2 matrices $\mathcal{P}^{\varepsilon}(A, A')$.
  - The aggregate and idiosyncratic shocks are correlated: the transition probabilities of the idiosyncratic shock depend on the business cycle (but not the other way around).

Optimal Individual Decision Rules

- We can construct a joint process z:
  - It has 4 states: $(g, 1)$, $(g, 0)$, $(b, 1)$ and $(b, 0)$
  - The transition probabilities of these four realizations are defined by the 4×4 matrix $\mathcal{P}^z$
  - $\mathcal{P}^z$ can be constructed from $\mathcal{P}^A$ and $\mathcal{P}^\varepsilon(A, A')$

Guessing the Law of Motion

- The maximization problem can be written as:

$$V(k, K, z_j) = \max_{k'} U(R(K, z_j)k + \varepsilon(z_j)W(K, z_j) - k') + Q(k', K', z_j) \tag{41}$$

$$\text{s.t.} \quad Q(k', K', z_j) = \beta \sum_{m=1}^{4} \mathcal{P}^z_{mj} V(k', K', z_m) \tag{42}$$

$$\log K' = \begin{cases} a_0^b + a_1^b \log K & \text{if } A(z_j) = A_1 \\ a_0^g + a_1^g \log K & \text{if } A(z_j) = A_2 \end{cases} \tag{43}$$

$$k' \geq \underline{k} \tag{44}$$

- We use value function iteration

- Individual capital: n_k-point grid

$$\{\underbrace{k_1 = \underline{k}}_{\text{borrowing constraint}}, k_2, \ldots, k_{n_k-1}, \underbrace{k_{n_k} = \overline{k}}_{\text{suitably chosen upper bound}}\}$$

Optimal Individual Decision Rules

- Aggregate capital: n_K-point grid

$$\{\underbrace{K_1}_{\text{suitably chosen lower bound}}, K_2, \ldots, K_{n_K-1}, \underbrace{K_{n_K}}_{\text{suitably chosen upper bound}}\}$$

- We restrict the households to choose one of the n_k gridpoints as their capital holding tomorrow, k'.

- Typically little curvature along the K dimension:

  - We do not restrict K' to be a gridpoint: linear interpolation

$$V(k, K'z) = \omega V(k, K_i, z) + (1-\omega)V(k, K_{i+1}, z)) \tag{63}$$

  - Typically $n_K \ll n_k$

- The solution is the decision rule $k' = g(k, K, z)$.

A Generic Solution Algorithm

- We need to operationalize the steps from our generic algorithm:

  1. Guess the law of motion Γ.
  2. Solve for the optimal individual decision rules given the law of motion.
  3. Aggregate the individual decisions in order to determine the aggregate demand functions.
  4. If the resulting aggregate demand functions are consistent with our guess about Γ, then we have found the solution.
  5. Otherwise update our guess. Rinse and repeat.

Aggregation

Aggregation

- We use simulation to aggregate individual decisions:
  - We simulate a panel for a large number of agents (N) over a large number of periods (T)
  - Aggregate capital is simply given by

$$K_t = \frac{1}{N} \sum_{i=1}^{N} k_{i,t} \qquad t = 1, \ldots, T \tag{54}$$

- Generated shocks:
  - We have to make sure there is no sampling uncertainty.
  - We use the same set of shocks for every simulation.

A Generic Solution Algorithm

- We need to operationalize the steps from our generic algorithm:

  1. Guess the law of motion Γ.
  2. Solve for the optimal individual decision rules given the law of motion.
  3. Aggregate the individual decisions in order to determine the aggregate demand functions.
  4. If the resulting aggregate demand functions are consistent with our guess about Γ, then we have found the solution.
  5. Otherwise update our guess. Rinse and repeat.

Evaluating our Guess

Evaluating our Guess

- Once we have simulated the model:
  - We estimate the actual law of motion based on the simulated time series for aggregate capital
  - For the subsample of bad time periods ($A_t = 1$):

$$\log K_t = \hat{a}_0^b + \hat{a}_1^b \log K_{t-1} \tag{47}$$

  - For the subsample of good time periods ($A_t = 2$):

$$\log K_t = \hat{a}_0^g + \hat{a}_1^g \log K_{t-1} \tag{48}$$

- The actual and perceived laws of motion are close to each other if

$$\left\| \begin{bmatrix} \hat{a}_0^g & \hat{a}_1^g & \hat{a}_0^b & \hat{a}_1^b \end{bmatrix} - \begin{bmatrix} a_0^g & a_1^g & a_0^b & a_1^b \end{bmatrix} \right\|_\infty \tag{49}$$

  is sufficiently small.

Evaluating our Guess

- We estimate the coefficients from a sample:
  - We need to check the accuracy of the law of motion carefully
  - We can use the R^2:
    - it is not clear what can be considered a low R^2
    - it can be a weak metric (see den Haan (2010a))
  - We can use the forecast error:
    - The purpose of the perceived law of motion is to predict the future values of K'
    - We can check the forecast error to see if it serves its purpose
    - KS use the 100 period forecast error: the maximum forecast error is 0.1 percent

Implementing the Algorithm

Implementing the Algorithm

- Utility function:

$$U(c) = \log c$$

- Parameter values:

Parameter	β	α	δ	$\underline{k}$	A_1	A_2	ε_1	ε_2
Value	0.99	0.64	0.025	0	0.99	1.01	0.15	0.7

$A'\varepsilon' \setminus A\varepsilon$	$(b,0)$	$(b,1)$	$(g,0)$	$(g,1)$
$(b,0)$	0.5250	0.0389	0.0938	0.0091
$(b,1)$	0.3500	0.8361	0.0313	0.1159
$(g,1)$	0.0313	0.0021	0.2917	0.0243
$(g,1)$	0.0938	0.1229	0.5833	0.8507

Implementing the algorithm

```
%% Parameters
% Model parameters
p.beta                  = 0.99;           % discount factor
p.alpha                 = 0.64;           % labour elasticity of output
p.delta                 = 0.025;          % depreciation rate

% Aggregate shock
p.mean_length_good      = 8;              % expected duration of good times
p.mean_length_bad       = 8;              % expected duration of bad times
p.A                     = [0.99 1.01];    % constant aggregate productivity
p.N_A                   = length(p.A);    % number of grid points along A dimension

% Idiosyncratic shock
p.fraction_u_good       = 0.04;           % unemployment ratio during good times
p.mean_length_u_good    = 1.5;            % expected duration of unemployment during good times
p.fraction_u_bad        = 0.1;            % unemployment ratio during bad times
p.mean_length_u_bad     = 2.5;            % expected duration of unemployment during bad times
p.cond_prob_1           = 1.25;           % conditional probability of staying unemployed: recession vs trough
p.cond_prob_2           = 0.75;           % conditional probability of staying unemployed: expansion vs peak
p.e                     = [0.15 0.7];     % productivity values in the two states
p.N_e                   = length(p.e);    % number of grid points along e dimension
ergdist_e_g             = [p.fraction_u_good 1-p.fraction_u_good];
                                          % stationary distribution of workers during good times
ergdist_e_b             = [p.fraction_u_bad 1-p.fraction_u_bad];
                                          % stationary distribution of workers during bad times
```

Implementing the algorithm

```
% Aggregate variables
p.H                   = p.e*[ergdist_e_b; ergdist_e_g]';
                                          % aggregate labour in the two aggregate states

% Solution method
p.N_K                 = 3;                % aggregate capital grid size
p.K_min               = 23;               % minimum value of aggregate capital
p.K_max               = 27;               % maximum value of aggregate capital
a_b                   = [0 1]';           % perceived ALM: log K' = a(1) + a(2)*log K
a_g                   = [0 1]';           % initial guess: a=[0 1], i.e. K is perceived to remain constant

p.T                   = 3500;             % sample size to be generated during simulation
p.T_burn              = 500;              % observations at the beginning of the sample to be dropped
p.N                   = 30000;            % number of agents to be simulated

p.N_k                 = 2000;             % individual capital grid size
p.k_min               = 0;                % minimum value of individual capital holding
p.k_max               = 100;              % maximum value of individual capital holding

p.update_weight       = 0.75;             % weight on current values during iteration

p.tol_num             = 1e-4;             % tolerance parameter for number of agents
p.tol_alm             = 1e-5;             % tolerance parameter for aggregate law of motion
p.tol_vfi             = 1e-6;             % tolerance parameter for value function iteration
p.tol_dist0           = p.tol_alm*100;    % tolerance parameter for starting wealth distribution
```

Implementing the algorithm

```
%% Initialization
% Constructing the transition matrix
targets                = [p.mean_length_good p.mean_length_bad...
                          p.fraction_u_good p.mean_length_u_good...
                          p.fraction_u_bad p.mean_length_u_bad...
                          p.cond_prob_1 p.cond_prob_2];          % calibration targets

[tran_prob, P_A, P_e] = tran_prob_KS(targets);                   % transition matrix in terms of probabilities
N                      = tran_number_of_agents_KS(p,P_e,targets); % transition matrix in terms of number of agents

% Capital spaces
k                      = linspace(p.k_min,p.k_max,p.N_k)';       % individual
K                      = linspace(p.K_min,p.K_max,p.N_K)';       % aggregate
p.K                    = ((1/p.beta-(1-p.delta))/(1-p.alpha))^(-1/p.alpha)*mean(p.H);
                                                                 % steady-state capital stock at mean labour supply

% Generating shocks
[shocks_a, shocks_e]   = Draw_shocks_KS(p,P_A,N);                % generating aggregate and individual shocks;
shocks_a=p.A(shocks_a);
shocks_e=p.e(shocks_e);
good_times             = shocks_a(p.T_burn+2:end)==p.A(2);       % good times dummy
and_bad                = shocks_a(p.T_burn+2:end)==p.A(1);       % bad times dummy

% Allocating memory for/Construct matrices used within the main loop
pol_func               = zeros(p.N_k,p.N_K,p.N_e,p.N_A);         % decision rule
V                      = zeros(p.N_k,p.N_K,p.N_e,p.N_A);         % value function; also our initial guess for V
```

Implementing the algorithm

```
%% Let's party
iter       = 0;                                                          % iteration counter
keep_going = true;                                                       % exit flag

k0         = interp1(k,k,p.K,'nearest')*ones(1,p.N);                     % initial guess: uniform
% main iteration
while keep_going
    iter                      = iter+1;

    % Obtaining decision rule
    [V, pol_func, iter_V]     = Individual_KS(p, tran_prob, K, k, a_g, a_b, V); % decision rule (indices)
    % Note: we also carry V over to the next iteration. It'll be our initial guess.

    % Obtaining time seris for aggregate capital by simulating the model
    [K_actual, k1, k_sim_max] = Simulate_model_KS(p, pol_func, k, K, k0, shocks_a, shocks_e);

    % Regression
    Y                         = log(K_actual(p.T_burn+3:end));                  % log K'
    X                         = [ones(size(Y)) log(K_actual(p.T_burn+2:end-1))]; % [1 log K]
    [a_g1,~,~,~,statsg]       = regress(Y(good_times),X(good_times,:));         % good times
    [a_b1,~,~,~,statsb]       = regress(Y(and_bad),X(and_bad,:));               % bad times

    change_in_coeffs          = max(abs([a_g-a_g1; a_b-a_b1]));
    keep_going                = change_in_coeffs>p.tol_alm;

    % updating initial cross-section: only if we're still far from convergence
    if change_in_coeffs>p.tol_dist0
        k0                    = k1;
    end

    fprintf('Iteration %03d: VFI iterations: %03d, minmax K: %2.2f %2.2f max k(i): %2.1f%% (%03d times)\n',
    fprintf('                good times coeffs: %1.3f->%1.3f   %1.3f->%1.3f  R2:  %1.6f\n',a_g(1),a_g1(1),a_
    fprintf('                bad times coeffs:  %1.3f->%1.3f   %1.3f->%1.3f  R2:  %1.6f\n',a_b(1),a_b1(1),a_
    fprintf('                change in coeffs: %1.2e\n',change_in_coeffs);

    % updating coefficients
    a_b = p.update_weight*a_b + (1-p.update_weight)*a_b1;
    a_g = p.update_weight*a_g + (1-p.update_weight)*a_g1;
end
```