Overview of the Code for the Extended BabyMAKRO

Red blocks outline a script and grey-filled boxes outline a section within the script. ☆ indicates I have further explained the section on the next page. Green Text refers to implementation of Government, pink refers to HtM, and Turkish refers to export rigidity.

`Baby-MAKRO.py`

Purpose: Create a Class Based on "EconModel" Library
Local imports: `blocks.py`, `steady_state.py` &

Settings

Purpose: Setup the fundamental settings

Table of Content:
Namespace: par, ss, ini, sol
Blocks (from `blocks.py`)
Variable List

- [Exogenous, Unknowns, Targets, Non-households, Households]

Setup

Purpose: Setup the "free" parameters

Table of Content:
Parameters

- [Time, Households, Firms, Government, Agencies, Foreign, Matching, bargaining]

Allocate

Purpose: All variables are allocated

Table of Content:
Allocate

- Setattr(object, name, value)
  - For non-households var or households var
- Exogenous
- Unknowns
- Targets

Steady State

Purpose: Find steady state using `steady_state.py`

Table of Content:
Find Steady State

- Using $m_{ss}^{S} = 0.50$

Set Functions

Purpose: Settings values to ss, setting unknowns to a value based on incremental change in Jacobian, and getting errors in target equation

Table of Content:
Set variables to steady state

- Households and non-household variables

Set exogenous to steady state
Set unknown ss
Set unknown
Get_errors (target equation errors)

Evaluate

Purpose: Evaluate if the current block in fact is the block

Table of Content:
Evaluate Block

- Set initial condition = SS ← always for our type of analysis

IRFs ☆

Purpose: Analyze the propagation of shocks to the economy.

Table of Content:
Calculating the Jacobian
Find IRF

- Using `broyden_Solver`.py
- *See explanation on the next page*

`blocks.py`

Purpose: Breaking the economy down in blocks

Auxiliary Functions

Purpose: Basically, it's used to ease the notation in the code section "Blocks"

Table of Content:
Lead
Lag
CES functions

Blocks

Purpose: Setup the blocks

Table of Content:
Blocks

- [Households' Search, Labor Agency, Production Firm, Bargaining, Repacking Firm - Prices, Foreign Economy, Capital Agency, Government, Households' Consumption, Repacking Firms – Components, Goods Market Clearing]

`steady_state.py`

Purpose: Solving for steady state with output being "ss.[varname]".
Local import: `blocks.py`

Household Behavior in Steady State ☆

Purpose: Setting up the behavior of the households as stated in in the mathematical steady state section in babyMAKRO and then use it to find optimal bequest in the next code section.

Table of content

- *See explanation on the next page*

Find Steady State

Purpose: Solving for steady state values

Table of Content:
Same structure as the mathematical solution to steady state

- However, we use root finder to solve the previous code section! ☆

`broyden_solver.py`

Purpose: Solving the equation system allowing us to analyze IRFs

Check Convergence

Purpose: Ensuring convergence to a solution.

Broyden Solver ☆

Purpose: Numerical equation system solver using Broyden's method.

Table of Content:

- *See explanation on the next page*

I recommend that you go through the bullets below alongside the code in the relevant `.py` files.

Explaining how we find steady state solution for households

The objective is to find Bq such that the households ' target equation holds.

- Looking at the function "household_ss()" in `steady_state.py` we initially guess that Bq is ss.Bq
- Then we find the consumption using final savings (when one is A-1 years old, i.e., the last year of ones life) and Euler
- Afterwards, we find the implied savings
- …And aggregate consumption and savings
- The function "household_ss()" returns our target equation stated as $B_{ss}^{q} - B_{A-1,ss}$ instead of $B_{ss}^{q} = B_{A-1,ss}$ as it is done in the math

Now interestingly – but not surprisingly - we know that if we find the root of $B_{ss}^{q} - B_{A-1,ss} = 0$ we have solved the math problem $B_{ss}^{q} = B_{A-1,ss}$.
With that in mind we can call a root.optimizer using Sciåy (a scientific library in Python).
Turning to section "g. household behavior" in the next function "find_ss()" we find the root by:

- Inputting an initial guess on household_ss, defining a space we want to search in here we say [0.1;100], and then providing arguments for the parameters and steady state to help the search
- Now we can input the solution (root) to our problem $B_{ss}^{q} - B_{A-1,ss} = 0$ in the function "household_ss()" from before
- Thus, we have found the ss.Bq that ensures $B_{ss}^{q} = B_{A-1,ss}$.

Explaining the section: IRFs

First, we calculate the Jacobian around steady state. This is done through the following steps:

- Find base (remember we want to make an incremental change away from our base, steady state)
  - We calculate target errors in target equations (the 5 equations!), which has dimensions (5*500,0[1])
- Built x_ss with size that fits the Jacobian (i.e., it must have size (5*500,0), such that we can impose a change to all entries in the Jacobian for every column
  - x_ss is all the unknowns
- Allocate space for Jacobian (remember, the Jacobian is just a gradient. I.e. the matrix of what the implied errors are in matrix of Target Equations when we change x_ss)
- Calculate Jacobian (This calculation is done for all 2.500 columns using a *for loop*):
  - Create Copy of x_ss called x, so that we can keep ss values in our x_ss
  - Add an incremental change to x_ss, such that $x = x_{ss} + \epsilon$ is not x_ss. In other words, x values deviate a tiny bit ($\epsilon = 1e-4$) from ss values!
  - Evaluate alternative target equation errors using x, i.e., the implied errors when changing our column with 2.500 unknowns a tiny bit
  - Calculate numerical derivative as $\frac{(alternative\ errors - base\ errors)}{\Delta x}$ in unknown variables. Hence, we have update column i in our Jacobian
  - Repeat the process above for all 2.500 columns!

Now we turn to `Find_IRF`:

- We initially guess that our x's are the ss
- Our objective is the errors in the target equation. If target errors are within tolerance close to zero, then we are back in equilibrium. The equation system of our target equation errors is solved with the method, Broyden!
- Now let's turn to the handwritten Broyden solver implemented in `broyden_solver.py`:
  - **We initially guess on the solution by guessing that the target errors are in ss**
  - **Then we evaluate that guess using the function check_convergence from `broyden_solver.py` to see if it is zero within some tolerance level set to 1e-8.**
  - **Next, if initial guess – not surprisingly – is different from zero, we iterate over max 100 computations:**
    - **Update x**
    - **Evaluate function**
    - **If converged return x.**
    - **Update Jacobian**
    - **Repeat the steps until converged**

The Broyden's method can be stated more formally as follows (own creation!☺). Note, subscripts are not yet correct:

A.Initial guess (here we are applying Newton's method)

$$\boldsymbol{x_n = x_{n-1} - J_n^{-1} f_n}$$

Evaluate if function is zero

$$\boldsymbol{f_n(x) = 0}$$

using tolerance criteria `np.max(np.abs(y)) < tol =1e-8`

B. iterate

1. If $\boldsymbol{f_n(x) \neq 0}$ given some tolerance update x:
   $$\boldsymbol{\Delta x_n = -J_n^{-1} f_n \Leftrightarrow J dx = -f_n}$$
   has converged using `np.linalg.solve(Jac,-y)`
2. Evaluate if function:
   $$\boldsymbol{f_n(x + \Delta x) = 0}$$
   using tolerance criteria `np.max(np.abs(y)) < tol =1e-8`
3. Update Jac
   $$\boldsymbol{\Delta f_n = f_n - f_{n-1}}$$
   $$\boldsymbol{J_n = J_{n-1} + \frac{\Delta f_n - J_{n-1}\Delta x_n}{\|\Delta x_n\|^2}\Delta x_n^T}$$

note we take the outer product on second term on the r.h.s using `np.outer(((dy - jac @ dx) / np.linalg.norm(dx)**2), dx)`

Now set $\boldsymbol{f_n = f_{n-1}}$ & $\boldsymbol{x = x + \Delta x}$ and repeat Steps from B.1 to B.3 until tolerance criteria is satisfied.

[1] I intentionally write 0 and not 1, as NumPy writes dimensions of a vector with a 0 and not 1.