The GEModelTools Package for Solving HANK Models in Python

Jeppe Druedahl
Emil Holst Partsch

Abstract

This note provides an overview of the `GEModelTools` package for solving general equilibrium models easily in Python. The package is specifically designed for finding the non-linear transition path to a MIT-shock.

You can learn how to use the package following these steps:

1. Read this document
2. Install the package
3. Run the example notebooks
4. Read the commented code for the example notebooks
5. Implement your own model

Literature: Boppart et al. (2018) and Auclert et al. (2021).

Structure:

Section 1 explains the required user inputs and available methods

Section 2 shows an example with a simple Aiyagari-model

Section 3 explains additional (and upcoming) features

Section 4 provides basic troubleshooting

Code:

Package: github.com/JeppeDruedahl/GEModelTools
Notebooks: github.com/JeppeDruedahl/GEModelToolsNotebooks

Requirements: Rely on `EconModel` and `ConSav`.

Packages:
github.com/NumEconCopenhagen/EconModel
github.com/NumEconCopenhagen/ConsumptionSaving

Notebooks:
github.com/NumEconCopenhagen/EconModelNotebooks
github.com/NumEconCopenhagen/ConsumptionSavingNotebooks

1 Overview

The central tool in `GEModelTools` is the `GEModelClass`, which is an add-on to the basic `EconModelClass` (documented here). An example of the setup is shown in Listing 1. The three methods `.settings()`, `.setup()` and `.allocate()` are all called automatically when the model is created.

A `model` of the `GEModelClass` consists of the following list of `namespaces`:

1. **Parameters:** `.par`
2. **Solution:** `.sol`
3. **Simulation:** `.sim`
4. **Steady state:** `.ss`
5. **Transition path:** `.path`
6. **Jacobians for household problem:** `.jac_hh`
7. **Full Jacobians:** `.jac`

The user is required to specify some **variable lists** in `.settings()` for:

1. **Aggregate variables:** `.varlist`. Used as `path.VARNAME`.
2. **Household variables:** `.varlist_hh`
   Used as `sol.VARNAME` and `sol.path_VARNAME`
   Remark I: `i` and `w` are used for saving simulation indices and weights.
   Remark II: `path_i` and `path_w` are used for saving simulation indices and weights.
3. **Household grids:** `.grid_hh`. Used as `par.VARNAME_grid`.
4. **Household policy functions:** `.pols_hh`. Should be in `.varlist_hh`.
5. **Household inputs:** `.inputs_hh`. Should be in `.varlist`.
6. **Household outputs:** `.outputs_hh`. Should be in `.varlist_hh`.
7. **Exogenous inputs:** `.inputs_exo`. Should be in `.varlist`.
8. **Endogenous inputs:** `.inputs_endo`. Should be in `.varlist`.
9. **Targets:** `.targets`. Should be in `.varlist`.

The user must choose the following **settings** in `.setup()` :

1. **Number of exogenous states:** `par.Nz`
2. **Number of grid points:** `par.Nendo1`, `par.Nendo2`,...
   where `endo1`, `endo2`,..., is in `.grids_hh`
3. **Length of transition period:** `par.transition_T`
4. **For each exogenous input:**
   Initial jump: `par.jump_VARNAME`
   Persistence: `par.rho_VARNAME`
5. **Optional solver settings:**
   `par.max_iter_solve`, `par.max_iter_simulate`, `par.max_iter_broyden`
   `par.tol_solve`, `par.tol_simulate`, `par.tol_broyden`

In the `.allocate()` method the internal **method** `.allocate_GE(sol_shape)` can now be called to allocate:

1. **Exogenous grids and transition matrices:**
   `par.z_grid_ss`, `shape=(par.Nz,)`
   `par.z_trans_ss`, `shape=(par.Nz,)`
   `par.z_ergodic_ss`, `shape=(par.Nz,)`
   `par.z_grid_path`, `shape=(par.transition_T,par.Nz)`
   `par.z_trans_path`, `shape=(par.transition_T,par.Nz,par.Nz)`

   Remark:
   `path.z_trans_path[t]` is the transition matrix from $t-1$ to t,
   and can depend on variables in period t.
2. **Distribution:**
   `sim.D`, `shape=sol_shape`
   `sim.path_D`, `shape=(par.transition_T,*sol_shape)`
3. **All variables in** `.sol`
   `sol.VARNAME`, `shape=sol_shape`
   `sol.path_VARNAME`, `shape=(par.transition_T,*sol_shape)`
4. **All variables in** `.path`
   `path.VARNAME`,
   `shape=(par.transition_T,len(inputs_endo)×par.transtion_T)`
   `ss.VARNAME`, scalar

   Remark:
   `path.VARNAME[0,t]` is the value in period t.
   `path.VARNAME[i,t]` for $i > 0$ should be considered *undefined behavior*.
5. **All variables in** `.jac_hh`
   `jac_hh.OUTPUTNAME.upper()_INPUTNAME`,
   `shape=(par.transition_T,par.transition_T)`

Finally, the user must also provide the following **functions**:

1. `grids.py` must contain `create_grids(model)`, which creates the grids for the endogenous variables and the grids and transition matrices for the exogenous variable. *This is called each time we solve for the steady state of the household problem.*
2. `household_problem.py` must contain the jitted[1] functions:
   `solve_hh_ss(par,sol,ss)`, finds `sol.VARNAME` for variables in `.outputs_hh`.
   `solve_hh_path(par,sol,ss,path)`, finds `sol.path_VARNAME` for all in `.outputs_hh`.
3. `find_strady_state.py` must contain the function `find_ss(model,do_print)`, which fills `ss`, and solve and simulate the household problem in steady state.

[1] The function should be decorated with `@numba.njit`.

4. `transition_path.py` must contain the jitted function `evaluate_path(par,sol,sim,ss,path,jac_hh,threads,use_jac_hh)`, where

   i) `use_jac_hh` is a boolean for whether or not to use the household Jacobians when evaluating household behavior (used when calculating the full Jacobian).

   ii) `threads` is 1 or `len(inputs_endo)`×`par.transtion_T`, where the latter is used to evaluate the path in parallel for all one step changes.

   iii) If `threads` > 1 then `use_jac_hh` should be `True`.

The following internal methods are now available:

1. `.solve_ss()`: Solve household problem in steady state → `sol.VARNAME`.
2. `.simulate_ss()`: Simulate household problem in steady state → `sim.D`.
3. `.solve_path()`: Solve household problem → `sol.path_VARNAME`
4. `.simulate_path()`: Simulate household problem → `sim.path_D`.
5. `.compute_jac_hh()`: Compute the household Jacobians → `jac_hh`.
6. `.compute_jac()`: Compute the full Jacobian → `jac`.
7. `.find_transition_path()`: Find transition path → `path`.

To solve and simulate household behavior in a jitted function use:

```
household_problem.solve_hh_path(par,sol,path)
prepare_simulation_1d_1d(par,sol,sol.path_endo1,par.endo1_grid)
simulate_hh_path(par,sol,sim)
```

```
1
2 from EconModel import EconModelClass
3 from GEModelTools import GEModelClass
4
5 class MyModelClass(EconModelClass,GEModelClass):
6
7     def settings(self):
8         """ fundamental settings """
9
10         self.grids_hh = [] # grids
11         self.pols_hh = [] # policy functions
12         self.inputs_hh = [] # inputs to household problem
13         self.outputs_hh = [] # output of household problem
14         self.varlist_hh = [] # variables in household problem
15         self.inputs_exo = [] # exogenous inputs
16         self.inputs_endo = [] # endogenous inputs
17         self.targets = [] # targets
18         self.varlist = [] # all variables
19
20     def setup(self):
21         """ set baseline parameters """
22
23         par = self.par
24         par.NVARNAME = 100 # number of grid points
25         par.jump_VARNAME = -0.01 # initial jump in %
26         par.rho_VARNAME = 0.8 # AR(1) coefficeint
27         par.transition_T = 500 # length of path
28
29     def allocate(self):
30         """ allocate model """
31
32         par = self.par
33         sol_shape = (par.Nfix,par.Nz,par.Nendo1)
34         self.allocate_GE(sol_shape)
35
```

Listing 1: Example: Setup

2 Solving the Aiyagari-model

In this section, we explain the non-linear sequence space solution method implemented in the package for a simple Aiygari-model.

2.1 Aiyagari-model

Overview. There is a continuum of measure one households who

1. Own stocks, a_{t-1} (measured end-of-period)
2. Supply labor with productivity z_t (exogenous and stochastic)

$$\begin{aligned} z_t &= \rho z_{t-1} + \varepsilon_t^z. \\ \mathbb{E}[z_t] &= 1. \\ \text{Var}[\varepsilon_t^z] &= \sigma_z^2. \end{aligned} \tag{1}$$

3. Consume, c_t

Firms rent capital, K_{t-1}, and hire labor, L_t, to produce goods,

$$Y_t = Z_t K_{t-1}^{\alpha} L_t^{1-\alpha}, \tag{2}$$

where Z_t is technology. Capital depreciates with the rate δ.
Both households are firms are **price takers** and

1. r_t^k is the (real) rental rate for capital
2. $r_t = r_t^k - \delta$ is the implied (real) interest rate
3. w_t is the (real) wage rate

Firms. Firms maximize profits implying the standard pricing equations

$$r_t^k = \alpha Z_t \left(K_{t-1}/L_t\right)^{\alpha\text{-}\mathbf{1}} \equiv r^k(Z_t, K_{t-1}, L_t). \tag{3}$$

$$\begin{aligned} w_t &= (1-\alpha) Z_t (K_{t-1}/L_t)^{\alpha} \\ &= (1-\alpha) Z_t \left(\frac{r_t^k}{\alpha Z_t}\right)^{\frac{\alpha}{\alpha-1}} \equiv w(r_t^k, Z_t). \end{aligned} \tag{4}$$

Households. Households have *perfect foresight* wrt. to the interest rate and the wage rate, $\{r_t, w_t\}_{t=0}^{\infty}$, and solve the problem

$$\begin{aligned} v_t(z_t, a_{t-1}) &= \max_{c_t} \frac{c_t^{1-\sigma}}{1-\sigma} + \beta \mathbb{E}_t \left[v_{t+1}(z_{t+1}, a_t) \right] \\ &\text{s.t.} \\ a_t + c_t &= (1 + r_t) a_{t-1} + w_t z_t \\ z_{t+1} &\sim \Gamma_z(z_t) \\ a_t &\geq 0, \end{aligned}$$

where

$$v_t \left(z_t, a_{t-1} \right) = v \left(z_t, a_{t-1}; \{r_\tau, w_\tau\}_{\tau=t}^{\infty} \right).$$

The FOC is $c_t^{-\sigma} = \beta \mathbb{E}_t[v_{a,t+1}]$ and the envelope condition is $v_{a,t} = (1+r_t)c_t^{-\sigma}$. The optimal saving and consumption functions $a_t^*(a_{t-1}, z_t)$ and $c_t^*(a_{t-1}, z_t)$ can be found using e.g. the EGM. These optimal policy functions live on the discretized grids, $z_t \in \{z^0, \ldots, z^{\#_z - 1}\}$ and $a_t \in \{a^0, \ldots, a^{\#_a - 1}\}$. Labor productivity, z_t, is discretized with the Rouwenhorst-method. In `grids.py` the user must provide the function `create_grids(model)` to setup the grids and transition matrices.

The user should provide two jiited functions in `household_problem.py`:

1. `solve_hh_ss(par,sol,ss)`: Solve for $a_{ss}^*(\bullet)$ and $c_{ss}^*(\bullet)$ in `sol.a` and `sol.c`.
2. `solve_hh_path(par,sol,path)`: Solve for $\{a_t^*(\bullet)\}_{t=0}^{T-1}$ and $\{c_t^*(\bullet)\}_{t=0}^{T-1}$ for arbitrary sequences $\{r_t, w_t\}_{t=0}^{T}$, where $a_T^*(\bullet) = a_{ss}^*(\bullet)$ and $c_T^*(\bullet) = c_{ss}^*(\bullet)$, in `sol.path_a` and `sol.path_c`.

Distribution. Let D_t be the distribution of households over z_t and a_{t-1}. The supply of capital then is

$$\mathcal{K}_t = \int a_t^*(a_{t-1}, z_t) dD_t = \int a_t dD_{t+1}. \tag{5}$$

The household problem implies a time-varying but non-stochastic law of motion for D_t,

$$D_t = \Gamma_D(D_{t-1}; a_{t-1}^*, \Gamma_z). \tag{6}$$

In practice, this simulation problem is generic. It is beneficial to use a simulation method, where households are always on the grid. The idea here is to re-distribute mass to grid points based on optimal decision. More precisely we calculate

$$\begin{aligned} D_{t+1}(e^k, a^l) = & \\ & \sum_{i=0}^{\#_e - 1} \Pr[e^k | e^i] \sum_{j=0}^{\#_a - 1} D_t(e^i, a^j) \omega(a_t^*(e^i, a^j), a^{\max\{l-1,0\}}, a^l, a^{\min\{l+1,\#_a-1\}}), \end{aligned} \tag{7}$$

where ω is a weight calculated using linear interpolation

$$\omega(a, \underline{a}, \tilde{a}, \overline{a}) = 1\{a \in [\underline{a}, \overline{a}]\} \begin{cases} \frac{\overline{a}-a}{\overline{a}-\tilde{a}} & \text{if } a \geq \tilde{a} \\ \frac{a-\underline{a}}{\tilde{a}-\underline{a}} & \text{if } a < \tilde{a} \end{cases}.$$

Extension to higher dimensions are straightforward. This is provided in the package as the methods `.simulate_hh_ss()` and `.simulate_hh_path()`. The indices and weights required for simulation is calculated automatically when calling the methods `.solve_hh_ss()` and `.solve_hh_path()`.

Market clearing. Market clearing requires

$$\begin{aligned} \text{Capital: } K_t = \mathcal{K}_t &= \int a_t dD_{t+1} = \int a_t^*(z_t, a_{t-1}) dD_t \\ \text{Labour: } L_t &= \int e_t dD_t = 1 \\ \text{Goods: } Y_t &= \int c_t^*(z_t, a_{t-1}) dD_t + K_t - K_{t-1} + \delta K_{t-1} \end{aligned}$$

2.2 Stationary equilibrium

A **stationary equilibrium** for a given Z_{ss} is one where

1. Quantities K_{ss} and L_{ss},
2. prices r_{ss} and w_{ss},
3. a distribution D_{ss} over a_{t-1} and z_t
4. and policy functions $a_{ss}^*(z_t, a_{t-1})$ and $c_{ss}^*(z_t, a_{t-1})$

are such that

1. $a_{ss}^*(\bullet)$ and $c_{ss}^*(\bullet)$ solves the household problem with $\{r_{ss}, w_{ss}\}_{t=0}^{\infty}$
2. D_{ss} is the invariant distribution implied by the household problem
3. Firms maximize profits, $r_{ss} = r(Z_{ss}, K_{ss}, L_{ss})$ and $w_{ss} = w(r_{ss}, Z_{ss})$
4. The labor market clears, i.e. $L_{ss} = \int e_t dD_{ss} = 1$
5. The capital market clears, i.e. $K_{ss} = \int a_{ss}^*(z_t, a_{t-1}) dD_{ss}$
6. The goods market clears, i.e. $Y_{ss} = \int c_{ss}^*(z_t, a_{t-1}) dD_{ss} + \delta K_{ss}$

We can **find the stationary equilibrium** by solving a root-finding problem

1. Guess on r_{ss}

2. Calculate $w_{ss} = w(r_{ss}, Z_{ss})$
3. Solve the infinite horizon household problem
4. Simulate until convergence of D_{ss}
5. Calculate supply $\mathcal{K}_{ss} = \int a^*_{ss}(z_t, a_{t-1}) dD_{ss}$
6. Calculate demand $K_{ss} = \left(\frac{r_{ss}+\delta}{\alpha Z_{ss}}\right)^{\frac{1}{\alpha-1}} L_{ss}$
7. If for some tolerance ϵ

$$|\mathcal{K}_{ss} - K_{ss}| < \epsilon$$

then stop, otherwise update r_{ss} appropriately and return to step 2

In `find_strady_state.py` the user must supply the function `find_ss(model,do_print)` to solve the problem. In practice we guess on r_{ss} and w_{ss} and derive Z_{ss} and δ_{ss} from the implied household behavior.

2.3 Transition path

A **transition path** for $t \in \{0, 1, 2, \dots\}$, given an initial distribution D_0 and a path of Z_t, is paths of quantities K_t and L_t, prices r_t and w_t, policy functions $a^*_t(\bullet)$ and $c^*_t(\bullet)$, distributions D_t, such that for all t

1. $a^*_t(\bullet)$ and $c^*_t(\bullet)$ solve the household problem given price paths
2. D_t are implied by the household problem given price paths and D_0
3. Firms maximizes profit, $r_t = r(Z_t, K_{t-1}, L_t)$ and $w_t = w(r_t, Z_t)$
4. The labor market clears, i.e. $L_t = \int z_t dD_t = 1$
5. The capital market clears, i.e. $K_{t-1} = \int a_{t-1} dD_t$
6. The goods market clears, i.e. $Y_t = \int c^*_t(\bullet) dD_t + K_t - K_{t-1} + \delta K_{t-1}$

This is also called an MIT-shock ≡ »shock in a world without shocks«.

In practice we consider a *truncated* transition path of length T, and everything is back to steady state afterwards.

2.4 Sequence space method

We can think of the model in terms of inputs are targets:

1. **1 exogenous input:** $\{Z_t\}_{t=0}^{T-1}$
2. **1 endogenous input:** $\{K_t\}_{t=0}^{T-1}$

3. **1 target:** Asset market clearing

The model is then captured by the equation system

$$\begin{aligned}\boldsymbol{H}(\{K_t, Z_t\}_{t=0}^T) = \mathbf{0} \Leftrightarrow \\ \left[\begin{array}{c} \text{Asset market clearing} \end{array}\right] = \mathbf{0} \\ \left[\begin{array}{c} K_t - \mathcal{K}_t \end{array}\right] = \left[\begin{array}{c} 0 \end{array}\right] \\ \forall t \in \{0, 1, \ldots, T-1\}\end{aligned}$$

where we have

$$\begin{aligned}L_t &= 1 \\ r_t &= \alpha Z_t (K_{t-1}/L_t)^{\alpha-1} \\ w_t &= (1-\alpha) Z_t \left(\frac{r_t + \delta}{\alpha Z_t}\right)^{\frac{\alpha}{\alpha-1}} \\ D_t &= \Gamma_{t-1,D}(D_{t-1}), \forall t > 0 \\ \mathcal{K}_t &= \int a_t^*(z_t, a_{t-1}) dD_t \\ K_{-1} &= K_{ss} \\ D_0 &= \Gamma_D(D_{ss}; a_{ss}^*, \Gamma_z) = D_{ss}\end{aligned}$$

In `transition_path.py` the user must supply the jitted function `evaluate_path(...)`, which given the inputs updates the value of all targets.

Jacobian. Defining $\boldsymbol{K} = (K_0, K_1, \ldots)$ and $\boldsymbol{Z} = (Z_0, Z_1, \ldots)$ we can write the equation system in time-stacked form

$$\boldsymbol{H}(\boldsymbol{K}, \boldsymbol{Z}) = \mathbf{0} \tag{8}$$

Total differentiation implies

$$\boldsymbol{H}_{\boldsymbol{K}} d\boldsymbol{K} + \boldsymbol{H}_{\boldsymbol{Z}} d\boldsymbol{Z} = 0 \Leftrightarrow d\boldsymbol{K} = -\boldsymbol{H}_{\boldsymbol{K}}^{-1} \boldsymbol{H}_{\boldsymbol{Z}} d\boldsymbol{Z} \tag{9}$$

where

$$\boldsymbol{H}_{\boldsymbol{K}} = \begin{bmatrix} \frac{\partial H_0}{\partial K_0} & \frac{\partial H_0}{\partial K_1} & \cdots \\ \frac{\partial H_1}{\partial K_0} & \ddots & \ddots \\ \vdots & \ddots & \ddots \end{bmatrix}, \boldsymbol{H}_{\boldsymbol{Z}} = \begin{bmatrix} \frac{\partial H_0}{\partial Z_0} & \frac{\partial H_0}{\partial Z_1} & \cdots \\ \frac{\partial H_1}{\partial Z_0} & \ddots & \ddots \\ \vdots & \ddots & \ddots \end{bmatrix}$$

and

$$\boldsymbol{H_K} = \mathcal{J}^{\mathcal{K},r}\mathcal{J}^{r,K} + \mathcal{J}^{\mathcal{K},w}\mathcal{J}^{w,K} - \boldsymbol{I} \tag{10}$$

$$\boldsymbol{H}_Z = \mathcal{J}^{\mathcal{K},r}\mathcal{J}^{r,Z} + \mathcal{J}^{\mathcal{K},w}\mathcal{J}^{w,Z} \tag{11}$$

where generically

$$\mathcal{J}^{x,y} = \begin{bmatrix} \frac{\partial x_0}{\partial y_0} & \frac{\partial x_0}{\partial y_1} & \cdots \\ \frac{\partial x_1}{\partial y_0} & \ddots & \ddots \\ \vdots & \ddots & \ddots \end{bmatrix}$$

Once the Jacobian $\boldsymbol{H_K}$, also referred to as "the full Jacobian", is calculated, the equation system can be solved with a quasi-Newton equation solver such as the Broyden-solver for an arbitrary path of technology, Z_t, returning to steady state. This is provided in the package with the method `.find_transition_path()`.

In the package there are two methods which both needs to be run to calculate the Jacobian:

1. `.compute_jac_hh()`: Compute the Jacobians of household problem, namely $\mathcal{J}^{\mathcal{K},r}$ and $\mathcal{J}^{\mathcal{K},w}$ using the fast fast news algorithm (see below).
2. `.compute_jac()`: Compute the full Jacobian using numerical differentiation relying on the Jacobians of the household problem through the chain-rule as in eq. (10).

2.5 Fake news algorithm

The fake news algorithm is explained in Auclert et al. (2021). The algorithm firstly uses that the policy functions only depend on the time span relative to when the shocks arrive. Therefore it is only necessary to find the policy functions solving backwards along the transition path one single time. The more surprising result is that it is only necessary to simulate forwards along the transition path one single time. The intuition is not straightforward, but as the algorithm is fully generic it can just be relied on as a black box.

Consider the following notation:

1. **Productivity:** z_t, indexed by i, lives on $\mathcal{G}_z = \{z^0, z^1, \ldots, z^{\#_z - 1}\}$ with transition matrix Π^e_{t+1} with elements

$$\pi^z_{t+1,[i,i+]} = \Pr[z_{t+1} = z^{i+1} | z_t = z^i].$$

2. **Assets:** a_t, indexed by j, lives on $\mathcal{G}_a = \{a^0, a^1, \ldots, a^{\#_a - 1}\}$.

3. **Value and policy functions:** $\boldsymbol{v}$, $\boldsymbol{a}^*$ and $\boldsymbol{c}^*$ lives on $\mathcal{G}_z \times \mathcal{G}_a$ with

$$\boldsymbol{v}_{[i,j]} = u(\boldsymbol{c}^*_{[i,j]}) + \sum_{j_+=0}^{\#_a - 1} \boldsymbol{P}^i_{[j,j_+]} \beta \sum_{k=0}^{\#_z - 1} \pi^e_{t+1,[i,i_+]} v_{[i_+ j_+]},$$

where $\boldsymbol{c}^*_{[i,j]} = c^*(z_i, a_j)$ and $\boldsymbol{P}^i_{[j,k]}$ are the weights implied by linear interpolation of $a^*(z_t, a_{t-1})$ at $\boldsymbol{a}^*_{[i,j]} = a^*(z_i, a_j)$ given by

$$\boldsymbol{P}^i_{[j,k]} = \begin{cases} \frac{a^*_{ij} - a^{j_+-1}}{a^{j_+} - a^{j_+-1}} & \text{if } j_+ > 0, \text{and } a^*_{ij} \in [a^{j_+-1}, a^{j_+}] \\ \frac{a^*_{ij} - a^{j_+}}{a^{j_++1} - a^{j_+}} & \text{if } j_+ < \#_a - 1, \text{and } a^*_{ij} \in [a^{j_+}, a^{j_++1}] \\ 0 & \text{else} \end{cases}.$$

Let $\overrightarrow{x}$ be the row-stacked version of the matrix x. The Bellman equation can be written

$$\overrightarrow{\boldsymbol{v}}_{\boldsymbol{t}} = u(\overrightarrow{\boldsymbol{c}}^*_t) + \beta \boldsymbol{P}_t \tilde{\Pi}^e_{t+1} \overrightarrow{\boldsymbol{v}}_{t+1} m \tag{12}$$

where $\tilde{\Pi}^e_{t+1} = \Pi^e_{t+1} \otimes \boldsymbol{I}_{\#_a \times \#_a}$ and P_t is the policy matrix given by

$$\boldsymbol{P}_t = \begin{bmatrix} \boldsymbol{P}^0_t & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \ddots & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{P}^{\#_e - 1}_t \end{bmatrix}, P^i_t = \begin{bmatrix} \ddots & \vdots & \iddots \\ \cdots & q^i_{[j,j_+]} & \cdots \\ \iddots & \vdots & \ddots \end{bmatrix}. \tag{13}$$

Simulation is now the inverse operation:

$$\overrightarrow{D}_{t+1} = \tilde{\Pi}^{e\prime}_{t+1} \boldsymbol{P}'_t \overrightarrow{D}_t, \tag{14}$$

where $'$ denoted transpose. The fake new algorithm now is:

Step 1: Solve backwards $T-1$ periods from a shock Δ_x to price x.
$\boldsymbol{a}^{*,x}_s$ is the optimal saving policy with s periods until shock arrival
$\boldsymbol{P}^x_s$ is the associated policy matrix
$\tilde{\Pi}^{e\prime}_s$ is transition matrix from $s-1$ to s

Step 2: *Numerical derivatives,*

$$\Delta^s_{D,x} = \frac{\tilde{\Pi}^{e\prime}_{ss} \boldsymbol{P}^{x\prime}_s \overrightarrow{D}_{ini} - \overrightarrow{D}_{ss}}{\Delta_x}, \Delta^s_{a,x} = \frac{\overrightarrow{\boldsymbol{a}}^{*,x\prime}_s \overrightarrow{D}_{ini} - \overrightarrow{\boldsymbol{a}^*_{ss}}^{*,\prime} \overrightarrow{D}_{ss}}{\Delta_x}$$

where $\overrightarrow{D}_{ini} = \tilde{\Pi}^{e\prime}_s \left(\tilde{\Pi}^{e\prime}_{ss}\right)^{-1} \overrightarrow{D}_{ss}$

Step 3: *Expectation factors,* $\mathcal{E}_t = \begin{cases} \boldsymbol{a}^*_{ss} & \text{if } t = 0 \\ \boldsymbol{P}_{ss} \tilde{\Pi}^e_{ss} \mathcal{E}_{t-1} & \text{else} \end{cases}$

Step 4: *Fake news matrix,* $\mathcal{F}^a_{[t,s]} = \begin{cases} \Delta^s_{a,x} & \text{if } t = 0 \\ \overrightarrow{\mathcal{E}}_{t-1} \Delta^s_{D,x} & \text{else} \end{cases}$

Step 5: *Jacobian,* $\mathcal{J}^{\mathcal{K},x}_{[t,s]} = \begin{cases} \mathcal{F}^a_{[t,s]} & \text{if } t = 0 \vee s = 0 \\ \sum_{k=0}^{\min\{t,s\}} \mathcal{F}^a_{[t-k,s-k]} & \text{else} \end{cases}$

3 Additional features

1. `.show_IRFS((...)`
   Show IRFs.
2. `.compare_IRFS(...)`
   Compare IRFs across models
3. `.print_unpack_varlist(...)`:
   Print unpacking of all variables for use in `.evaluate_path(...)` to screen.
4. `.test_hh_path(...)`:
   Test time-invariance when inputs are at their steady state values.
5. `.test_jac_hh(...)`:
   Compare Jacobians calculated with a direct and the fake news algorithm.
6. `.test_path(...)`:
   Test time-invariance when inputs are at their steady state values.

Upcoming features:

1. More general shock processes
2. Computation of covariance-matrices
3. Efficient computation of multiple household Jacobians
4. Interface to fast C++ code
5. Check for determinancy

4 Troubleshooting

The transition path cannot be found. Considering the following

1. Use finer tolerances for finding the steady state
   `par.tol_solve`↓
   `par.tol_simulate`↓
2. Extend the transition period
   `par.transition_T`↑
3. Decrease the size and persistence of the size
   `par.jump_VARNAME`↓
   `par.rho_VARNAME`↓
4. Change other parameters making the model more stable
   (e.g. more strict Taylor rule, less sticky prices/wages)

References

Auclert, A., Bardóczy, B., Rognlie, M., and Straub, L. (2021). Using the Sequence-Space Jacobian to Solve and Estimate Heterogeneous-Agent Models. *Econometrica*, 89(5):2375–2408.

Boppart, T., Krusell, P., and Mitman, K. (2018). Exploiting MIT shocks in heterogeneous-agent economies: the impulse response as a numerical derivative. *Journal of Economic Dynamics and Control*, 89:68–92.