## Overall Structure

The overall idea of the computational procedure is to represent the economy as one large system of functional equations, with the unknown functions being the choice variables of the household problems and the asset prices. We then use "policy function iteration" to compute the policy and price functions that depend on the state variables. The key steps are solving of a system on nonlinear equations at each point in the state space given a guess for future policy and price functions, and then updating these approximated functions based on the newly found solution. Iterate on this until convergence. For more theoretical background on this, see e.g. Kuebler and Schmedders (2003). A variant of this method is often referred to as "parameterized expectations".

**Creating Model Configuration** The script "`main_setup`" is used to set model parameters, which are specified in the script "`experdef`". `experdef` specifies a set of baseline parameters, and allows to define additional numerical experiments by specifying deviation from the baseline. `main_setup` reads a experiment definition from `experdef` and computes the non-stochastic steady state of the economy. It then calculates an initial guess for policies and prices based on the steady state values. It writes the result into a file that serves as input for the actual computation script. The unknown functions required for the computation, that need to be set to the initial guess, are approximated by multivariate linear interpolation. The code creates objects for these functions based on the `+grid` library. The library allows the approximation basis to be switched to e.g. polynomials without affecting the rest of the code. However, for the current example, multivariate linear interpolation works best. Also, `main_setup` invokes Matlab's symbolic math toolbox to derive the analytical Jacobian for the system of equations, evaluated at the specific parameter values of the given experiment. It generates a Jacobian function that is passed to the nonlinear equation solver at computation time.

**Running the Computation** Script "`main_execute`" reads the contents of the file created by "`main_setup`", and starts the computation. It either runs until convergence or until it hits the maximum number of iterations. `main_execute` contains several functions that are called during the policy function iteration. Effectively, the code loops over all points in the state space, and at each point invokes the nonlinear equation solver 'fsolve'.

The actual model computation is in the functions 'calcStateTransition' and 'calcEquations' (these are located as individual files in the main directory of the code). The first method's inputs are the current state and choice variables. It uses budget constraints and market clearing conditions, and evaluates the transition functions to calculate the state variables for the next period. It is useful to separate this part of the code, since it is also needed to simulate the model after the solution has been calculated. The second method uses current choices/prices, and the values of next-period state variables as inputs to compute the equilibrium equations (first-order conditions and complementary slackness conditions of constraints). Note that these functions are called at each point in the state space by the nonlinear equation solver 'fsolve' many times. If you are unsure how nonlinear equation solvers work, look at the Matlab documentation for 'fsolve' (or generally research how "Newton's method" works for finding roots of systems of nonlinear equations).

**Simulating the Results** Once a computation has converged, "`sim_stationary`" can be used to simulate the economy for many periods and compute moments of the variables. The script also calculates Euler equation errors along the simulated path. It then calls the function 'computeSimulationMoments' that contains model-specific post-processing of simulation output. Finally, it writes simulation results into several Excel spreadsheets, and a Matlab file. The output of "`sim_stationary`" in turn is used as input for "`sim_trans`", which computes impulse response functions or transition dynamics. `sim_trans` first reads the ergodic distribution computed by `sim_stationary`, then initializes simulation of many short model paths (unlike `sim_stationary`, which performs the simulation of a single long model path). To get IRFs, `sim_trans` simulates these paths for different initial exogenous productivity states – the "impulses". `sim_trans` then writes the result into yet another Matlab file. To actually make IRF graphs, `plot_trans` reads the output of `sim_trans`, and plots graphs for the selected model variables.

## Details

**Approximation Basis** The most important part of any numerical solution procedure is the technique used to approximate the unknown functions. Since we are looking to use derivative-based optimization methods, we generally want to use twice-differentiable functions. Polynomials are easy and fast to use, but dont work well for certain types of functions due to their lack of shape

preservation (e.g., theory requires certain function to be strictly increasing and convex, but once approximated by a polynomial, this is no longer guaranteed). In practice, if there are enough grid points, piecewise linear approximation on a tensor product of univariate grids for multiple dimensions is usually fast and reliable. The library +grid implements different kinds of approximations transparent to the evaluating function calls. I did not include all approximation methods I have used in the library, since they are not yet all coded in a "user-friendly" way.

**Algorithm Implementation** First, define bounds for the endogenous state variables. We are approximating the system in a hypercube, hoping that it is stationary within these bounds. For the current model, the two endogenous state variables are farmers' wealth share and capital. Choose the kind of approximation, e.g. piecewise linear. Then create an initial guess of the policy, transition functions and other functions required to compute expectations in Euler equations, usually called "forecasting" functions (these are often, but not always, a subset of the policy functions). Using this guess, the algorithm proceeds as follows:

1. Loop over all individual points in the state space, and solve a nonlinear system of N equations (FOCs and constraints) in N unknowns (choices, prices, and Lagrange multipliers), using last guess to compute expectations. Save the solution vector at each point.
2. Update policy and price functions using new solution. This becomes the next guess.
3. If distance between this new guess and last guess is below a certain threshold, stop. Otherwise go back to step 1.

**Transition Functions for Endogenous State Variables** Depending on the specifics of the problem, it may not be possible to compute next period's state variables in closed-form form only based today's state variables, choices, and prices. For example, wealth next period may depend on next period's prices, which in turn are a function of next period's wealth. However, if the problem has a first-order Markov structure in its state variables (which is usually the case for the kind of problems we consider), then we can always define a mapping from today's state variables into tomorrow's states, and approximate these transition functions in the same way as the policy or price functions.

**Handling Portfolio Constraints** To find the global solution to a problem with occasionally binding constraints, we explicitly include the constraints as functional equations and the Lagrange multipliers as choice variables. An effective way to do this involves rewriting the inequality constraints as equations by defining transformations of the Lagrange multipliers. As a simple example, consider the following constraint for some choice variable $b$,

$$b \geq 0.$$

The Lagrange multiplier may appear in some optimality condition, e.g. an Euler equation

$$q = \lambda + \mathrm{E}[\cdot],$$

with the associated complementary slackness condition

$$\lambda b = 0.$$

We can instead define variables $\tilde{\lambda}^+$, $\tilde{\lambda}^-$ and $x$ such that

$$\tilde{\lambda}^+(x) = max\{0, x\}$$
$$\tilde{\lambda}^-(x) = max\{0, -x\}$$

We can then rewrite the original inequality constraint by combining it with the complementary slackness condition as

$$b - \tilde{\lambda}^-(x) = 0,$$

and the optimality condition becomes

$$q = \tilde{\lambda}^+(x) + \mathrm{E}[\cdot].$$

See Judd, Kuebler, and Schmedders (2002) for details. The method was first used by Garcia and Zangwill.

**Challenges**

- Finding the right bounds for the state variables: It is often difficult to know good bounds of the endogenous state variables ex ante. E.g. for the wealth share of farmers, we know that it

must be between 0 and 1, but it is most likely unnecessary to approximate the system on this whole range because in the stationary equilibrium it stays contained in a much smaller subset. In addition, due to risk premia in the stochastic model, state variables are not necessarily centered around their nonstochastic steady state values.

- Finding a good initial guess: If the initial guess is not good, the nonlinear equation solver (fsolve in MATLAB) may not be able to solve the system at some points in the state space. This is true in particular for systems with many constraints, that may or may not be binding in different combinations. Here it is hence useful to have the solver try different combinations of binding constraints in case it gets stuck.

- To further deal with this problem, there is also a method that revisits failed points (where the solver could not find a solution) at the end of each iteration. The method tries to solve these points again, but not using last iteration's solution as guess. Instead, we use as guess the solution of the closest successfully solved point from the current iteration.