econpizza

Release 0.6.3

Gregor Boehl

Apr 20, 2024

USER GUIDE

This document is automatically created by sphinx, the Python documentation generator. It is synced with the online package documentation that is hosted at Read the Docs.

CHAPTER

ONE

OVERVIEW: ECONPIZZA

Econpizza is a framework to solve and simulate *fully nonlinear* perfect foresight models, with or without heterogeneous agents. The package implements the solution method proposed in HANK on Speed: Robust Nonlinear Solutions using Automatic Differentiation *(Gregor Boehl, 2023, SSRN No. 4433585)*. It allows to specify and solve nonlinear macroeconomic models quickly in a simple, high-level fashion.

The package builds heavily on automatic differentiation via JAX. A central philosophy is to consequently separate the low-level routines for *model solution* (which is what happens under the hood) from *model specification* (via a `yaml` file) and *model analysis* (what the user does with the model).

The package can solve nonlinear models with heterogeneous households or firms with one or two assets and portfolio choice. Steady state and nonlinear impulse responses (including, e.g., the ZLB) can typically be found within a few seconds. It not only allows to study the dynamics of aggregate variables, but also the complete nonlinear transition dynamics of the cross-sectional distribution of assets and disaggregated objects. Routines for models with a representative agents are also provided. These are faster and more reliable than the extended path method in dynare due to the use of automatic differentiation for the efficient Jacobian decompositions during each Newton-step. Nonlinear perfect-foresight transition dynamics can - even for large-scale nonlinear models with several occassionally binding constraints - be computed in less than a second.

CHAPTER

TWO

HOW TO CITE

Please cite with

```
@article{boehl2023goodpizza,
    title       = {HANK on Speed: Robust Nonlinear Solutions using Automatic␣
↪Differentiation},
    author      = {Boehl, Gregor},
    journal     = {Available at SSRN 4433585},
    year        = {2023}
}
```

CHAPTER THREE

INSTALLATION

The package is under active development. Stable releases can be installed from the official Python repositories, which are updated frequently.

Installing the repository version from PyPi is as simple as typing

```
pip install econpizza
```

in your terminal or Anaconda Prompt.

The current version is supporting Python versions 3.9 to 3.11 on Windows, Mac and Linux. The changelog and release history can be found on GitHub.

CHAPTER

FOUR

SPECIFYING MODELS

Models are specified in a YAML file, which uses the YAML markup language. The YAML format is widely used due to its intuitive handling, for example for configuration files or in applications where data is being stored or transmitted which should be human readable. For general information about the format and its syntax see Wikipedia.

The YAML file contains all relevant information from model equations, variable declarations and steady state values. Models specified as a YAML files can be parsed into a *econpizza.PizzaModel* using *econpizza.parse()* or *econpizza.load()*. An instance of *econpizza.PizzaModel* holds all the relevant information and functionality of the model.

4.1 The YAML file

The YAML files follow a simple structure:

1. list all variables, parameters and shocks
2. provide the nonlinear equations. Note that each equation starts with a ~.
3. define the values of the parameters and fixed steady state values in the `steady_state` section
4. optionally provide auxiliary equations that are not directly part of the nonlinear system
5. optionally provide initial guesses for all other steady state values and parameters

I will first briefly discuss the YAML of the small scale *representative* agents NK model *from the quickstart tutorial* and then turn to a more complex HANK model. A collection of examples is provided with the package.

4.2 YAML: representative agent models

The GitHub version of the YAML file for the small scale NK model can be found here. The first block (`variables` and `shocks`) is self explanatory:

```
variables: [y, c, pi, r, rn, beta, w, chi]
shocks: [e_beta]
```

Note that it is not necessary to define shocks. You can also simply set the initial values of any (exogenous) state.

```
parameters: [ theta, psi, phi_pi, phi_y, rho, h, eta, rho_beta, chi ]
```

Use the `parameters` block to define any *parameters*. Parameters are treated the same as variables, but they are time invariant. During steady state search they are treated exactly equally. For this reason their values are provided in the `steady_state` block.

```
definitions: |
    from jax.numpy import log, maximum
```

The second block (`definitions`) defines general definitions and imports, which are available at all stages.

```
equations:
    ~ w = chi*(c - h*cLag)*y**eta  # labor supply
    ~ 1 = r*betaPrime*(c - h*cLag)/(cPrime - h*c)/piPrime  # euler equation
    ~ psi*(pi/piSS - 1)*pi/piSS = (1-theta) + theta*w + psi*betaPrime*(c-h*cLag)/(cPrime-
↪h*c)*(piPrime/piSS - 1)*piPrime/piSS*yPrime/y  # Phillips curve
    ~ c = (1-psi*(pi/piSS - 1)**2/2)*y  # market clearing
    ~ rn = (rSS*((pi/piSS)**phi_pi)*((y/yLag)**phi_y))**(1-rho)*rnLag**rho  # monetary␣
↪policy rule
    ~ r = maximum(1, rn)  # zero lower bound on nominal rates
    ~ log(beta) = (1-rho_beta)*log(betaSS) + rho_beta*log(betaLag) + e_beta  # exogenous␣
↪discount factor shock
```

`equations`. The most central part of the yaml. Here you define the model equations, which will then be parsed such that each equation prefixed by a ~ must hold. Use `xPrime` for variable *x* in *t+1* and `xLag` for *t-1*. Access steady-state values with `xSS`. You could specify a representative agent model with just stating the equations block (additional to variables). Importantly, `equations` are *not* executed subsequently but simultaneously! Note that you need one equation for each variable defined in `variables`.

```
steady_state:
    fixed_values:
        # parameters
        theta: 6.  # demand elasticity
        psi: 96  # price adjustment costs
        phi_pi: 4  # monetary policy rule coefficient #1
        phi_y: 1.5  # monetary policy rule coefficient #2
        rho: .8  # interest rate smoothing
        h: .44  # habit formation
        eta: .33  # inverse Frisch elasticity
        rho_beta: .9  # autocorrelation of discount factor shock

        # steady state values
        beta: 0.9984
        y: .33
        pi: 1.02^.25

    init_guesses: # the default initial guess is always 1.1
        chi: 6
```

Finally, the `steady_state` block allows to fix parameters and, if desired, some steady state values, and provide initial guesses for others. Note that the default initial guess for any variable/parameter not specified here will be `1.1`.

4.3 YAML: heterogeneous agent models

Let us have a look of the YAML of a hank model we will discuss in *the tutorial*. The GitHub version of the file (link) also contains exhaustive additional comments. The first line reads:

```
functions_file: 'hank_functions.py'
```

The relative path to a functions-file, which may provide additional functions. The GitHub version of the functions file for this model can be found here. In this example, the file defines the functions `transfers`, `wages`, `hh`, `labor_supply` and `hh_init`.

```
definitions: |
    from jax.numpy import log, maximum
    from econpizza.tools import percentile, jax_print
```

General definitions and imports (as above). These are available during all three stages (decisions, distributions, equations). We will use the `percentile` function to get some distributional statistics. `jax_print` is a JAX-jit-able print function that can be used during `call` stages for debugging.

```
variables: [ div, y, y_prod, w, pi, R, Rn, Rr, Rstar, tax, z, beta, C, n, B, Top10C,␣
↪Top10A ]
```

All the *aggregate* variables that are being tracked on a global level. If a variable is not listed here, you will not be able to recover it later. Since these are aggregate variables, they have dimensionality one.

```
parameters: [ sigma_c, sigma_l, theta, psi, phi_pi, phi_y, rho, rho_beta, rho_r, rho_z ]
shocks: [ e_beta, e_rstar, e_z ]
```

Define the model parameters and shocks, as above.

```
distributions:
  # the name of the first distribution
  dist:
    # ordering matters. The ordering here is corresponds to the shape of the axis of the␣
↪distribution
    # the naming of the dimensions (skills, a) is arbitrary
    skills:
      # first dimension
      type: exogenous_rouwenhorst
      rho: 0.966
      sigma: 0.6
      n: 4
    a:
      # second dimension. Endogenous distribution objects require inputs from the␣
↪decisions stage. An object named 'a' assumes that the decisions stage returns a␣
↪variable named 'a'
      type: endogenous_log
      min: 0.0
      max: 50
      n: 50
```

The distributions block. Defines a distribution (here `dist`) and all its dimensions. The information provided here is used to construct the distribution-forward-functions. If this is not supplied, econpizza assumes that you are providing a representative agent model.

Exogenous grids are grids for idiosyncratic shocks. A grid type “exogenous_rouwenhorst” requires the parameters `rho`, `sigma` and `n`. Alternatively, a grid type “exogenous_generic” only needs `n` and expects the grid variable and the transition matrix to be defined somewhere else.

Endogenous grids are grids for idiosyncratic state variables. A grid type “exogenous_log” requires the parameters `min`, `max` and `n`. Based on these, a log grid will be created. Alternatively, a grid type “endogenous_generic” only needs `n` and expects the grid variable to be defined somewhere else.

```
decisions:
  # define the multidimensional input "WaPrime", in addition to all aggregated variables␣
↪(defined in 'variables')
  inputs: [WaPrime]
  # calls executed during the decisions stage
  calls: |
    # these functions are defined in functions_file
    tfs = transfers(skills_stationary, div, tax, skills_grid)
    WaPrimeExp = skills_transition @ WaPrime
    Wa, a, c = egm_step(WaPrimeExp, a_grid, skills_grid, w, n, tfs, Rr, beta, sigma_c,␣
↪sigma_l)
  # the 'outputs' values are stored for the following stages
  outputs: [a,c]
```

The decisions block. Only relevant for heterogeneous agents models. It is important to correctly specify the dynamic inputs (here: marginals of the value function) and outputs, i.e. those variables that are needed as inputs for the distribution stage. Note that calls are evaluated one after another.

```
aux_equations: |
    # `dist` here corresponds to the dist *at the beginning of the period*
    aggr_a = jnp.sum(dist*a, axis=(0,1))
    aggr_c = jnp.sum(dist*c, axis=(0,1))
    # calculate consumption and wealth share of top-10%
    top10c = 1 - percentile(c, dist, .9)
    top10a = 1 - percentile(a, dist, .9)
```

Auxiliary equations. This again works exactly as for the representative agent model. These are executed before the `equations` block, and can be used for all sorts of definitions that you may not want to keep track of. For heterogeneous agents models, this is a good place to do aggregation. Auxiliary equations are also executed subsequently.

The distribution (`dist`) corresponds to the distribution **at the beginning of the period**, i.e. the distribution from last period. This is because the outputs of the decisions stage correspond to the asset holdings (on grid) at the beginning of the period, while the distribution calculated *from* the decision outputs holds for the next period.

```
# final/main stage: aggregate equations
equations:
    # definitions
    ~ C = aggr_c
    ~ Top10C = top10c
    ~ Top10A = top10a

    # firms
    ~ n = y_prod/z # production function
    ~ div = -w*n + (1 - psi*(pi/piSS - 1)**2/2)*y_prod # dividends
    ~ y = (1 - psi*(pi/piSS - 1)**2/2)*y_prod # "effective" output
    ~ psi*(pi/piSS - 1)*pi/piSS = (1-theta) + theta*w + psi*piPrime/R*(piPrime/piSS -␣
↪1)*piPrime/piSS*y_prodPrime/y_prod # NKPC
```

(continues on next page)

(continued from previous page)

```
    # government
    ~ tax = (Rr-1)*BLag # balanced budget
    ~ Rr = RLag/pi # real ex-post bond return
    ~ Rn = (Rstar*((pi/piSS)**phi_pi)*((y/yLag)**phi_y))**(1-rho)*RnLag**rho # MP rule␣
↪on shadow nominal rate
    ~ R = maximum(1, Rn) # ZLB

    # clearings
    ~ C = y # market clearing
    ~ B = aggr_a # bond market clearing
    ~ n**sigma_l = w # labor market clearing

    # exogenous
    ~ beta = betaSS*(betaLag/betaSS)**rho_beta*exp(e_beta) # exogenous beta
    ~ Rstar = RstarSS*(RstarLag/RstarSS)**rho_r*exp(e_rstar) # exogenous rstar
    ~ z = zSS*(zLag/zSS)**rho_z*exp(e_z) # exogenous technology
```

Equations. This also works exactly as for representative agents models.

```
steady_state:
    fixed_values:
        # parameters:
        sigma_c: 2 # intertemporal elasticity of substitution
        sigma_l: 2 # inverse Frisch elasticity of labour supply
        theta: 6. # elasticity of substitution
        psi: 60. # parameter on the costs of price adjustment
        phi_pi: 1.5 # Taylor rule coefficient on inflation
        phi_y: 0.1 # Taylor rule coefficient on output
        rho: 0.8 # persistence in (notional) nominal interest rate
        rho_beta: 0.9 # persistence of discount factor shock
        rho_r: 0.9 # persistence of MP shock
        rho_z: 0.9 # persistence of technology shocks

        # steady state
        y: 1.0 # effective output
        y_prod: 1.0 # output
        C: 1.0 # consumption
        pi: 1.0 # inflation
        beta: 0.98 # discount factor
        B: 5.6 # bond supply
        # definitions can be recursive: theta is defined above
        w: (theta-1)/theta # wages
        n: w**(1/sigma_l) # labor supply
        div: 1 - w*n # dividends
        z: y/n # technology

    init_guesses:
        Rstar: 1.002 # steady state target rage
        Rr: Rstar # steady state real rage
        Rn: Rstar # steady state notional rage
        R: Rstar # steady state nominal rage
```

(continues on next page)

(continued from previous page)

```
        tax: 0.028
        WaPrime: egm_init(a_grid, skills_stationary)
```

The steady state block. `fixed_values` are those steady state values that are fixed ex-ante. `init_guesses` are initial guesses for steady state finding. Values are defined from the top to the bottom, so it is possible to use recursive definitions, such as `n: w**frisch`.

Note that for heterogeneous agents models it is required that the initial value of inputs to the decisions-stage are given (here `WaPrime`).

Note: *Econpizza* is written in JAX, which is a machine learning framework for Python developed by Google. JAX provides automatic differentiation and just-in-time compilation ("jitting"), which makes the package fast and robust. However, running jitted JAX code brings along a few limitations. Check the common gotchas in JAX for details.

4.4 Model parsing

Models specified as a YAML files can be parsed and loaded using *`econpizza.parse()`* and *`econpizza.load()`*.

`econpizza.`**`parse`**`(`*mfile*`)`

Parse model dictionary from yaml file. This can be desirable if values should be exchanged before loading the model.

Parameters
`mfile` (*`string`*) – path to a yaml file to be parsed

Returns
mdict – the parsed yaml as a dictionary

Return type
dict

This returns a dictionary containing all the informations provided in the YAML file. Parsing before loading allows to change some features of the model manually. The dictionary can then be forwarded to *`econpizza.load()`*:

`econpizza.`**`load`**`(`*model_ref*, *raise_errors=True*, *verbose=True*`)`

Load a model from a dictionary or a YAML file.

Parameters

- **`model_ref`** (*`dict or string`*) – either a dictionary or the path to a YAML file to be parsed
- **`raise_errors`** (*`bool, optional`*) – whether to raise errors while checking. False will let the model fail siliently for debugging. Defaults to True
- **`verbose`** (*`bool, optional`*) – inform that parsing is done. Defaults to True

Returns
model – The parsed model

Return type
PizzaModel

If desired, *`econpizza.load()`* can also parse the YAML-file directly. The function then returns an instance of *`econpizza.PizzaModel`*, which holds all the relevant information and functionality of the model:

`class econpizza.`**`PizzaModel`**`(`*mdict*, **args*, ***kwargs*`)`

Base class for models. Contains all necessary methods and informations.

CHAPTER

FIVE

THE STEADY STATE

The steady state search can be evoked by calling the function *econpizza.PizzaModel.solve_stst()* documented below. The function collects all available information from `steady_state` key of the YAML and attempts to find a set of variables and parameters that satisfies the aggregate equations using the routine outlined in the paper.

Upon failure, the function tries to be as informative as possible. If the search is not successful, a possible path to find the error is to set the function's keyword argument `raise_errors` to `False`. The function then raises a warning instead of failing with an exception, and returns a dictionary containing the results from the root finding routine, such as, e.g. the last Jacobian matrix.

Note: A classic complaint is "**The Jacobian contains NaNs**". This is usually due to numerical errors somewhere along the way. While the package tries to provide more information about where the error occurred, a good idea is to follow JAX's hints on how to debug NaNs.

Tip:

- A common gotcha for heterogeneous agent models is that the distribution contains negative values. The routine will be informative about that. This is usually due to rather excessive interpolation outside the grid and can often be fixed by using a grid with larger minimum/maximum values.
- The steady state procedure is based on the pseudoinverse of the Jacobian. If the procedure fails, it will try to tell you the rank of the Jacobian and the number of degrees of freedom. More degrees of freedom than the Jacobian rank normally implies that you should fix more steady state values and vice versa.
- If the desired precision is not reached, the error message will tell you in which equation the maximum error did arise. You can use the `equations` key to get the list of equations (as strings), e.g. `print(model['equations'][17])` to get the equation with index 17.

`econpizza.PizzaModel.`**`solve_stst`**(*self*, *tol=1e-08*, *maxit=15*, *tol_backwards=None*, *maxit_backwards=2000*, *tol_forwards=None*, *maxit_forwards=5000*, *force=False*, *raise_errors=True*, *check_rank=False*, *verbose=True*, ***newton_kwargs*)

Solves for the steady state.

Parameters

- **`tol`** (*`float, optional`*) – tolerance of the Newton method, defaults to `1e-8`
- **`maxit`** (*`int, optional`*) – maximum of iterations for the Newton method, defaults to `15`
- **`tol_backwards`** (*`float, optional`*) – tolerance required for backward iteration. Defaults to `tol`
- **`maxit_backwards`** (*`int, optional`*) – maximum of iterations for the backward iteration. Defaults to `2000`

- **`tol_forwards`** (*`float, optional`*) – tolerance required for forward iteration. Defaults to `tol*1e-2`
- **`maxit_forwards`** (*`int, optional`*) – maximum of iterations for the forward iteration. Defaults to `5000`
- **`force`** (*`bool, optional`*) – force recalculation of steady state, even if it is already evaluated. Defaults to `False`
- **`raise_errors`** (*`bool, optional`*) – raise an error if Newton method does not converge. Useful for debuggin models. Defaults to `True`
- **`check_rank`** (*`bool, optional`*) – force checking the rank of the Jacobian, even if the Newton method was successful. Defualts to `False`
- **`verbose`** (*`bool, optional`*) – level of verbosity. Defaults to `True`
- **`newton_kwargs`** (*`keyword arguments`*) – keyword arguments passed on to the Newton method

Returns

rdict – a dictionary containing information about the root finding result. Note that the results are added to the model (PizzaModel instance) automatically, *rdict* is hence only useful for model debugging.

Return type

dict

CHAPTER

SIX

NONLINEAR SIMULATIONS

The main functionality of nonlinear simulations is provided by the function *econpizza.PizzaModel.find_path()* The main arguments are either `shock` or `init_state`, which allows to specify an economic shock as a tuple of the shock name (as specified in `shocks` in the YAML) and the size of the shock, or a vector of initial states, respectively. The function *econpizza.PizzaModel.get_distributions()* allows to retrieve the full nonlinear sequence of the distribution.

Note: All numerical methods are subject to numerical errors. To reduce these, you can decrease the numerical tolerance `tol`. However, this should not be below the tolerance level used for the steady state search.

Hint: A sufficient condition for convergence of the solution routine is that the generalized eigenvalues of the sequence space Jacobian and its steady-state pendant are all positive.[1] If the procedure does not converge, the `use_solid_solver=True` flag can be used to check if the model solves when using a conventional Newton method with the true Jacobian (this may take quite a while).

econpizza.PizzaModel.**find_path**(*self*, *shock=None*, *init_state=None*, *init_dist=None*, *pars=None*, *horizon=200*, *use_solid_solver=False*, *skip_jacobian=False*, *verbose=True*, *raise_errors=True*, ***newton_args*)

Find the expected trajectory given an initial state.

Parameters

- **shock** (*tuple, optional*) – shock in period 0 as in *(shock_name_as_str, shock_size)*
- **init_state** (*array, optional*) – initial state, defaults to the steady state values
- **init_dist** (*array, optional*) – initial distribution, defaults to the steady state distribution
- **pars** (*dict, optional*) – alternative parameters. Warning: do only change those parameters that are invariant to the steady state.
- **horizon** (*int, optional*) – number of periods until the system is assumed to be back in the steady state. Defaults to `200`
- **use_solid_solver** (*bool, optional*) – calculate the full jacobian and use a standard Newton method. Defaults to `False`
- **skip_jacobian** (*bool, optional*) – whether to skip the calculation of the steady state sequence space Jacobian. If True, the last cached Jacobian will be used. Defaults to `False`
- **verbose** (*bool, optional*) – degree of verbosity. `0/False` is silent. Defaults to `False`

[1] Unfortunately, this is prohibitory expensive to check as it would require to calculate the full sequence space Jacobian and its eigenvalues.

- **raise_errors** (*bool, optional*) – whether to raise errors as exceptions, or just inform about them. Defaults to `True`
- **newton_args** (*optional*) – any additional arguments to be passed on to the Newton solver (see the documentations of the solvers)

Returns

- **x** (*array*) – array of the trajectory
- **flag** (*bool*) – Error flag. Returns *False* if the solver was successful, otherwise returns *True*

If the model has heterogeneous agents, the routine will automatically compute the steady state sequence space Jacobian. This can be skipped using the `skip_jacobian` flag.

Any additional argument will be passed on to the specific Newton method. For models with heterogeneous agents this is *econpizza.utilities.newton.newton_for_jvp()*:

`econpizza.utilities.newton.`**`newton_for_jvp`**(*jvp_func*, *jacobian*, *x_init*, *verbose*, *tol=1e-08*, *maxit=20*, *nsteps=30*, *factor=1.5*)

Newton solver for heterogeneous agents models as described in the paper.

Parameters

- **tol** (*float, optional*) – tolerance of the Newton method, defaults to `1e-8`
- **maxit** (*int, optional*) – maximum of iterations for the Newton method, defaults to 20
- **nsteps** (*int, optional*) – number of function evaluations per Newton iteration, defaults to 30
- **factor** (*float, optional*) – dampening factor (gamma in the paper), Defaults to 1.5

For models with representative agents, the Newton method is *econpizza.utilities.newton.newton_for_banded_jac()*:

`econpizza.utilities.newton.`**`newton_for_banded_jac`**(*jav_func*, *nvars*, *horizon*, *X*, *shocks*, *verbose*, *maxit=30*, *tol=1e-08*)

Newton solver for representative agents models.

Parameters

- **tol** (*float, optional*) – tolerance of the Newton method, defaults to `1e-8`
- **maxit** (*int, optional*) – maximum of iterations for the Newton method, defaults to 20

If `use_solid_solver` is set to *True*, the Newton method newton_jax_jit from the grgrjax package is used.

The function *econpizza.PizzaModel.get_distributions()* allows to retrieve the sequence of distributions and decision variables. To that end it requires the shocks and initial distribution together with the trajectory of aggregated variables as input.

`econpizza.PizzaModel.`**`get_distributions`**(*self*, *trajectory*, *init_dist=None*, *shock=None*, *pars=None*)

Get all disaggregated variables for a given trajectory of aggregate variables.

Note that the output objects do, other than the result from *find_path* with stacking, not include the time-T objects and that the given distribution is as from the beginning of each period.

Parameters

- **trajectory** (*array*) – a _full_ trajectory of aggregate variables
- **init_dist** (*array, optional*) – the initial distribution. Defaults to the steady state distribution

- **`shock`** (`array, optional`) – shock in period 0 as in *(shock_name_as_str, shock_size)*. Defaults to no shock

Returns
: **rdict** – a dictionary of the distributions

Return type
: dict

CHAPTER SEVEN

UNDER THE HOOD

The functional representations of the economic model are written dynamically during parsing/loarding (in `econpizza/parser/__init__.py`).

```
[1]: import econpizza as ep
     example_hank = ep.examples.hank
```

```
[2]: mod = ep.load(example_hank)
```

```
WARNING:jax._src.xla_bridge:No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_
↪LEVEL=0 and rerun for more info.)
```

```
(load:) Parsing done.
```

The model instance is a dictionary, containing all the informations of the model. For instance, it contains the dynamically created functions as strings:

```
[3]: mod['func_strings'].keys()
```

```
[3]: dict_keys(['func_backw', 'func_eqns'])
```

The function `func_backw` corresponds to function $W(\cdot)$ from the paper and `func_eqns` is $f(\cdot)$. The other functions are static.

Lets inspect f:

```
[4]: print(mod['func_strings']['func_eqns'])
```

```
def func_eqns(XLag, X, XPrime, XSS, shocks, pars, distributions=[], decisions_
↪outputs=[]):


 (BLag, betaLag, CLag, divLag, nLag, piLag, RLag, RnLag, RrLag, RstarLag, taxLag,␣
↪Top10ALag, Top10CLag, wLag, yLag, y_prodLag, zLag, ) = XLag

 (B, beta, C, div, n, pi, R, Rn, Rr, Rstar, tax, Top10A, Top10C, w, y, y_prod, z, ) = X

 (BPrime, betaPrime, CPrime, divPrime, nPrime, piPrime, RPrime, RnPrime, RrPrime,␣
↪RstarPrime, taxPrime, Top10APrime, Top10CPrime, wPrime, yPrime, y_prodPrime, zPrime, )␣
↪= XPrime

 (BSS, betaSS, CSS, divSS, nSS, piSS, RSS, RnSS, RrSS, RstarSS, taxSS, Top10ASS,␣
↪Top10CSS, wSS, ySS, y_prodSS, zSS, ) = XSS
```

(continues on next page)

(continued from previous page)

```
(sigma_c, sigma_l, theta, psi, phi_pi, phi_y, rho, rho_beta, rho_r, rho_z, ) = pars

(e_beta, e_rstar, e_z, ) = shocks

(dist, ) = distributions

(a, c, ) = decisions_outputs

# NOTE: summing over the first two dimensions e and a, but not the time dimension␣
↪(dimension 2)
# `dist` here corresponds to the dist *at the beginning of the period*
aggr_a = jnp.sum(dist*a, axis=(0,1))
aggr_c = jnp.sum(dist*c, axis=(0,1))
# calculate consumption and wealth share of top-10%
top10c = 1 - percentile(c, dist, .9)
top10a = 1 - percentile(a, dist, .9)

root_container0 = C  - ( aggr_c)
root_container1 = Top10C  - ( top10c)
root_container2 = Top10A  - ( top10a)
root_container3 = n  - ( y_prod/z)
root_container4 = div  - ( -w*n + (1 - psi*(pi/piSS - 1)**2/2)*y_prod)
root_container5 = y  - ( (1 - psi*(pi/piSS - 1)**2/2)*y_prod)
root_container6 = psi*(pi/piSS - 1)*pi/piSS  - ( (1-theta) + theta*w + psi*piPrime/
↪R*(piPrime/piSS - 1)*piPrime/piSS*y_prodPrime/y_prod)
root_container7 = tax  - ( (Rr-1)*BLag)
root_container8 = Rr  - ( RLag/pi)
root_container9 = Rn  - ( (Rstar*((pi/piSS)**phi_pi)*((y/yLag)**phi_y))**(1-
↪rho)*RnLag**rho)
root_container10 = R  - ( maximum(1, Rn))
root_container11 = C  - ( y)
root_container12 = B  - ( aggr_a)
root_container13 = n**sigma_l  - ( w)
root_container14 = beta  - ( betaSS*(betaLag/betaSS)**rho_beta*exp(e_beta))
root_container15 = Rstar  - ( RstarSS*(RstarLag/RstarSS)**rho_r*exp(e_rstar))
root_container16 = z  - ( zSS*(zLag/zSS)**rho_z*exp(e_z))

return jnp.array([root_container0, root_container1, root_container2, root_container3,␣
↪root_container4, root_container5, root_container6, root_container7, root_container8,␣
↪root_container9, root_container10, root_container11, root_container12, root_
↪container13, root_container14, root_container15, root_container16]).T.ravel()
```

This function is then automatically compiled and the callable can be found in `model['context']`:

```
[5]: mod['context']['func_eqns']
```

```
[5]: <function econpizza.parser.func_eqns(XLag, X, XPrime, XSS, shocks, pars,␣
↪distributions=[], decisions_outputs=[])>
```

The `model['context']` itself contans the name space in which all model functions and definitions are evaluated. This may be useful for debugging:

```
[6]: mod['context'].keys()
```

```
[6]: dict_keys(['__name__', '__doc__', '__package__', '__loader__', '__spec__', '__path__', '_
↪_file__', '__cached__', '__builtins__', 'yaml', 're', 'os', 'sys', 'tempfile', 'jax',
↪'jaxlib', 'jnp', 'iu', 'deepcopy', 'copy', 'getmembers', 'isfunction', 'jax_print',
↪'het_agent_base_funcs', 'build_functions', 'write_dynamic_functions', 'func_forw_
↪generic', 'func_forw_stst_generic', 'compile_func_basics_str', 'compile_backw_func_str
↪', 'get_forw_funcs', 'compile_eqn_func_str', 'checks', 'func_pre_stst', 'check_if_
↪defined', 'check_dublicates', 'check_determinancy', 'check_initial_values', 'check_
↪shapes', 'check_if_compiled', 'grids', 'dists', 'interp', 'cached_mdicts', 'cached_
↪models', 'd2jnp', '_load_as_module', 'parse', '_eval_strs', '_parse_external_functions_
↪file', '_initialize_context', '_initialize_cache', '_load_external_functions_file', '_
↪compile_init_values', '_define_subdict_if_absent', '_define_function', '_get_pre_stst_
↪mapping', 'compile_stst_inputs', 'load', 'log', 'exp', 'sqrt', 'max', 'min', 'egm_init
↪', 'egm_step', 'interpolate', 'transfers', 'maximum', 'percentile', 'skills_grid',
↪'skills_stationary', 'skills_transition', 'a_grid', 'func_backw', 'func_forw', 'func_
↪forw_stst', 'func_eqns', 'sigma_c', 'sigma_l', 'theta', 'psi', 'phi_pi', 'phi_y', 'rho
↪', 'rho_beta', 'rho_r', 'rho_z', 'y', 'y_prod', 'C', 'pi', 'beta', 'B', 'w', 'n', 'div
↪', 'z', 'Rstar', 'Rr', 'Rn', 'R', 'tax', 'WaPrime', 'init_run'])
```

CHAPTER

EIGHT

QUICKSTART

Take a small-scale nonlinear New Keynesian model with ZLB as a starting point, which is provided as an example (find the yaml file here). Here is how to simulate it and plot nonlinear impulse responses. Start with some misc imports and load the package:

```
[1]: import matplotlib.pyplot as plt
     import econpizza as ep

     # only necessary if you run this in a jupyter notebook:
     %matplotlib inline
```

Next, load the model and solve for the steady state.

```
[2]: # the path to the example YAML
     example_nk = ep.examples.nk

     # load the NK model
     mod = ep.load(example_nk)
     _ = mod.solve_stst()
```

```
WARNING:jax._src.xla_bridge:No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_
↪LEVEL=0 and rerun for more info.)
```

```
(load:) Parsing done.
    Iteration   1 | max. error 7.89e-01 | lapsed 0.4894
    Iteration   2 | max. error 5.07e-01 | lapsed 0.5595
(solve_stst:) Steady state found (0.69372s). The solution converged.
```

Finally, set a 4% discount factor shock and simulate it:

```
[3]: # shock the discount factor by 4%
     shk = ('e_beta', .04)

     # find the nonlinear trajectory
     x, flag = mod.find_path(shock=shk)
```

```
    Iteration   1 | max error 1.71e-01 | lapsed 0.7477s
    Iteration   2 | max error 3.89e-01 | lapsed 0.7514s
    Iteration   3 | max error 2.35e-01 | lapsed 0.7544s
    Iteration   4 | max error 2.50e-01 | lapsed 0.7573s
    Iteration   5 | max error 6.81e-02 | lapsed 0.7601s
    Iteration   6 | max error 2.14e-02 | lapsed 0.7629s
    Iteration   7 | max error 5.58e-06 | lapsed 0.7656s
```

(continues on next page)

(continued from previous page)

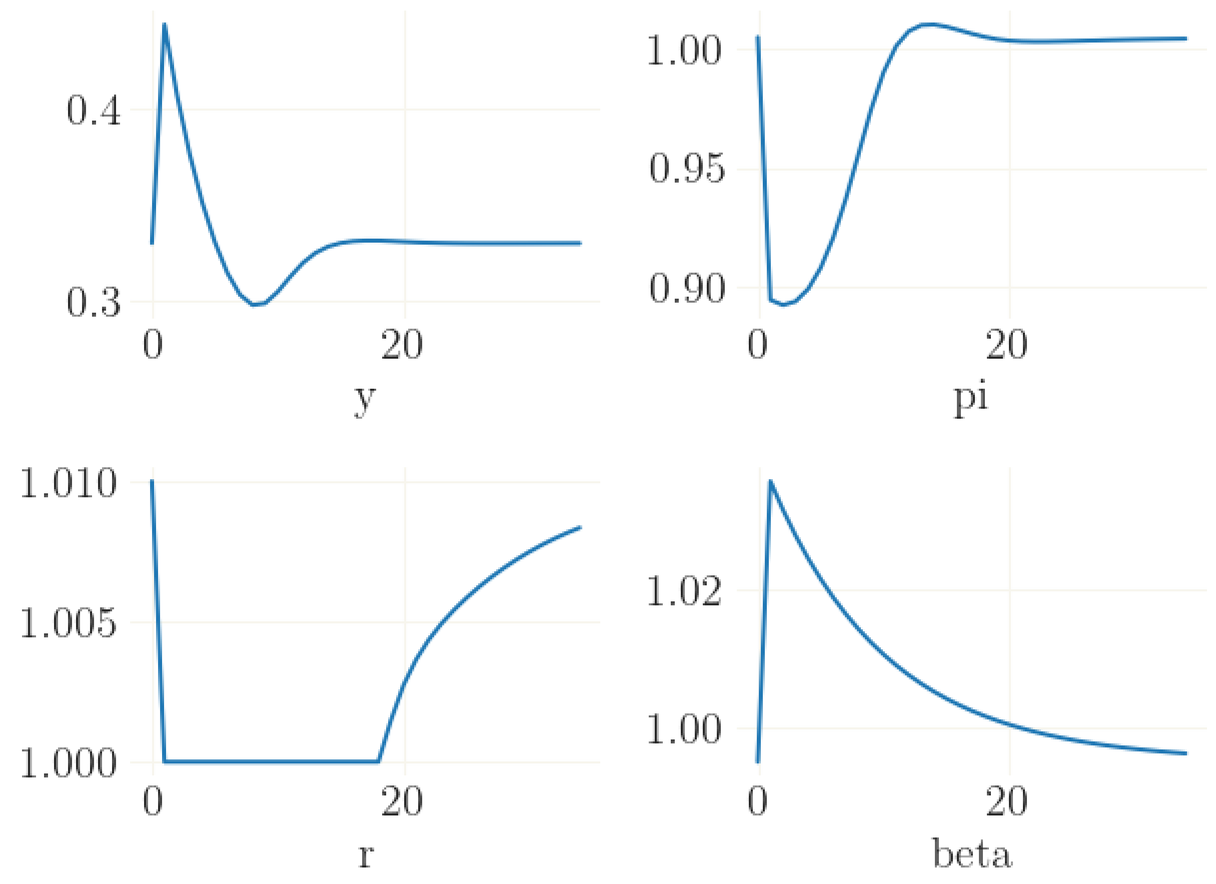
```
    Iteration   8 | max error 3.97e-12 | lapsed 0.7683s
(find_path:) Stacking done (0.886s).
```

The rest is plotting…

[4]:
```
# plotting
fig, axs = plt.subplots(2,2)
for i,v in enumerate(('y', 'pi', 'r', 'beta')):

    axs.flatten()[i].plot(x[:35,mod['variables'].index(v)])
    axs.flatten()[i].set_xlabel(v)

fig.tight_layout()
```



The impulse responses are the usual dynamics of a nonlinear DSGE model with the zero-lower bound on nominal interest rates.

Alternatively to specifying a shock, you can instead provide the initial conditions:

[5]:
```
# use the jax implementation of numpy
import jax.numpy as jnp

# get the steady state as initial condion
x0 = mod['stst'].copy()
```

(continues on next page)

(continued from previous page)

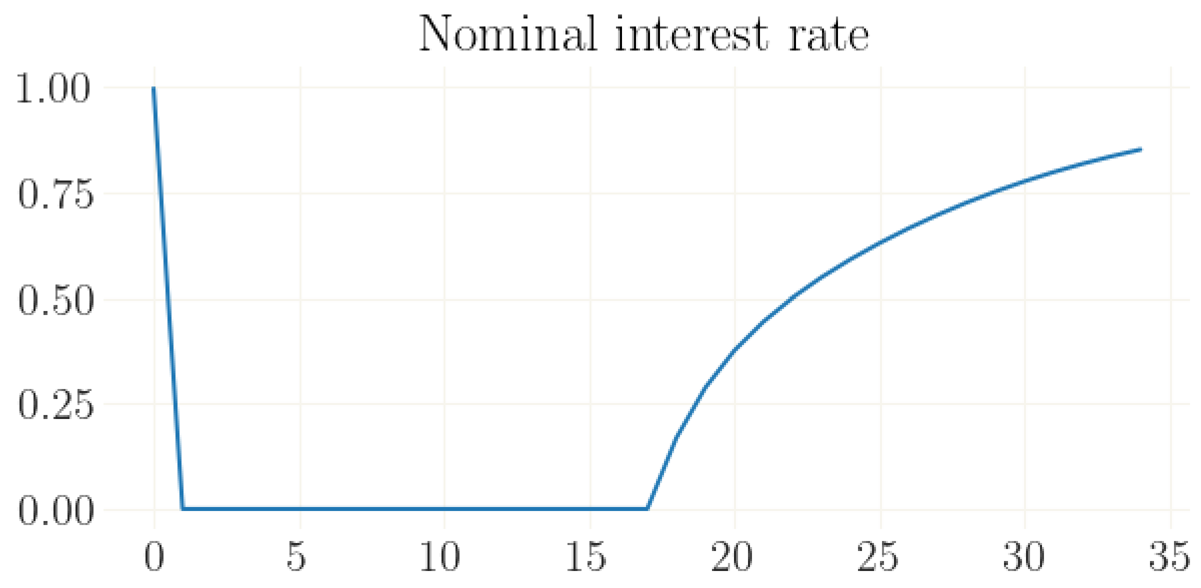
```
# and emulate again a 4% shock
x0['beta'] *= 1.04

# solving...
x, flag = mod.find_path(init_state=x0.values())

# plotting...
plt.figure(figsize=(7,3))
plt.plot(100*jnp.log(x[:35,mod['variables'].index('r')]))
plt.title('Nominal interest rate')
```

```
    Iteration   1 | max error 1.51e-01 | lapsed 0.0034s
    Iteration   2 | max error 2.92e-01 | lapsed 0.0066s
    Iteration   3 | max error 1.57e-01 | lapsed 0.0095s
    Iteration   4 | max error 3.66e-02 | lapsed 0.0124s
    Iteration   5 | max error 3.93e-03 | lapsed 0.0151s
    Iteration   6 | max error 3.90e-05 | lapsed 0.0177s
    Iteration   7 | max error 5.68e-10 | lapsed 0.0203s
(find_path:) Stacking done (0.092s).
```

```
[5]: Text(0.5, 1.0, 'Nominal interest rate')
```



CHAPTER

NINE

RANK TUTORIAL

Let us dive a bit deeper into the functionalities of the package by looking at a nonlinear medium scale DSGE model in the style of Smets & Wouters (2003,2007). The model features Rothemberg instead of Calvo pricing, the zero-lower bound on the nominal interest rate, and downwards nominal wage rigidity. The full model specification can be found in the appendix of the original paper, whereas the yaml file can be found here.

Start with some misc imports and load the package. The rest of these tutorials rely on the `grgrlib` for plotting, which can be installed via the official repositories ("`pip install grgrlib`").

[1]:
```
import jax.numpy as jnp # use jax.numpy instead of normal numpy
from grgrlib import grplot # nice backend for batch plotting with matplotlib
import econpizza as ep # pizza
import matplotlib.pyplot as plt

# only necessary if you run this in a jupyter notebook:
%matplotlib inline
```

The YAML file called `dsge.yml` is, together with a few other examples, provided with the package and can be found in this folder.

These examples can be imported from the `econpizza.examples` submodule.

[2]:
```
# the path to the example YAML
example_dsge = ep.examples.dsge
```

This is nothing else than the local path to the YAML file:

[3]:
```
print(example_dsge)
```
```
/home/gboehl/github/econpizza/econpizza/examples/dsge.yml
```

Let us make use of the functionality to parse the model before loading it, so that we can make some manual adjustments. This is especially useful if we want to loop over different parameter values.

[4]:
```
model_dict = ep.parse(example_dsge)
model_dict.keys()
```

[4]:
```
dict_keys(['name', 'description', 'variables', 'parameters', 'shocks', 'equations',
↪'steady_state', 'path', 'vars'])
```

`model_dict` now contains all information on the model. Let's, for example, change the sensitivity of the monetary policy rule w.r.t. inflation, and then load the model. Note that I'm loading `model_dict` instead of the path to the YAML. `ep.load` would accept both as input, but of course only `model_dict` contains the changed parameter value.

```
[5]: model_dict['steady_state']['fixed_values']['phi_pi'] = 2.
     # load the model
     mod = ep.load(model_dict)
     type(mod)
```

```
(load:) Parsing done.
```

```
[5]: econpizza.__init__.PizzaModel
```

mod is now an instance of the class PizzaModel, which is the generic model class. Note that this still contains the original dictionary together with some compiled information:

```
[6]: mod.keys()
```

```
[6]: dict_keys(['name', 'description', 'variables', 'parameters', 'shocks', 'equations',
     ↪'steady_state', 'path', 'vars', 'context', 'cache', 'func_strings'])
```

Lets find the steady state.

```
[7]: _ = mod.solve_stst()
```

```
    Iteration   1 | max. error 1.90e+00 | lapsed 1.5861
    Iteration   2 | max. error 2.31e+00 | lapsed 1.6636
    Iteration   3 | max. error 9.71e-01 | lapsed 1.6644
    Iteration   4 | max. error 1.13e-01 | lapsed 1.6650
    Iteration   5 | max. error 4.88e-02 | lapsed 1.6656
    Iteration   6 | max. error 1.79e-02 | lapsed 1.6662
    Iteration   7 | max. error 4.54e-03 | lapsed 1.6668
    Iteration   8 | max. error 4.67e-04 | lapsed 1.6674
    Iteration   9 | max. error 6.07e-06 | lapsed 1.6680
(solve_stst:) Steady state found (1.9064s). The solution converged.
```

Note that the result gets cached and will not be re-evaluated if called again (this can be bypassed by using the force=True flag in solve_stst()).

```
[8]: _ = mod.solve_stst()
```

```
(solve_stst:) Steady state already known.
```

…but you could change a parameter or steady state value and reevaluate again:

```
[9]: model_dict['steady_state']['fixed_values']['sigma_c'] = 1.5
     # load the model
     mod = ep.load(model_dict)
     newton_dict = mod.solve_stst()
```

```
(load:) Parsing done.
    Iteration   1 | max. error 1.90e+00 | lapsed 1.4342
    Iteration   2 | max. error 2.31e+00 | lapsed 1.4352
    Iteration   3 | max. error 9.71e-01 | lapsed 1.4358
    Iteration   4 | max. error 1.13e-01 | lapsed 1.4365
    Iteration   5 | max. error 4.88e-02 | lapsed 1.4371
    Iteration   6 | max. error 1.79e-02 | lapsed 1.4377
    Iteration   7 | max. error 4.54e-03 | lapsed 1.4383
    Iteration   8 | max. error 4.67e-04 | lapsed 1.4389
    Iteration   9 | max. error 6.07e-06 | lapsed 1.4395
(solve_stst:) Steady state found (1.5254s). The solution converged.
```

Also note that this was much faster thant the first run above, because the function `solve_stst()` is now cached. This makes it much faster to try out different steady state values.

The object `newton_dict` contains the results from the Newton-based root finding, which may be interesting for debugging (you must use the `raise_errors=False` flag to avoid raising an error and to get the dictionary):

[10]:
```
model_dict_broken = ep.copy(model_dict) # ep.copy is an alias for deepcopy
model_dict_broken['steady_state']['fixed_values']['mc'] = 200. # so wrong!
# load the model
mod = ep.load(model_dict_broken)
newton_dict = mod.solve_stst(raise_errors=False)
```

```
(load:) Loading cached model.
    Iteration   1 | max. error 1.20e+03 | lapsed 0.0004
    Iteration   2 | max. error 1.20e+03 | lapsed 0.0011
    Iteration   3 | max. error 1.20e+03 | lapsed 0.0016
    Iteration   4 | max. error 1.20e+03 | lapsed 0.0022
    Iteration   5 | max. error 1.20e+03 | lapsed 0.0027
    Iteration   6 | max. error 1.26e+03 | lapsed 0.0033
    Iteration   7 | max. error 1.89e+04 | lapsed 0.0038
    Iteration   8 | max. error 2.38e+04 | lapsed 0.0044
    Iteration   9 | max. error 1.20e+03 | lapsed 0.0049
    Iteration  10 | max. error 1.20e+03 | lapsed 0.0055
    Iteration  11 | max. error 1.20e+03 | lapsed 0.0060
    Iteration  12 | max. error 1.20e+03 | lapsed 0.0066
    Iteration  13 | max. error 1.20e+03 | lapsed 0.0071
    Iteration  14 | max. error 1.20e+03 | lapsed 0.0076
(solve_stst:) Steady state FAILED (max. error is 1.20e+03 in eqn. 13). Maximum number of␣
↪15 iterations reached.
```

This failed because marginal costs are a function of the values of `theta`. While in this case the reason is clear, in other cases you could have a look at the Newton dictionary to debug this:

[11]:
```
print(newton_dict.keys())
```

```
dict_keys(['success', 'message', 'x', 'niter', 'fun', 'jac', 'det', 'initial_values'])
```

Let us better return to the working model. The model has many shocks. We'll go for a risk premium shock, `e_u`.

[12]:
```
mod = ep.load(model_dict)
_ = mod.solve_stst()
print(mod['shocks'])

# shock the risk premium
shk = ('e_u', .01)
```

```
(load:) Loading cached model.
(solve_stst:) Steady state already known.
['e_beta', 'e_z', 'e_g', 'e_p', 'e_w', 'e_i', 'e_r', 'e_u']
```

Simulation works as before…

[13]:
```
# find the nonlinear trajectory
x, flag = mod.find_path(shock=shk)
```

```
    Iteration   1 | max error 2.24e+00 | lapsed 2.3029s
    Iteration   2 | max error 7.13e-02 | lapsed 2.3286s
```

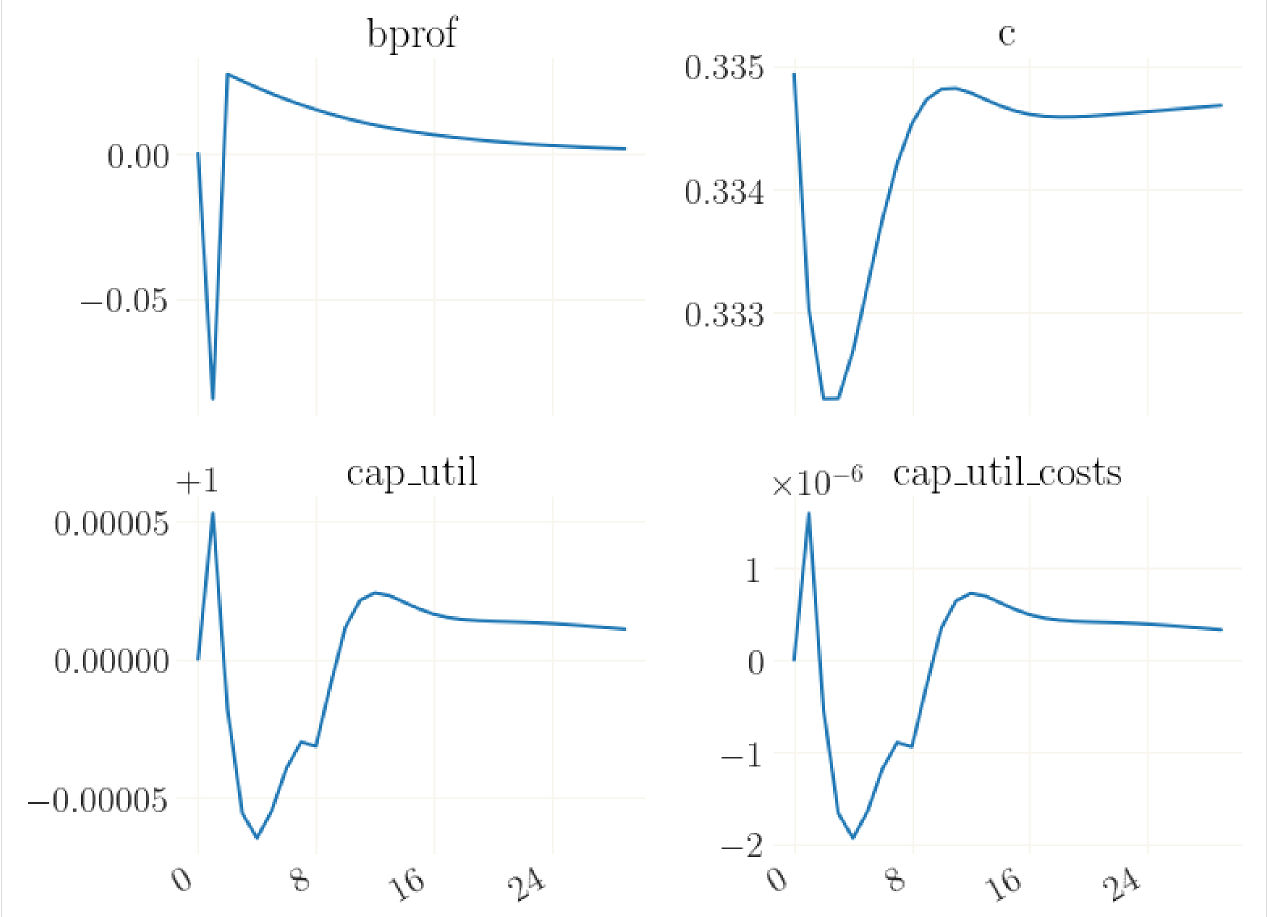
(continues on next page)

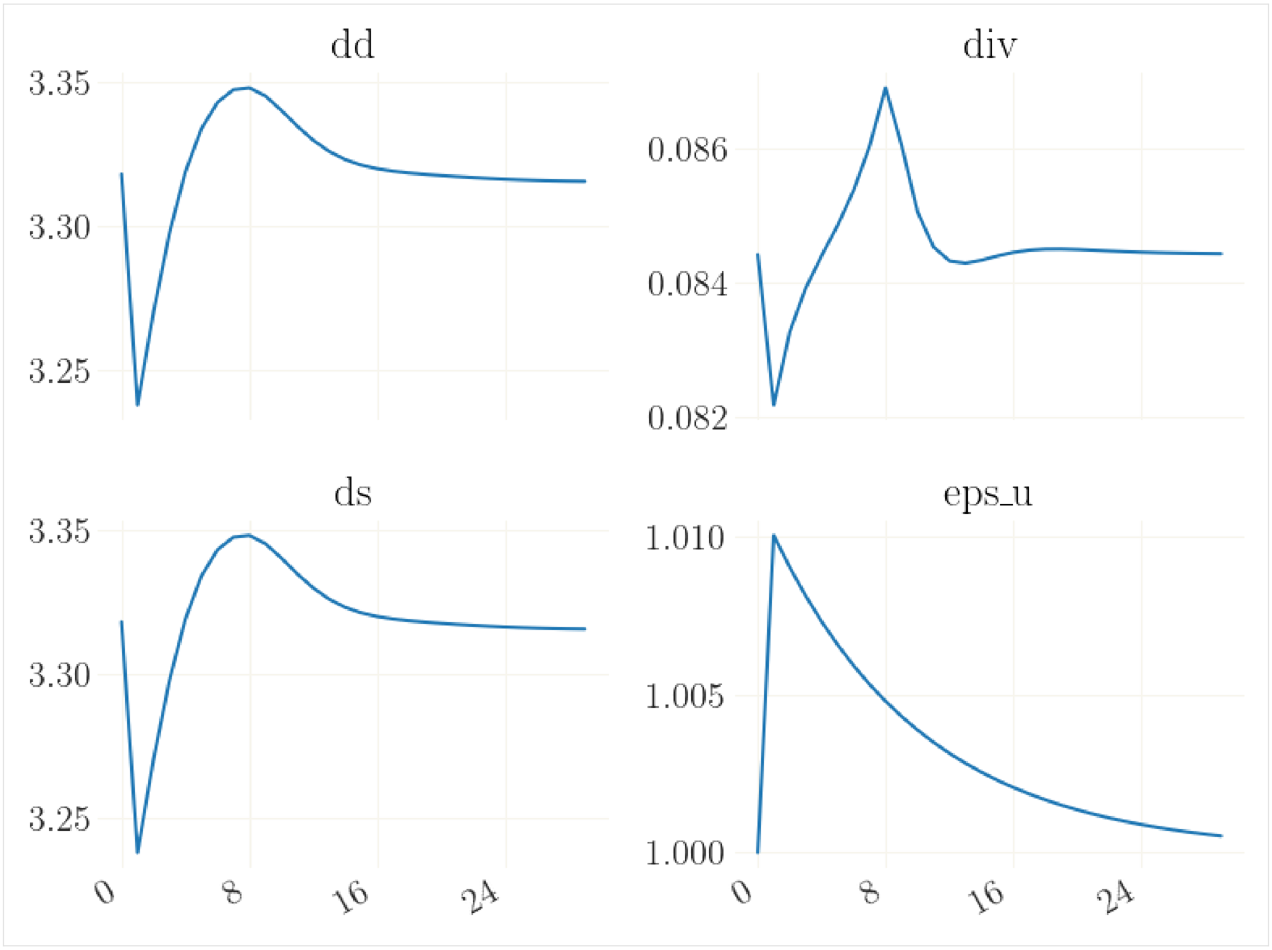
(continued from previous page)

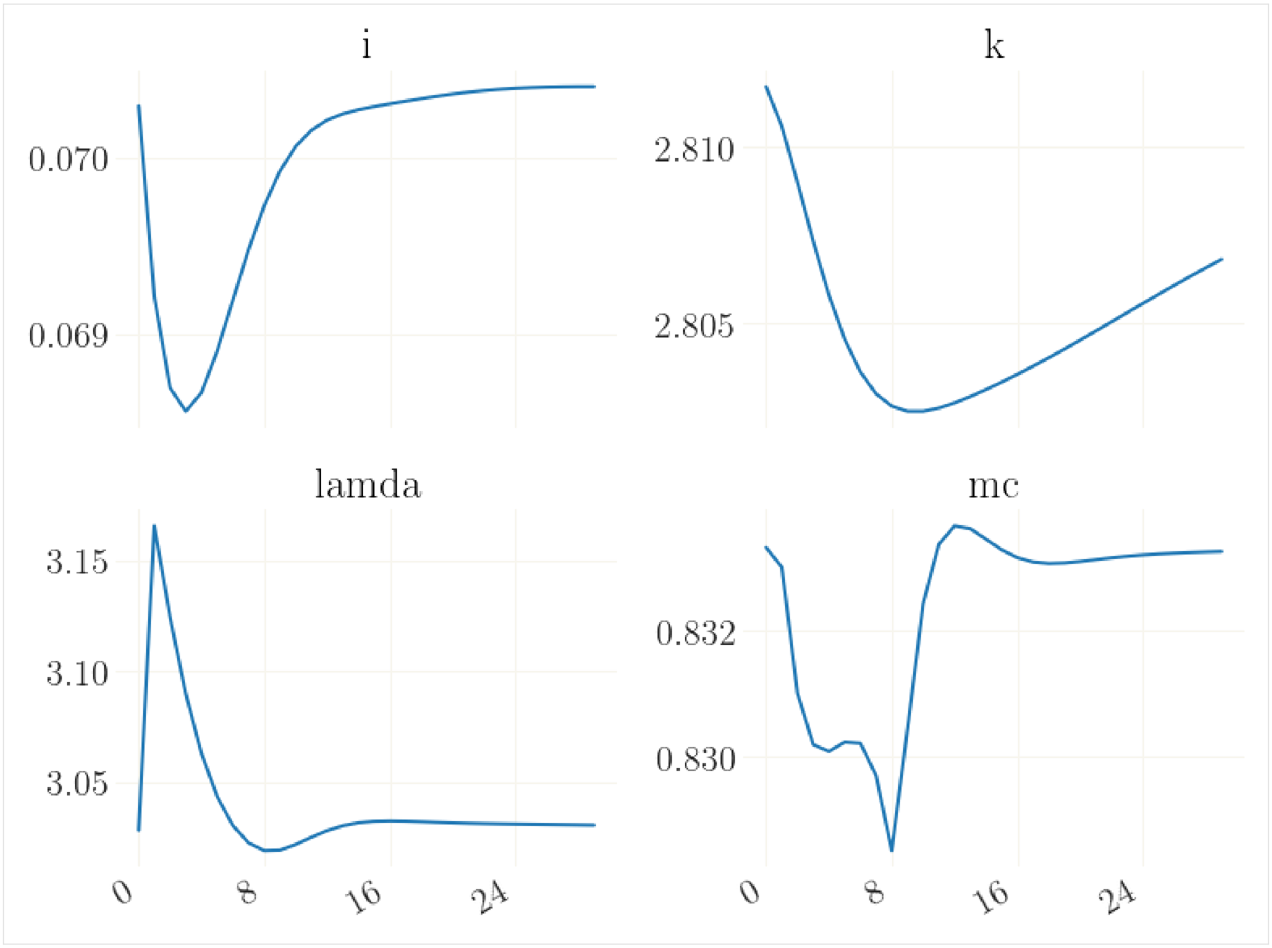
```
    Iteration   3 | max error 3.33e-02 | lapsed 2.3434s
    Iteration   4 | max error 3.10e-02 | lapsed 2.3574s
    Iteration   5 | max error 2.39e-02 | lapsed 2.3715s
    Iteration   6 | max error 1.37e-02 | lapsed 2.3856s
    Iteration   7 | max error 2.72e-03 | lapsed 2.3997s
    Iteration   8 | max error 1.61e-05 | lapsed 2.4138s
    Iteration   9 | max error 6.39e-11 | lapsed 2.4279s
(find_path:) Stacking done (2.596s).
```

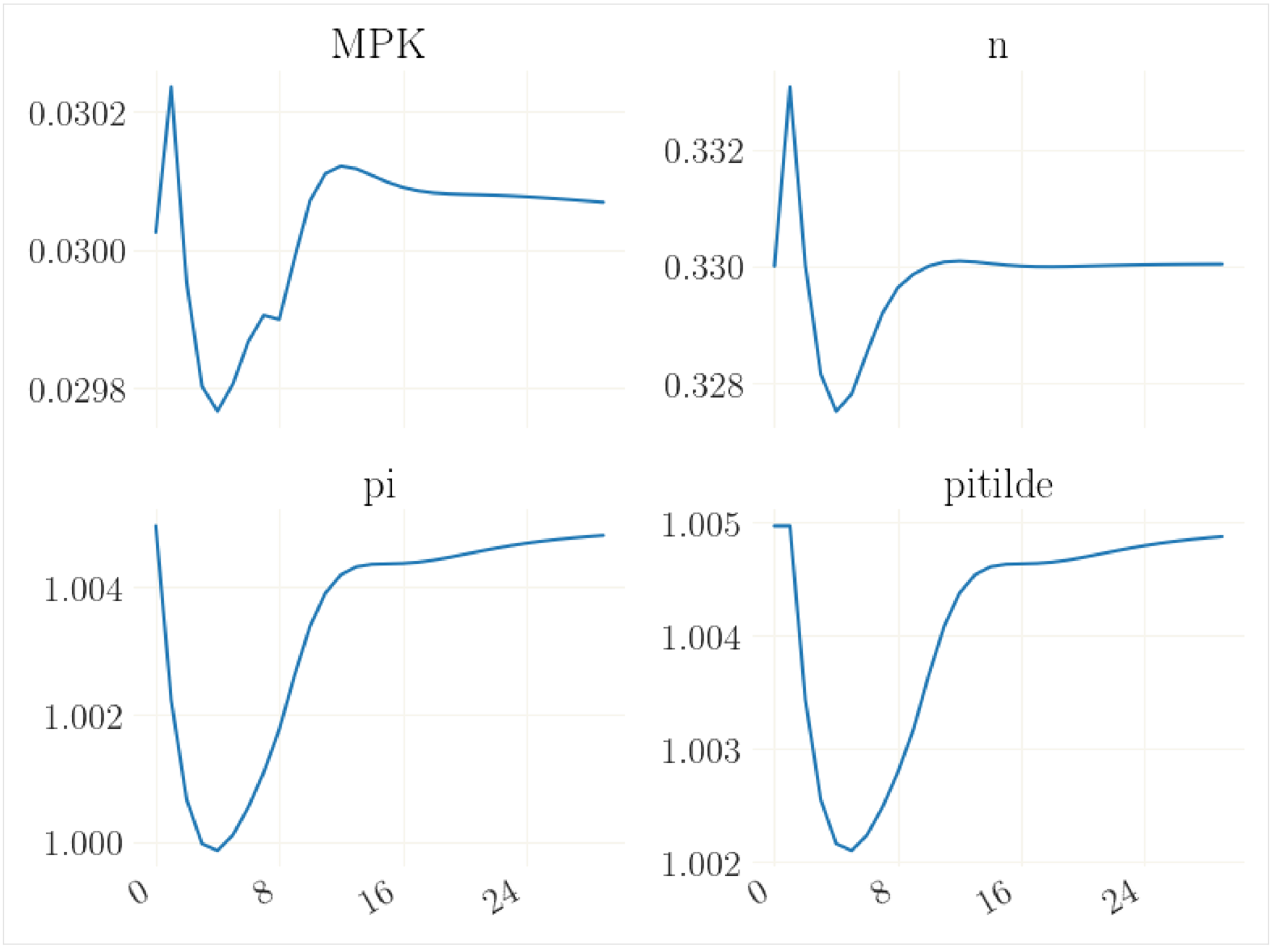
Now x contains the trajectory in response to the shock. Let us plot this. Note that the dynamics are somewhat "twisted" because of the downwards nominal wage rigidigy.

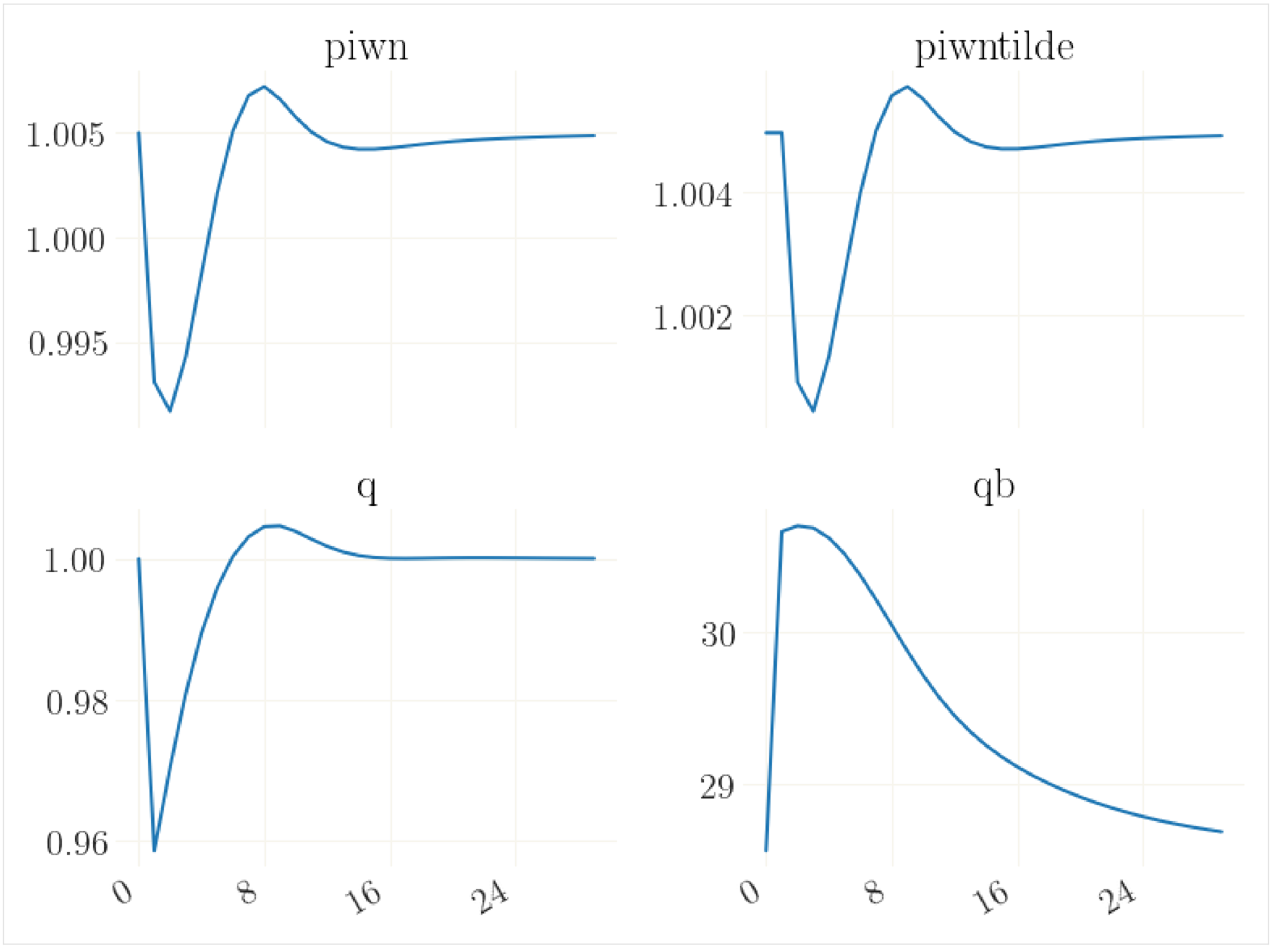
```
[14]: _ = grplot(x[:30], labels=mod['variables'])
```

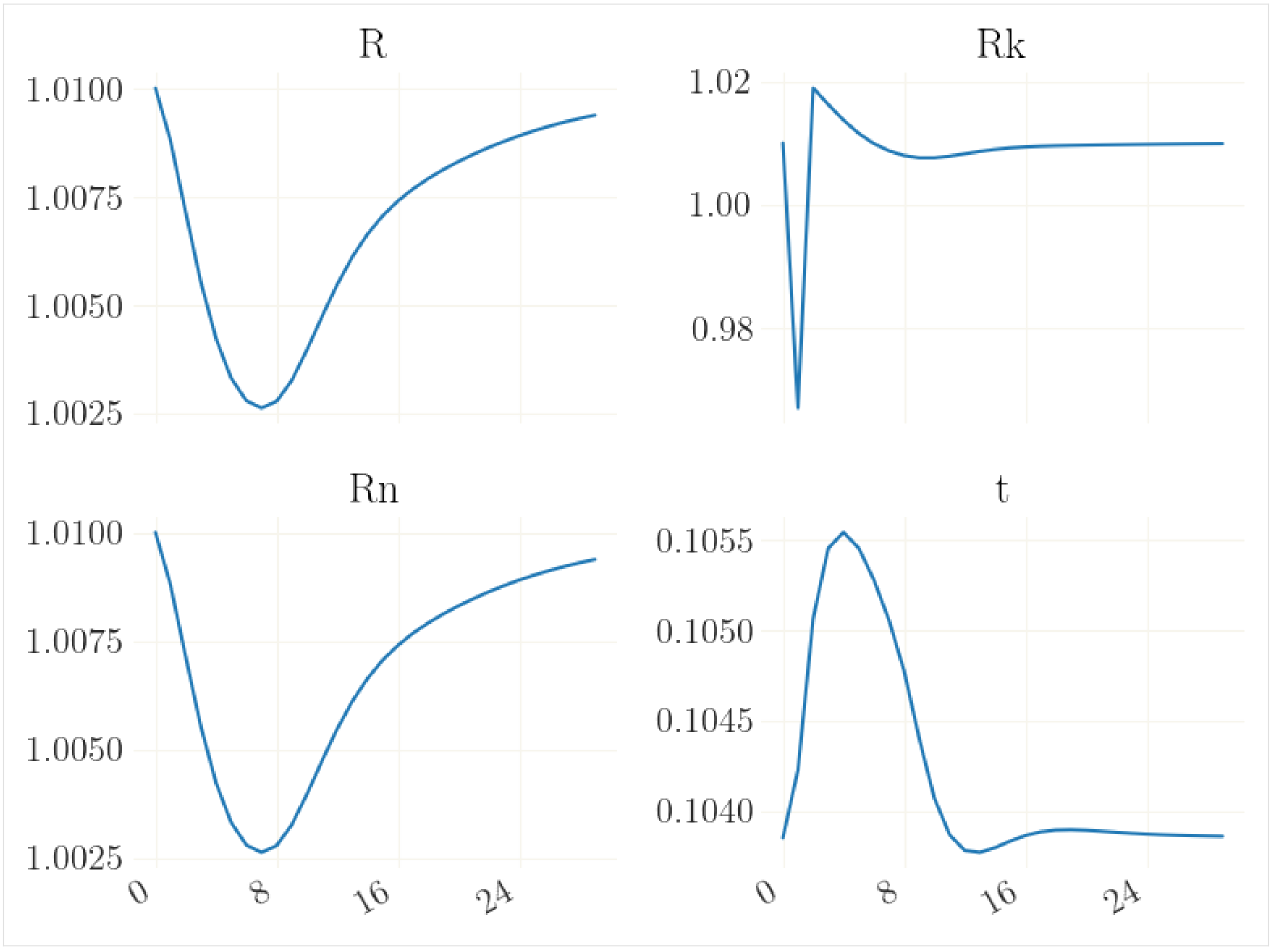


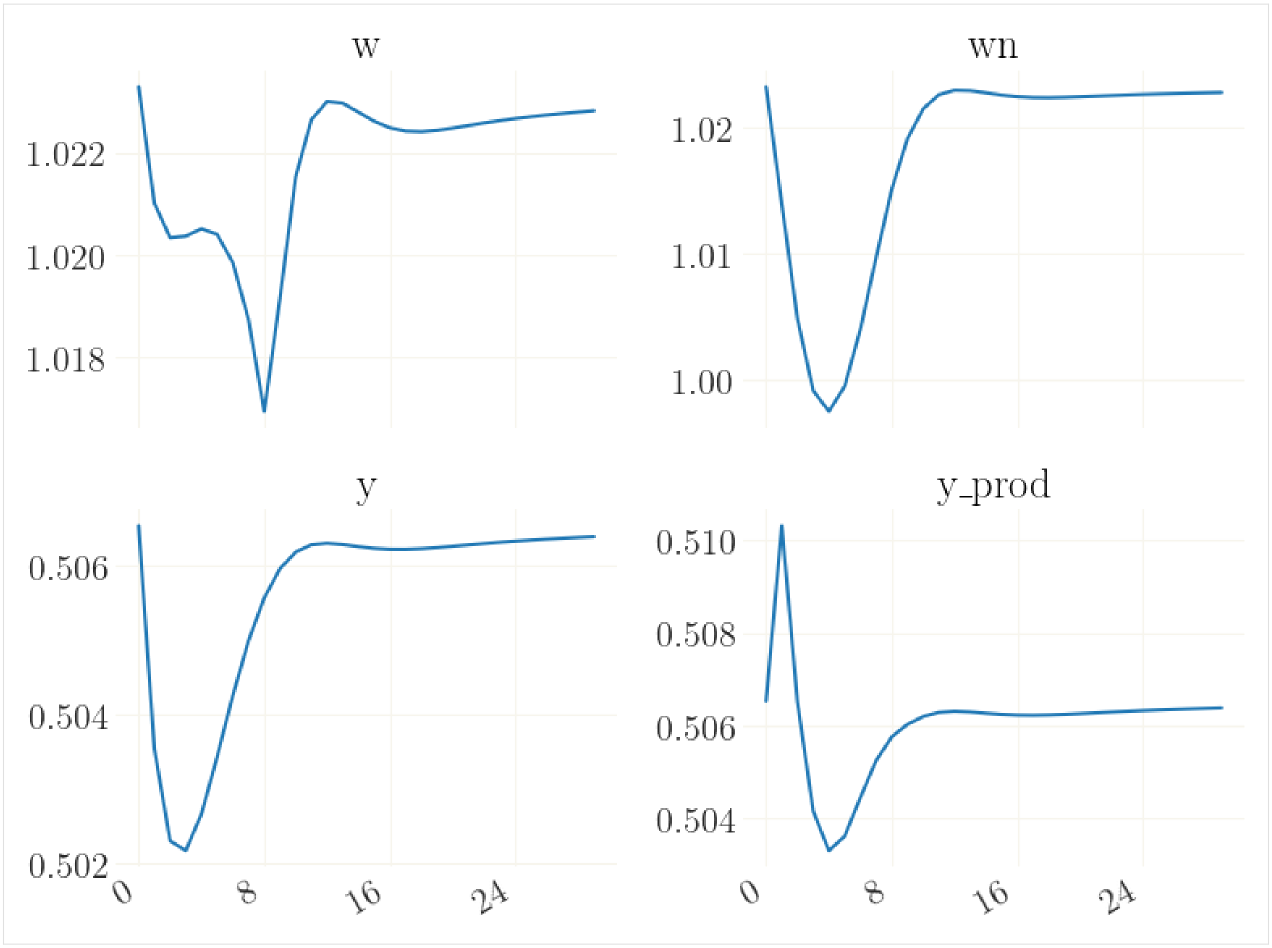

dd
div
ds
eps_u
3.35
3.30
3.25
0.086
0.084
0.082
1.010
1.005
1.000
0
8
16
24


i
0.070
0.069
k
2.810
2.805
lamda
3.15
3.10
3.05
mc
0.832
0.830
0
8
16
24
0
8
16
24


MPK
n
pi
pitilde
0.0302
0.0300
0.0298
0.332
0.330
0.328
1.004
1.002
1.000
1.005
1.004
1.003
1.002
0
8
16
24
0
8
16
24


piwn
1.005
1.000
0.995
piwntilde
1.004
1.002
q
1.00
0.98
0.96
qb
30
29
0
8
16
24
0
8
16
24


R
1.0100
1.0075
1.0050
1.0025
Rk
1.02
1.00
0.98
Rn
1.0100
1.0075
1.0050
1.0025
t
0.1055
0.1050
0.1045
0.1040
0
8
16
24
0
8
16
24


w
wn
y
y_prod
1.022
1.020
1.018
1.02
1.01
1.00
0.506
0.504
0.502
0.510
0.508
0.506
0.504
0
8
16
24
0
8
16
24

CHAPTER

TEN

ONE-ASSET HANK TUTORIAL

The package supports heterogeneous agent models with and without porfolio chocie (i.e., with one asset and two assets, respectively). Start again with misc imports and load the package:

[1]:
```
import jax.numpy as jnp # use jax.numpy instead of normal numpy
from grgrlib import figurator, grplot # a nice backend for batch plotting with matplotlib
import econpizza as ep # pizza
import matplotlib.pyplot as plt

# only necessary if you run this in a jupyter notebook:
%matplotlib inline
```

We now look at the one-asset HANK, which is documented in the appendix of in the paper. The YAML file, with many comments, can be found in the examples folder.

Start with loading the example file:

[2]:
```
example_hank = ep.examples.hank
```

As before, `example_hank` is nothing else than the path to the YAML file:

[3]:
```
print(example_hank)
```

```
/home/gboehl/github/econpizza/econpizza/examples/hank_with_comments.yml
```

Parse the example hank model from the yaml and compile the model:

[4]:
```
# parse model
hank1_dict = ep.parse(example_hank)
# compile the model
hank1 = ep.load(hank1_dict)
```

```
(load:) Parsing done.
```

The first step creates a raw dictionary from the yaml. The second translates everything to a model instance with compiled and tested functions. If something specific in your model does not work, you should have been informed by now.

Lets continue with the steady state:

[5]:
```
stst_result = hank1.solve_stst()
```

```
    Iteration   1 | max. error 7.48e-01 | lapsed 5.2069
    Iteration   2 | max. error 7.56e-02 | lapsed 5.3326
    Iteration   3 | max. error 7.36e-04 | lapsed 5.3654
```

(continues on next page)

(continued from previous page)

```
    Iteration   4 | max. error 7.06e-08 | lapsed 5.3978
(solve_stst:) Steady state found (5.6612s). The solution converged.
```

By default, the final message is rather verbose. The rank of the Jacobian is important because quite often, the steady state is indetermined and some steady state values need to be fixed in advance. Econpizza can deal with that by using the Pseudoinverse during the Newton steps. Fixing some of the variables is also what I did here. Since the function has 12 degrees of freedom and 6 fixed variables for a total of 18 variables, we're fine and the steady state solver nicely converges.

The resulting `stst_result` is similar to the return object from `scipy.optimize.root` and contains all sorts of nice information to help you debugging if you have problems finding the steady state:

[6]:
```
print(stst_result.keys())
```

```
dict_keys(['success', 'message', 'x', 'niter', 'fun', 'jac', 'aux', 'det', 'initial_
↪values'])
```

[7]:
```
print(stst_result['fun']) # the steady state function at the solution x
```

```
[-2.38671438e-09  1.77635684e-15  2.94209102e-15  0.00000000e+00
 -2.77555756e-17  0.00000000e+00 -2.22044605e-16  2.38671397e-09
  0.00000000e+00  4.65979351e-17  0.00000000e+00  0.00000000e+00
  8.35687075e-12  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00]
```

The pizza automatically stores the steady state values as a dictionary in the model object:

[8]:
```
hank1['stst']
```

[8]:
```
{'B': Array(5.6, dtype=float64),
 'beta': Array(0.98, dtype=float64),
 'C': Array(1., dtype=float64),
 'div': Array(0.23927423, dtype=float64),
 'n': Array(0.91287093, dtype=float64),
 'pi': Array(1., dtype=float64),
 'R': Array(1.00351564, dtype=float64),
 'Rn': Array(1.00351564, dtype=float64),
 'Rr': Array(1.00351564, dtype=float64),
 'Rstar': Array(1.00351564, dtype=float64),
 'tax': Array(0.01968759, dtype=float64),
 'Top10A': Array(0.39757979, dtype=float64),
 'Top10C': Array(0.20057934, dtype=float64),
 'w': Array(0.83333333, dtype=float64),
 'y': Array(1., dtype=float64),
 'y_prod': Array(1., dtype=float64),
 'z': Array(1.09544512, dtype=float64)}
```

Let us, out of curiousity, have a look at the steady state distribution. It is stored under `hank1['steady_state']`. Note that at the same location, also steady state `decisions` (the value function) are stored.

[9]:
```
dist = hank1['steady_state']['distributions'][0]
grid = hank1['context']['a_grid']
```

`hank1[context]` is a dictionary that also stores some other model specific variables. Better have a look youself if you care. For those who really miss Dynare and the global access to model variables and objects, you can simply add the model context to globals:

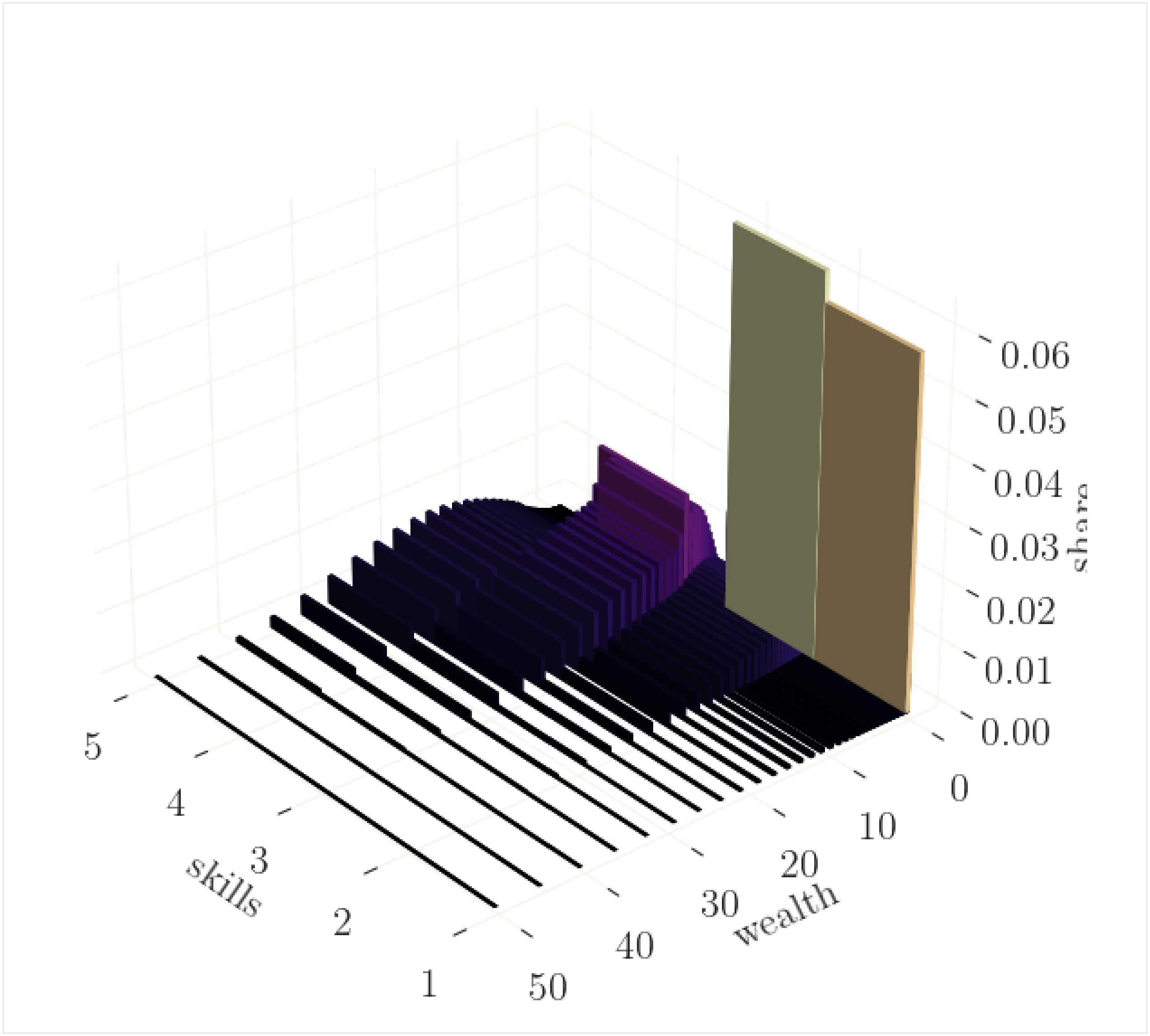
```
[10]: globals().update(hank1['context'])
      print(a_grid) # this is now available while it was not before
```

```
[0.00000000e+00 2.85761999e-02 6.04187966e-02 9.59011550e-02
 1.35439317e-01 1.79496881e-01 2.28590436e-01 2.83295620e-01
 3.44253869e-01 4.12179939e-01 4.87870284e-01 5.72212399e-01
 6.66195222e-01 7.70920734e-01 8.87616874e-01 1.01765194e+00
 1.16255064e+00 1.32401196e+00 1.50392908e+00 1.70441160e+00
 1.92781022e+00 2.17674438e+00 2.45413291e+00 2.76322829e+00
 3.10765474e+00 3.49145079e+00 3.91911658e+00 4.39566661e+00
 4.92668860e+00 5.51840896e+00 6.17776579e+00 6.91249027e+00
 7.73119729e+00 8.64348644e+00 9.66005463e+00 1.07928214e+01
 1.20550689e+01 1.34615974e+01 1.50288988e+01 1.67753502e+01
 1.87214295e+01 2.08899549e+01 2.33063532e+01 2.59989575e+01
 2.89993393e+01 3.23426792e+01 3.60681788e+01 4.02195210e+01
 4.48453815e+01 5.00000000e+01]
```

Let’s plot the distribution:

```
[11]: from grgrlib import grbar3d # a nice backend to 3D-plots with matplotlib

      ax, _ = grbar3d(dist, xedges=jnp.arange(1,5), yedges=grid, figsize=(9,7), depth=.5) #␣
      ↪create 3D plot
      # set axis labels
      ax.set_xlabel('skills')
      ax.set_ylabel('wealth')
      ax.set_zlabel('share')
      # rotate
      ax.view_init(azim=140)
```



Nice. As expected, agents with higher income hold more assets, and vice versa. Note however that quantities are here given as shares of nodes on a log-grid (rather than true densities), meaning that shares for larger values on the grid are overrepresented.

Let’s continue with calculating some impulse response functions. We’ll have a look at a shock to the households’ discount factor β.

Find the *nonlinear* IRFs (we will treat linear IRFs in the next tutorial):

[12]:
```
# define the shock as (shock_name, value)
shock = ('e_beta', 0.005)
# simulate
xst, flags = hank1.find_path(shock)
```

```
(get_derivatives:) Derivatives calculation done (6.270s).
(get_jacobian:) Jacobian accumulation and decomposition done (1.211s).
    Iteration  1 | fev.   1 | max. error 7.02e-02 | dampening 1.000
    Iteration  2 | fev.  15 | max. error 1.68e-02 | dampening 1.000
```

(continues on next page)

(continued from previous page)

```
    Iteration  3 | fev.  26 | max. error 1.11e-03 | dampening 1.000
    Iteration  4 | fev.  29 | max. error 9.26e-06 | dampening 1.000
    Iteration  5 | fev.  34 | max. error 5.85e-07 | dampening 1.000
    Iteration  6 | fev.  40 | max. error 3.50e-08 | dampening 1.000
    Iteration  7 | fev.  46 | max. error 1.77e-09 | dampening 1.000 | lapsed 11.0975s
(find_path:) Stacking done (18.775s). The solution converged.
```

Alternatively, take the steady state as the initial value, and alter the initial value of β directly:

[13]:
```
# this is a dict containing the steady state values
x0 = hank1['stst'].copy()
# setting a large shock on the discount factor
x0['beta'] *= 1.009

# simulate again with the different initial state:
xst, flags = hank1.find_path(init_state=x0.values())
```

```
    Iteration  1 | fev.   1 | max. error 1.13e-01 | dampening 1.000
    Iteration  2 | fev.  32 | max. error 4.76e-02 | dampening 1.000
    Iteration  3 | fev.  63 | max. error 1.09e-01 | dampening 0.619
    Iteration  4 | fev.  94 | max. error 3.84e-03 | dampening 1.000
    Iteration  5 | fev. 108 | max. error 2.20e-04 | dampening 1.000
    Iteration  6 | fev. 132 | max. error 9.08e-06 | dampening 1.000
    Iteration  7 | fev. 157 | max. error 8.16e-07 | dampening 1.000
    Iteration  8 | fev. 182 | max. error 7.31e-08 | dampening 1.000
    Iteration  9 | fev. 207 | max. error 6.54e-09 | dampening 1.000 | lapsed 6.0853s
(find_path:) Stacking done (6.172s). The solution converged.
```

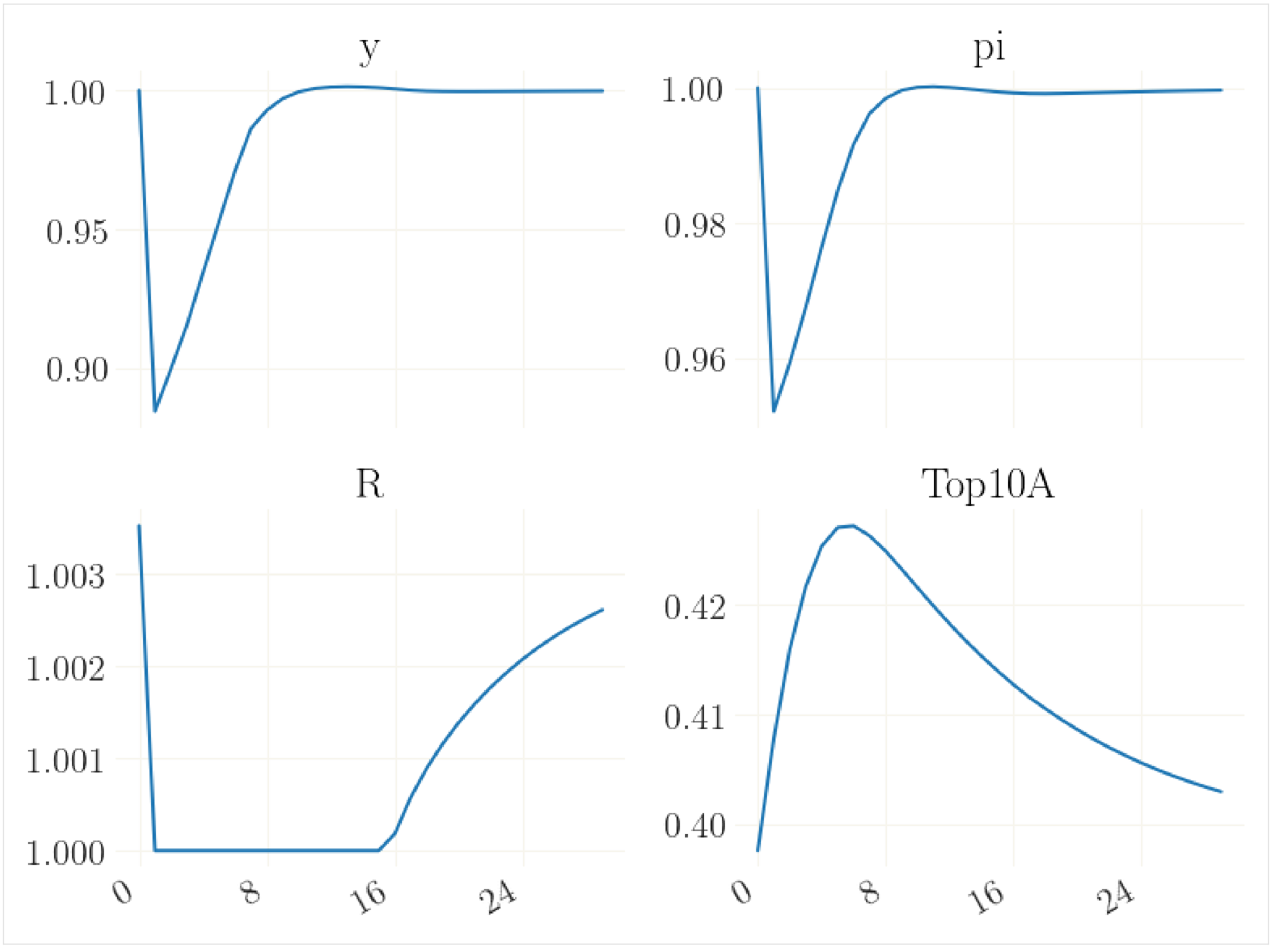
That went smoothly. Again, you will get (hopefully) meaningful and (hopefully) infomative final messages. Note that the second run was **much** faster than the first one. This is because the steady state sequence space Jacobian was already calculated and all functions were already compiled.

Let's plot only a few of the variables for space restrictions: output y_t (Y), inflation π_t (pi), the nominal interst rate R_t (Rn), and the percentage share of wealth held by the top-10% richest, Top10A.

[14]:
```
# this is how *all* aggregate variables could be plotted:
#grplot(xst[:30], labels=hank1['variables'])

variables = 'y', 'pi', 'R', 'Top10A'
inds = [hank1['variables'].index(v) for v in variables] # get indices of variables

_ = grplot(xst[:30, inds], labels=variables)
```



See how the effective lower bound is binding for quite a while, and how the endogenous distribution adjusts accordingly.

In case you want to study the distributional dynamics in detail, you can also back out the exact *nonlinear sequences* of the disaggregated variables and their distribution. That is, their complete history given the trajectory `xst` of aggregated variables.

```
[15]: # note that the sequence of aggregated variables is the input
het_vars = hank1.get_distributions(xst)
```

The function will return a dictionary with the disaggregated variables (`outputs` of the decision stage) and the distribution as key:

```
[16]: print(het_vars.keys())
```

```
dict_keys(['a', 'c', 'dist'])
```

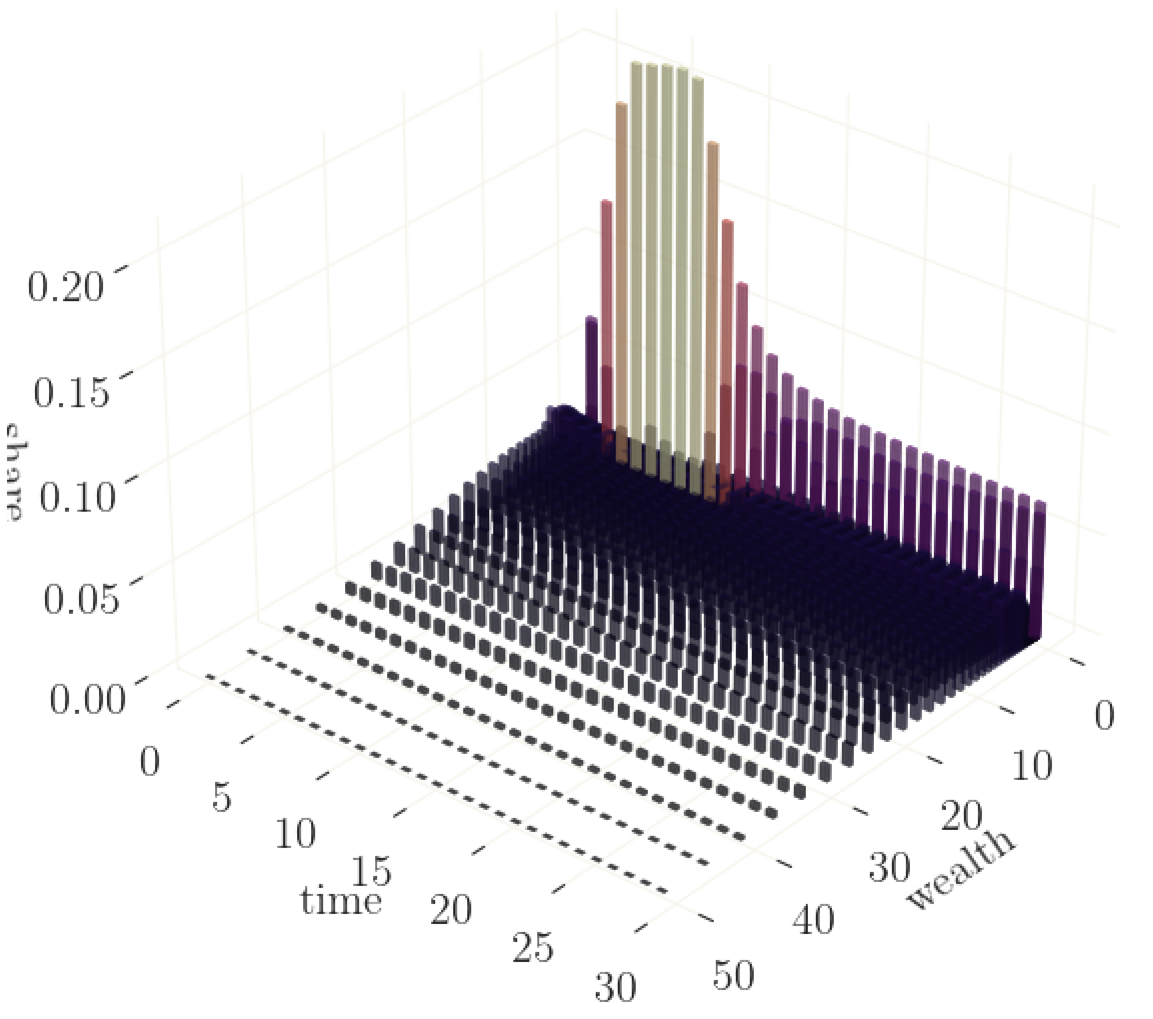
Each of the objects has shape `*distribution shape, number_of_periods`:

```
[17]: print(het_vars['a'].shape)
```

```
(4, 50, 199)
```

For example, we can use this to plot the distribution of wealth over time:

```
[18]: dist = het_vars['dist']
      a_grid = hank1['context']['a_grid']
      # plot
      ax, _ = grbar3d(dist[...,:30].sum(0), xedges=a_grid, yedges=jnp.arange(30), figsize=(9,
      ↪7), depth=.5, width=.5, alpha=.5)
      # set axis labels
      ax.set_xlabel('wealth')
      ax.set_ylabel('time')
      ax.set_zlabel('share')
      # rotate
      ax.view_init(azim=40)
```



The graph shows that the discount factor shock (and the ZLB) mainly affects the wealth of the housholds which hold no or very few assets, giving incentive to hold more assets for a short period of time. Again, since these are shares of nodes on a log-grid (rather than true densities), the shares for larger values on the grid are overrepresented.

CHAPTER

ELEVEN

TWO-ASSET-HANK TUTORIAL

Again start with misc imports and load the package:

[1]:
```
import jax.numpy as jnp
from grgrlib import figurator, grplot
import econpizza as ep # pizza
import matplotlib.pyplot as plt

# only necessary if you run this in a jupyter notebook:
%matplotlib inline
```

The provided example is the medium-scale two-asset model from the paper, which is again documented in the appendix. The YAML file can also be found in the examples folder. This model features a portfolio choice problem for households and all the bells ans whistles of the medium scale DSGE model.

[2]:
```
example_hank2 = ep.examples.hank2
# parse model
hank2_dict = ep.parse(example_hank2)
# compile the model
hank2 = ep.load(hank2_dict)
```

```
(load:) Parsing done.
```

[3]:
```
stst_result = hank2.solve_stst()
```

```
    Iteration   1 | max. error 3.64e+01 | lapsed 16.1393
(solve_stst:) Steady state found (19.409s). The solution converged.
```

Again, look at a discount factor shock and calculate the pefect foresight solution:

[4]:
```
# this is a dict containing the steady state values
x0 = hank2['stst'].copy()
# setting a shock on the discount factor
x0['beta'] *= 1.01
```

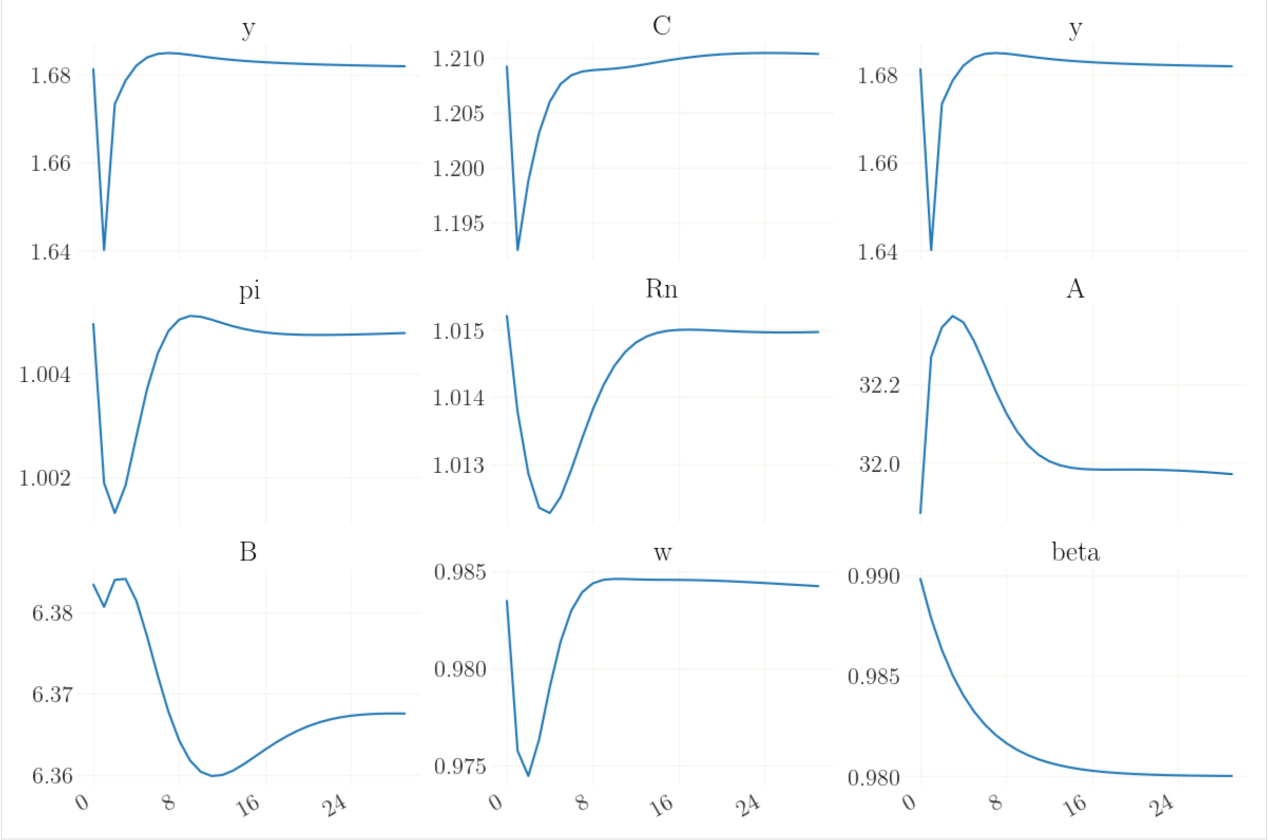
But let us this time start with a *linear* IRFs, just because we can:

[5]:
```
xlin, flags = hank2.find_path_linear(init_state=x0.values())
```

```
(get_derivatives:) Derivatives calculation done (13.100s).
(get_jacobian:) Jacobian accumulation and decomposition done (6.249s).
(find_path_linear:) Linear solution done (26.710s).
```

```
[6]: variables = 'y', 'C', 'y', 'pi', 'Rn', 'A', 'B', 'w', 'beta'
inds = [hank2['variables'].index(v) for v in variables]

figs, axs = figurator(3,3, figsize=(12,8))
_ = grplot(xlin[:30, inds], labels=variables, ax=axs)
```

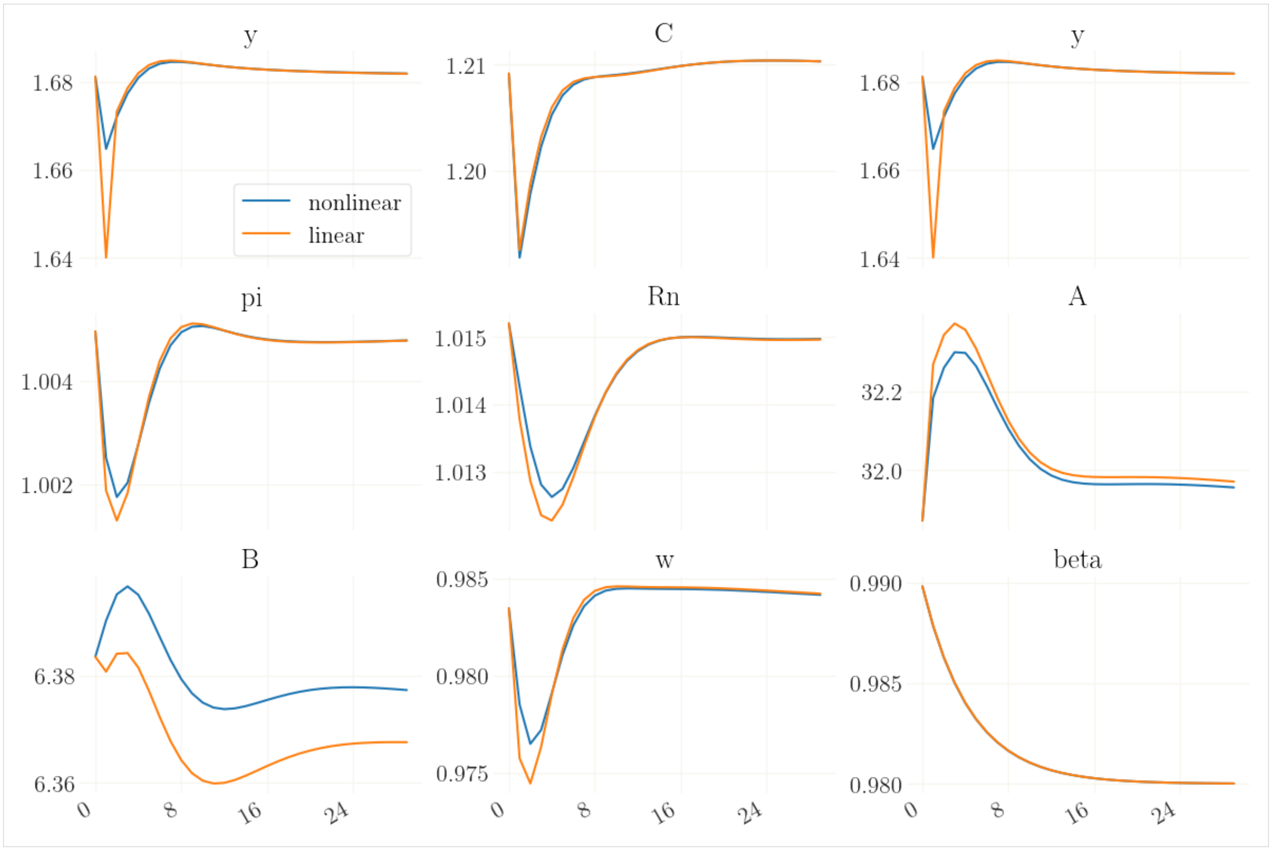


Great. Finally, we can compare this with the fully nonlinear responses:

```
[7]: xst, flags = hank2.find_path(init_state=x0.values())
```

```
    Iteration  1 | fev.   1 | max. error 4.35e-01 | dampening 1.000
    Iteration  2 | fev.   3 | max. error 5.81e-03 | dampening 1.000
    Iteration  3 | fev.   4 | max. error 9.66e-07 | dampening 1.000
    Iteration  4 | fev.   5 | max. error 3.31e-10 | dampening 1.000 | lapsed 25.7429s
(find_path:) Stacking done (25.885s). The solution converged.
```

```
[8]: figs, axs = figurator(3,3, figsize=(12,8))
_ = grplot((xst[:30, inds], xlin[:30, inds]), labels=variables, ax=axs, legend=(
↪'nonlinear', 'linear'))
_ = axs[0].legend(fontsize=16)
```


y
C
y
pi
Rn
A
B
w
beta
nonlinear
linear
1.68
1.66
1.64
1.21
1.20
1.004
1.002
1.015
1.014
1.013
32.2
32.0
6.38
6.36
0.985
0.980
0.975
0.990
0.985
0.980
0
8
16
24

INDEX