econpizza

Release 0.1.12

Gregor Boehl

Jul 05, 2022

CONTENTS

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

ECONPIZZA

Solve nonlinear heterogeneous agent models using machine learning techniques

Econpizza is a framework to solve and simulate nonlinear perfect foresight models, with or without heterogeneous agents. A parser allows to express economic models in a simple, high-level fashion as yaml-files. Additionally, generic and robust routines for steady state search are provided.

The baseline solver is a Newton-based stacking method in the spirit of Boucekkine (1995), Juillard (1996) and others. Hence, the method is similar to the solver in dynare, but faster and more robust due to the use of automatic differentiation and sparse jacobians. Even perfect-foresight IRFs for large-scale nonlinear models with, e.g., occassionally binding constraints can be computed in less than a second.

The package makes heavy use of automatic differentiation via JAX.

Econpizza can solve nonlinear HANK models. The approach to deal with the distribution is inspired by the Sequence-Space Jacobian method (Auclert et al., 2022, ECMA). Steady state and nonlinear impulse responses (including, e.g., the ELB) can typically be found within a few seconds.

There is a model parser to allow for the simple and generic specification of models (with or without heterogeneity).

1.1 Documentation

The documentation and a **tutorial** can be found here.

1.2 Installation

Installing the repository version from PyPi is as simple as:

```
pip install econpizza
```

Alternatively, the most recent version from GitHub with some experimental features can be installed via

```
pip install git+https://github.com/gboehl/grgrlib
pip install git+https://github.com/gboehl/econpizza
```

Note that the latter requires git to be installed.

1.2.1 Installation on Windows

Econpizza needs **JAX** to be installed. This is not a problem for MacOS and Linux, but the time for JAX to fully support Windows has not yet come. Fortunately, there is help out there (see here for the somewhat cryptic original reference). To install JAX, run

```
pip install "jax[cpu]===0.3.14" -f https://whls.blob.core.windows.net/unstable/index.
↪html --use-deprecated legacy-resolver
```

prior to installing Econpizza. Econpizza then runs just fine (proof).

1.3 References

econpizza is developed by Gregor Boehl to simulate nonlinear perfect foresight models. Please cite it with

```
@Misc{boehl2022pizza,
title         = {Econpizza: solving nonlinear heterogeneous agents models using machine␣
↪learning techniques},
author        = {Boehl, Gregor},
howpublished  = {\url{https://econpizza.readthedocs.io/_/downloads/en/latest/pdf/}},
year = {2022}
}
```

For the Boehl-Hommes method: Boehl and Hommes (2021). Rational vs. Irrational Beliefs in a Complex World. *IMFS Working papers*

```
@techreport{boehl2021rational,
title         = {Rational vs. Irrational Beliefs in a Complex World},
author        = {Boehl, Gregor and Hommes, Cars},
year          = 2021,
institution   = {IMFS Working Paper Series}
}
```

I appreciate citations for **econpizza** because it helps me to find out how people have been using the package and it motivates further work.

CHAPTER

TWO

GETTING STARTED

This package contains two methods. *Stacking*, the main method, is a generic nonlinear solver that should work on all sorts of problems. *Shooting* is the method of Boehl & Hommes (2021), which is useful for models with nonlinear, chaotic dynamics.

2.1 Quickstart

An small-scale nonlinear New Keynesian model with ZLB is provided as an example. Here is how to simulate it and plot some nonlinear impulse responses:

```python
import matplotlib.pyplot as plt
import econpizza as ep
from econpizza import example_nk


# use the NK model again
mod = ep.load(example_nk)
_ = mod.solve_stst()
_ = mod.solve_linear()

# increase the discount factor by .02 (this is _not_ percentage deviation!)
shk = ('e_beta', .02)

# use the stacking method. As above, you could also feed in the initial state instead
x, x_lin, flag = mod.find_stack(shock=shk)

# plotting. x_lin is the linearized first-order solution
for i,v in enumerate(mod['variables']):

    plt.figure()
    plt.plot(x[:,i])
    plt.plot(x_lin[:,i])
    plt.title(v)
```

The impulse responses are the usual dynamics of a nonlinear DSGE.

The folder yaml files also contains a medium scale New Keynesian DSGE model as an example file (`med_scale_nk.yaml`, see here). It can be imported with:

```
from econpizza import example_dsge

mod = ep.load(example_dsge)
```

2.2 The *.yaml-file

All relevant information is supplied via the yaml file. For general information about the YAML markup language and its syntax see Wikipedia. The yaml files follow a simple structure:

1. define all variables and shocks
2. provide the nonlinear equations. Note that each equation starts with a ~.
3. define the parameters
4. define the values of the parameters in the *steady_state* section
5. optionally provide some steady state values and/or values for initial guesses
6. optionally provide auxilliary equations that are not directly part of the nonlinear system (see the yaml for the BH model)

I will first briefly discuss the yaml of the small scale representative agents model above and then turn to more complex HANK model.

2.2.1 Representative agent models

The file for the small scale NK model can be found here. The first block is self explanatory:

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

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

The second block defines general definitions and imports, which are available at all times.

```
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 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`.

```
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.

```
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`.

2.2.2 Heterogeneous agent models

Let us have a look of the yaml of a hank model we will discuss in the tutorial. The file can be found here. The first line reads:

```
functions_file: '../examples/hank_functions.py'
```

The relative path to a functions-file, which may provide additional functions. In this example, the file defines the functions `transfers`, `wages`, `hh`, `labor_supply` and `hh_init`.

```
# these are available during all three stages (decisions, distributions, equations)
definitions: |
    from jax.numpy import log, maximum
    from jax.experimental.host_callback import id_print as jax_print
```

General definitions and imports (as above). These are available during all three stages (decisions, distributions, equations).

```
variables: [Div, Y, Yprod, w, pi, Rn, Rs, R, Rstar, Tax, Z, beta, vphi, C, L, 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.

```
distributions:
  dist: # the name of the first distribution
    # ordering matters. The ordering here is corresponds to the shape of the axis of the␣
↪distribution
    skills: # first dimension
      type: exogenous
      grid_variables: [skills_grid, skills_stationary, skills_transition] # returns␣
↪skills_grid, skills_stationary, skills_transition
      rho: 0.966
      sigma: 0.6
      n: 4
    a: # second dimension
      type: endogenous
      grid_variables: a_grid # a variable named a_grid will be made available during␣
↪decisions calls and distributions calls
      min: 0.0
      max: 50
      n: 40
```

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

```
decisions: # stage one: iterating the decisions function backwards
  inputs: [VaPrime] # additional to all aggregated variables defined in 'variables'
  calls: |
    # these are executed subsequently, starting with the last in time T and then␣
↪iterating forwards
    # Each call takes the previous outputs as given
    T = transfers(skills_stationary, Div, Tax, skills_grid)
    VaPrimeExp = skills_transition @ VaPrime
    Va, a, c = hh(VaPrimeExp, a_grid, skills_grid, w, n, T, R, beta, eis, frisch)
  # the 'outputs' values are stored for the following stages
  # NOTE: each output must have the same shape as the distribution (4,40)
  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.

```
# stage three (optional): aux_equations
aux_equations: |
    A = jnp.sum(dist*a, axis=(0,1)) # note that we are summing over the first two␣
↪dimensions e and a, but not the time dimension (dimension 2)
    aggr_c = jnp.sum(dist*c, axis=(0,1))
    # `dist` here corresponds to the dist from the *previous* period.


    # calculate consumption share of top-10% cumsumers
    c_flat = c.reshape(-1,c.shape[-1]) # consumption flattend for each t
    dist_sorted_c = jnp.take_along_axis(dist.reshape(-1,c.shape[-1]), jnp.argsort(c_flat,
↪ axis=0), axis=0) # distribution sorted after consumption level, flattend for each t
    top10c = jnp.where(jnp.cumsum(dist_sorted_c, axis=0) > .9, c_flat, 0.).sum(0)/c_flat.
↪sum(axis=0) # must use `where` for jax. All sums must be taken over the non-time axis
```

(continues on next page)

(continued from previous page)

```
    # calculate wealth share of top-10% wealth holders
    a_flat = a.reshape(-1,a.shape[-1]) # assets flattend for each t
    dist_sorted_a = jnp.take_along_axis(dist.reshape(-1,a.shape[-1]), jnp.argsort(a_flat,
↪ axis=0), axis=0) # as above
    top10a = jnp.where(jnp.cumsum(dist_sorted_a, axis=0) > .9, a_flat, 0.).sum(0)/a_flat.
↪sum(axis=0)
```

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.

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

    # firms
    ~ n = Yprod / Z # production function
    ~ Div = - w * n + (1 - psi*(pi/piSS - 1)**2/2)*Yprod # dividends
    ~ Y = (1 - psi*(pi/piSS - 1)**2/2)*Yprod # "effective" output
    ~ psi*(pi/piSS - 1)*pi/piSS = (1-theta) + theta*w + psi*piPrime/Rn*(piPrime/piSS -␣
↪1)*piPrime/piSS*YprodPrime/Yprod # NKPC

    # government
    ~ R = RsLag/pi # real rate ex-post
    ~ Rs = (Rstar*((pi/piSS)**phi_pi)*((Y/YLag)**phi_y))**(1-rho)*RsLag**rho # MP rule␣
↪on shadow nominal rate
    ~ Rn = maximum(1, Rs) # ZLB
    ~ Tax = (R-1) * BLag # balanced budget

    # clearings
    ~ C = Y # market clearing
    ~ B = A # bond market clearing
    ~ w**frisch = n # labor market clearing

    # exogenous
    ~ beta = betaSS*(betaLag/betaSS)**rho_beta # exogenous beta
    ~ Rstar = RstarSS*(RstarLag/RstarSS)**rho_rstar # exogenous rstar
    ~ Z = ZSS*(ZLag/ZSS)**rho_Z # exogenous technology
```

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

```
parameters: [ eis, frisch, theta, psi, phi_pi, phi_y, rho, rho_beta, rho_rstar, rho_Z ]
```

Define the model parameters, as above.

```
steady_state:
    fixed_values:
        # parameters:
        eis: 0.5
        frisch: 0.5
        theta: 6.
        psi: 96
        phi_pi: 1.5
        phi_y: .25
        rho: .8
        rho_beta: .9
        rho_rstar: .9
        rho_Z: .8

        # steady state
        Y: 1.0
        pi: 1.0
        beta: 0.97
        B: 5.6
        w: (theta-1)/theta
        n: w**frisch

    init_guesses:
        Rstar: 1.002
        Div: 1 - w
        Tax: 0.028
        R: Rstar
        VaPrime: hh_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 `VaPrime`).

2.3 Boehl-Hommes method

The package also contains an alternative “shooting” method much aligned to the one introduced in Boehl & Hommes (2021). In the original paper we use this method to solve for chaotic asset price dynamics. The method can be understood as a policy function iteration where the initial state is the only fixed grid point and all other grid points are chosen endogenously (as in a “reverse” EGM) to map the expected trajectory.

The main advantage (in terms of robustness) over the stacking method comes from exploiting the property that most determined perfect foresight models are a contraction mapping both, forward and backwards. The model is given by

```
f(x_{t-1}, x_t, x_{t+1}) = 0.
```

We iterate on the expected trajectory itself instead of the policy function. We hence require

```
d f(x_{t-1}, x_t, x_{t+1} ) < d x_{t-1},
d f(x_{t-1}, x_t, x_{t+1} ) < d x_{t+1}.
```

This is also the weakness of the method: not every DSGE model (that is determined in the Blanchard-Kahn sense) is such backward-and-forward contraction. In most cases the algorithm converges anyways, but convergence is not guaranteed.

The following example shows how to use the shooting method on the simple New Keynesian model.

```
import numpy as np
import matplotlib.pyplot as plt
import econpizza as ep
from econpizza import example_nk

# load the example.
# example_nk is nothing else but the path to the yaml, hence you could also use␣
↪`filename = 'path_to/model.yaml'`
mod = ep.load(example_nk)
# solve for the steady state
_ = mod.solve_stst()

# get the steady state as an initial state
state = mod['stst'].copy()
# increase the discount factor by one percent
state['beta'] *= 1.02

# simulate the model
x, _, flag = mod.find_path(state.values())

# plotting
for i,v in enumerate(mod['variables']):

    plt.figure()
    plt.plot(x[:,i])
    plt.title(v)
```

Lets go for a second, numerically more challenging example: the chaotic rational expectations model of Boehl & Hommes (2021)

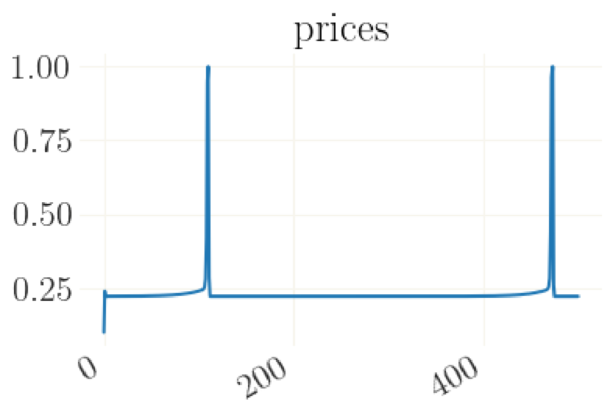
```
import numpy as np
import matplotlib.pyplot as plt
import econpizza as ep
from econpizza import example_bh

# parse the yaml
mod = ep.load(example_bh, raise_errors=False)
_ = mod.solve_stst()

# choose an interesting initial state
state = np.zeros(len(mod['variables']))
state[:-1] = [.1, .2, 0.]

# solve and simulate. The lower eps is not actually necessary
x, _, flag = ep.find_path(mod, state, T=500, max_horizon=1000, tol=1e-8)

# plotting
for i,v in enumerate(mod['variables']):
```

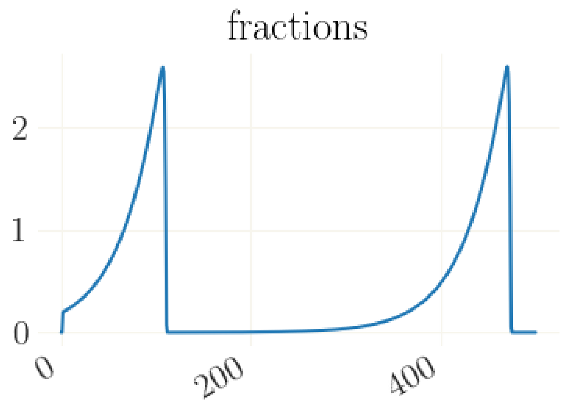
(continues on next page)

(continued from previous page)

```
    plt.figure()
    plt.plot(x[:,i])
    plt.title(v)
```

This will give you boom-bust cycles in asset pricing dynamics:





CHAPTER

THREE

HANK TUTORIAL

The package supports heterogeneous agent models with and without porfolio chocie (i.e., with one asset and two assets, respectively). Both examples are keept relatively close to the ones used in the Sequence-Space Jacobian package for reasons of comparability.

While for models without heterogenous agents, the calculation of the sequence-space jacobian for the nonlinear extended path is already speed and memmory optimized, this is not yet the case for HANK (but would be relatively straightforward).

Start with some misc imports and load the package:

[1]:
```
import jax.numpy as jnp # use jax.numpy instead of normal numpy. Yes, this is one of the␣
↪reasons why it is nice not to have all numpy objects imported on the lowest hirarchy
from grgrlib import figurator,grplot # a nice backend for batch plotting with matplotlib
import econpizza as ep # pizza
import matplotlib.pyplot as plt


# for nicer text in figures
plt.rc('text', usetex=True)
# only necessary if you run this in a jupyter notebook:
%matplotlib inline
```

3.1 One-Asset HANK

The provided example is the same model as used by Auclert et al., 2022, which is documented in this notebook (and in the appendix of their paper). There are some deviations.

First, I use GHH preferences (Greenwood et al., 1988):

$$u(c_{i,t}, n_{i,t}; z_{i,t}) = \frac{x_{i,t}^{1-\sigma}}{1-\sigma},$$

with the composite good $x_{i,t}$ defined as

$$x_{i,t} = c_{i,t} - z_{i,t}\frac{n_{i,t}^{1+\varphi}}{1+\varphi}.$$

A version with standard preferences and individual labor choice as in Auclert et al. (2022) is provided as `example_hank_labor`, which links to this yaml file.

Second, the NK-Phillips Curve is the conventional nonlinear Phillips Curve as derived from Rothemberg pricing,

$$\psi\left(\frac{\pi_t}{\pi_{SS}} - 1\right)\frac{\pi_t}{\pi_{SS}} = (1-\theta) + \theta w_t + E_t\left\{\psi\frac{\pi_{t+1}}{R_t}\left(\frac{\pi_{t+1}}{\pi_{SS}} - 1\right)\frac{\pi_{t+1}}{\pi_{SS}}\frac{Y_{t+1}^p}{Y_t^p}\right\},$$

where the inverse real rate $\frac{\pi_{t+1}}{R_t}$ is used to avoid a representative household discount factor (e.g, $\beta_{t+1}\frac{C_t}{C_{t+1}}$).

Third, the central bank sets the *shadow rate* to follow a conventional monetary policy rule with interest rate inertia,

$$R_{s,t} = \left[R^* \left(\frac{\pi_t}{\pi_{SS}}\right)^{\phi_\pi} \left(\frac{Y_t}{Y_{t-1}}\right)^{\phi_y}\right]^{1-\rho} R^{\rho}_{s,t-1},$$

and the actual nominal interest rate equals this *shadow rate* subject to the zero lower bound on nominal interest rates,

$$R_t = \max\{1, R_{s,t}\}$$

Details can be found in the section on the `yaml` file above.

Load the example file:

[2]:
```
from econpizza import example_hank
```

`example_hank` is nothing else than the path to the yaml file we discussed before:

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

```
/home/gboehl/github/econpizza/econpizza/examples/hank.yaml
```

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 8.14e-01 | lapsed 2.9173
    Iteration   2 | max error 4.32e-01 | lapsed 2.9820
    Iteration   3 | max error 1.83e-01 | lapsed 3.0093
    Iteration   4 | max error 5.61e-02 | lapsed 3.0345
    Iteration   5 | max error 9.06e-03 | lapsed 3.0574
    Iteration   6 | max error 9.71e-04 | lapsed 3.0798
    Iteration   7 | max error 2.47e-08 | lapsed 3.1020
(solve_stst:) Steady state found in 3.7059 seconds. 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 you need to fix some steady state values. 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', 'det'])
```

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

```
[-1.50512003e-09 -4.16333634e-16  5.55111512e-17  1.11022302e-16
 -1.50509302e-09  1.50509305e-09 -2.22044605e-16  0.00000000e+00
  1.24359332e-16  0.00000000e+00  1.50509312e-09  1.50509338e-09
  2.04964934e-11  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']
```

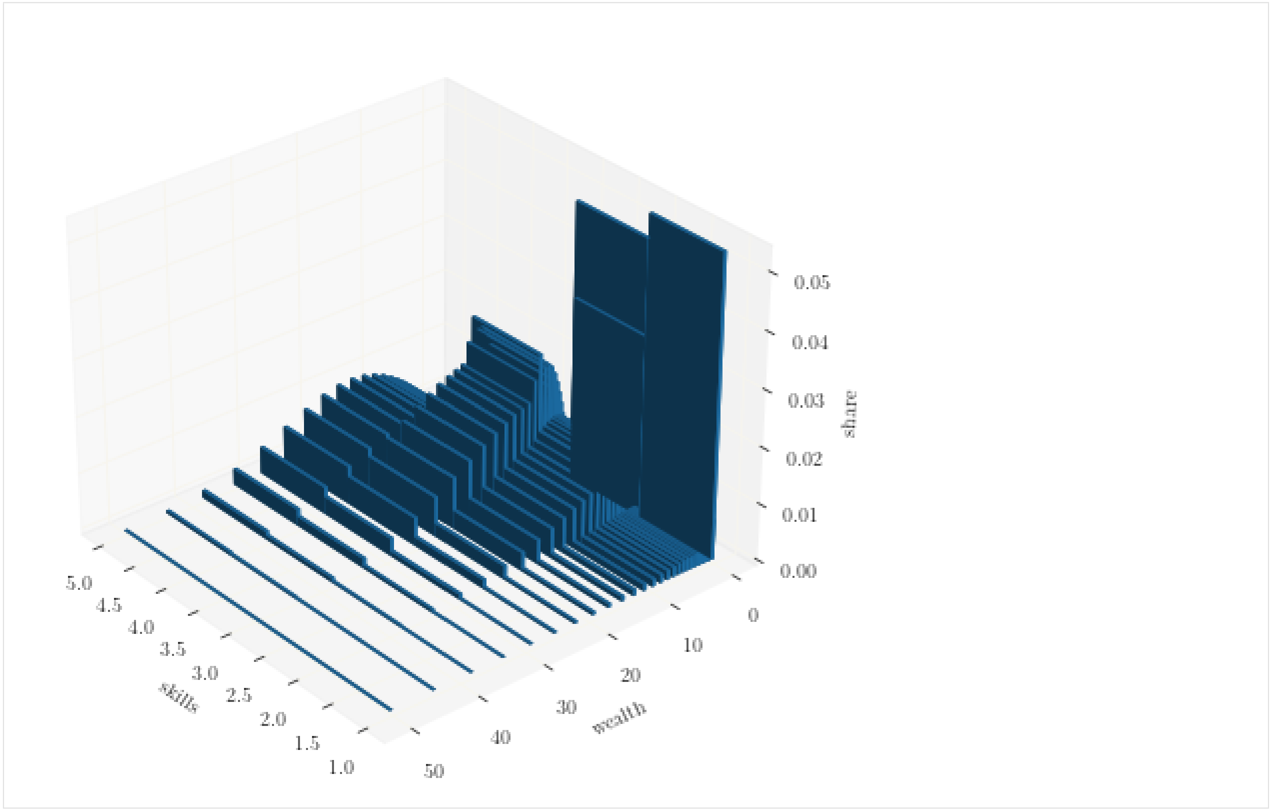
```
[8]: {'B': DeviceArray(5.6, dtype=float64),
      'beta': DeviceArray(0.97, dtype=float64),
      'C': DeviceArray(1., dtype=float64),
      'Div': DeviceArray(0.23927422, dtype=float64),
      'n': DeviceArray(0.91287093, dtype=float64),
      'pi': DeviceArray(1., dtype=float64),
      'R': DeviceArray(1.01427653, dtype=float64),
      'Rn': DeviceArray(1.01427653, dtype=float64),
      'Rs': DeviceArray(1.01427653, dtype=float64),
      'Rstar': DeviceArray(1.01427653, dtype=float64),
      'Tax': DeviceArray(0.07994855, dtype=float64),
      'Top10A': DeviceArray(0.26070033, dtype=float64),
      'Top10C': DeviceArray(0.2355648, dtype=float64),
      'w': DeviceArray(0.83333333, dtype=float64),
      'Y': DeviceArray(1., dtype=float64),
      'Yprod': DeviceArray(1., dtype=float64),
      'Z': DeviceArray(1.09544511, 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']
```

Under the dict behind the `context` keyword, many other model specific variables are stored. Better have a look youself if you care. Lets plot the distribution:

```
[10]: 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.

Okay, let's continue with calculating some impulse response functions. We'll have a look at a shock to the households' discount factor β. Take the steady state as the initial value, and alter the value of β:

```
[11]: # this is a dict containing the steady state values
x0 = hank1['stst'].copy()
# setting a large shock on the discount factor
x0['beta'] = 1.
```

Find the IRFs:

```
[12]: xst, _, flags = hank1.find_stack(x0.values(), horizon=100, tol=1e-8)
```

```
(find_path_stacked:) Solving stack (size: 1700)...
    Iteration   1 | max error 3.30e-01 | lapsed 4.3197
    Iteration   2 | max error 1.79e-01 | lapsed 5.5242
    Iteration   3 | max error 5.28e-02 | lapsed 6.7160
    Iteration   4 | max error 9.15e-03 | lapsed 7.9173
    Iteration   5 | max error 6.63e-06 | lapsed 9.1092
(find_path_stacked:) Stacking done after 11.741 seconds. The solution converged.
```

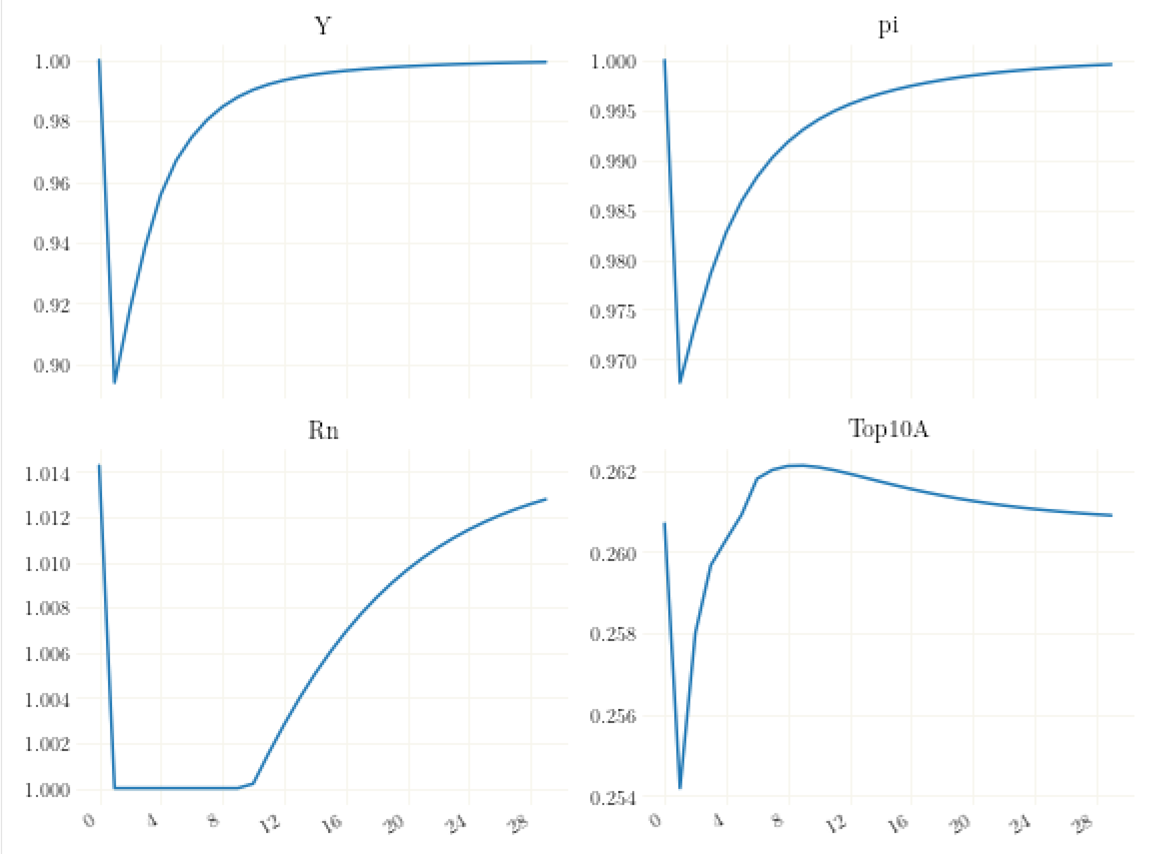
That went smoothly. Again, you will get meaningful and (hopefully) infomative final messages. Let's plot only a few of the variables for space restrictions: output y_t (Y), inflation π_t (pi), the nominal interst rate R_t (Rn), and the percentage share of wealth held by the top-10% richest, Top10A.

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

(continues on next page)

(continued from previous page)

```
variables = 'Y', 'pi', 'Rn', '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 and 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.

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

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

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

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

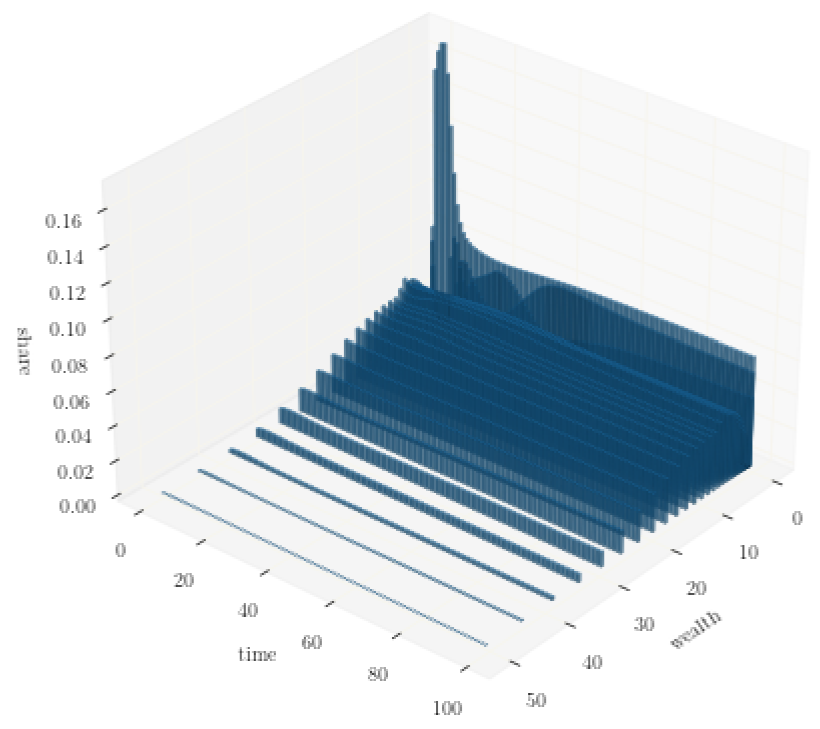
Each of the objects has shape `*distribution shape, number_of_periods`:

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

```
(4, 40, 99)
```

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

```
[17]: dist = het_vars['dist']
a_grid = hank1['context']['a_grid']
# plot
ax, _ = grbar3d(dist.sum(0), xedges=a_grid, yedges=jnp.arange(dist.shape[-1]),␣
↪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 liquid assets, causing them to hold more assets.

3.2 Two-Asset HANK

This second example is the two-asset HANK model from Auclert et al., 2022, which is documented here.

There are, again, some deviations. First, the NK-Phillips Curve is the conventional nonlinear Phillips Curve as derived from Rothemberg pricing,

$$\psi\left(\frac{\pi_t}{\pi_{SS}}-1\right)\frac{\pi_t}{\pi_{SS}} = (1-\theta)+\theta\widehat{MC}_t + E_t\left\{\psi\frac{\pi_{t+1}}{R_t}\left(\frac{\pi_{t+1}}{\pi_{SS}}-1\right)\frac{\pi_{t+1}}{\pi_{SS}}\frac{Y^p_{t+1}}{Y^p_t}\right\},$$

which also features the inverse real rate $\frac{\pi_{t+1}}{R_t}$.

Importantly, the wage Phillips Curve is modified in a similar fashion.

Further, the central bank sets the nominal interest rate to follow a conventional monetary policy rule with interest rate inertia,

$$R_{s,t} = \left[R^* \left(\frac{\pi_t}{\pi_{SS}}\right)^{\phi_\pi} \left(\frac{Y_t}{Y_{t-1}}\right)^{\phi_y}\right]^{1-\rho} R^{\rho}_{s,t-1}.$$

Finally, capital adjustment costs are defined as in the Smets & Wouters Model (but without capital utilization):

$$E_t R_{t+1} q_t = \alpha E_t \left\{ Z_{t+1} \left(\frac{N_{t+1}}{K_t}\right)^{1-\alpha} \widehat{MC}_{t+1} \right\} + (1-\delta) E_t q_{t+1}, \tag{3.1}$$

$$1 = q_t \left[1 - S\left(\frac{I_t}{I_{t-1}}\right) - S'\left(\frac{I_t}{I_{t-1}}\right)\frac{I_t}{I_{t-1}}\right] + E_t \left\{ \frac{q_{t+1}}{R_{t+1}} S'\left(\frac{I_{t+1}}{I_t}\right)\left(\frac{I_{t+1}}{I_t}\right)^2 \right\} \tag{3.2}$$

The `yaml` file of the model can be found here. Let's quickly run this model in a similar fashion as above.

[18]:
```
from econpizza import example_hank2
# parse model
hank2_dict = ep.parse(example_hank2)
# compile the model
hank2 = ep.load(hank2_dict)
```

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

[19]:
```
stst_result = hank2.solve_stst()
```

```
    Iteration   1 | max error 1.19e+01 | lapsed 7.0374
    Iteration   2 | max error 4.82e-01 | lapsed 7.4848
    Iteration   3 | max error 1.35e-02 | lapsed 7.8852
    Iteration   4 | max error 7.42e-04 | lapsed 8.2888
    Iteration   5 | max error 3.63e-08 | lapsed 8.6953
(solve_stst:) Steady state found in 11.289 seconds. The solution converged.
```

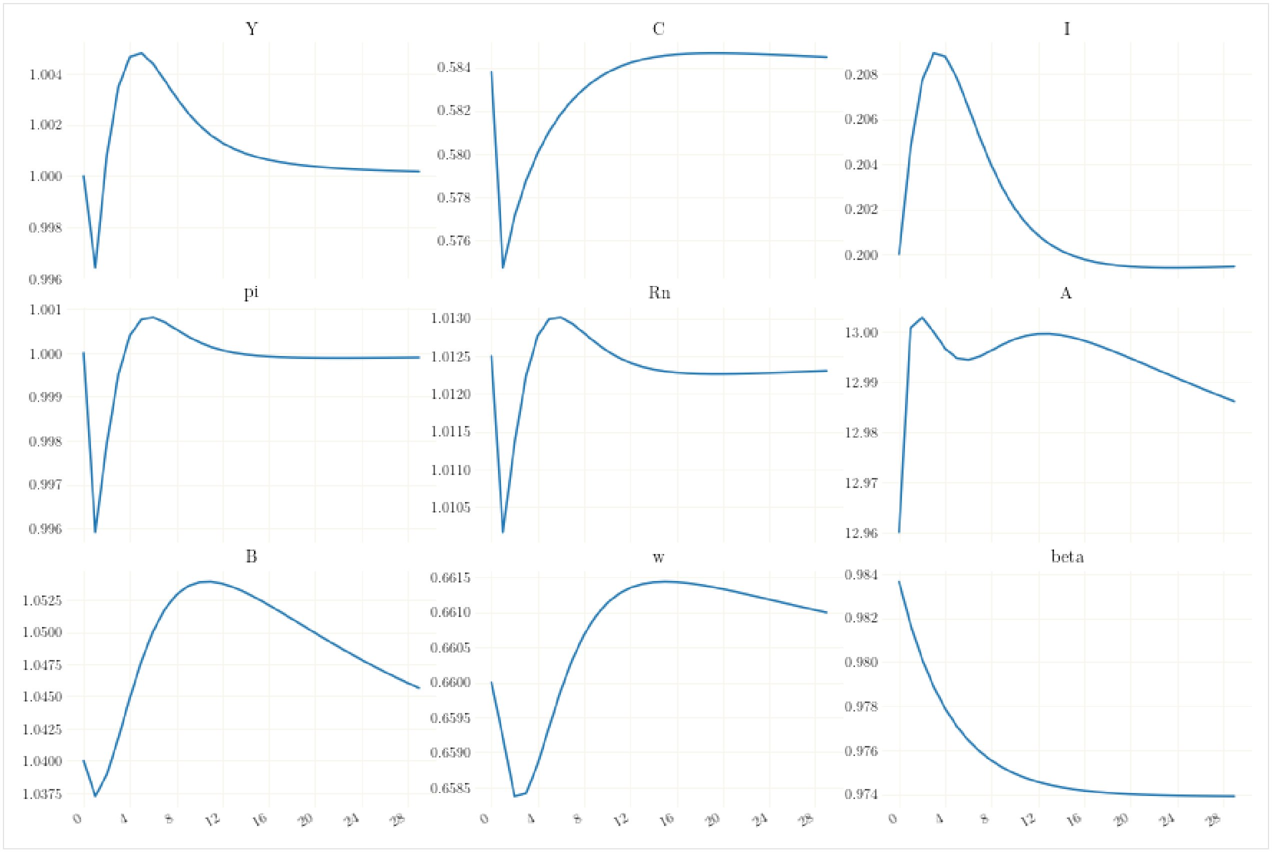
Again, look at a discount factor shock and calculate the pefect foresight solution:

[20]:
```
# this is a dict containing the steady state values
x0 = hank2['stst'].copy()
# setting a shock on the discount factor
x0['beta'] *= 1.01
```

[21]:
```
xst, _, flags = hank2.find_stack(x0.values(), horizon=100, tol=1e-8)
```

```
(find_path_stacked:) Solving stack (size: 2400)...
    Iteration   1 | max error 5.42e-02 | lapsed 27.2941
    Iteration   2 | max error 3.67e-03 | lapsed 41.7114
    Iteration   3 | max error 3.92e-05 | lapsed 56.0187
(find_path_stacked:) Stacking done after 85.550 seconds. The solution converged.
```

[22]:
```
variables = 'Y', 'C', 'I', 'pi', 'Rn', 'A', 'B', 'w', 'beta'
inds = [hank2['variables'].index(v) for v in variables]

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



This ends this short tutorial. Further details on the implementation of heterogeneous agent models are given in the technical section.

CHAPTER

FOUR

UNDER THE HOOD

The methodology extends Auclert et al. (2022, ECMA) to solve the system fully nonlinearly. Most heterogeneous agent models can be expressed in terms of the three functions V, D and F:

$$v_t = V(x_{t-1}, x_t, x_{t+1}, v_{t+1}) \tag{4.1}$$

$$d_t = D(x_{t-1}, x_t, x_{t+1}, v_t, d_{t-1}) \tag{4.2}$$

$$0 = f(x_{t-1}, x_t, x_{t+1}, d_t, v_t) \tag{4.3}$$

where x_t is the time-t vector of *aggregate* variables, with each aggregate variables $x_{i,t}$ indexed over $i \in 1, 2, \ldots, N$. v_t containts the idiosyncratic agents decision functions and d_t is the distribution across agents.

4.1 The steady state

Let the steady state values be denoted by a bar, i.e. $\bar{x}$, $\bar{d}$ and $\bar{v}$. The steady state is often not unique. In this case it is necessary to fix some variables ex-ante, i.e. to given values. Denote the set of indices of the values to be fixed as K and the fixed values as $\hat{x}$. Further, let $b(x)$ be a function that sets $x_K = \hat{x}$, i.e.

$$b(x) = \{x_{\notin K} \wedge \hat{x}\}.$$

The steady state then satisfies

$$\bar{v} = V(\bar{x}, \bar{x}, \bar{x}, \bar{v}) \tag{4.4}$$

$$\bar{d} = D(\bar{x}, \bar{x}, \bar{x}, \bar{v}, \bar{d}) \tag{4.5}$$

$$0 = f(\bar{x}, \bar{x}, \bar{x}, \bar{d}, \bar{v}) \tag{4.6}$$

$$\bar{x} = b(\bar{x}) \tag{4.7}$$

For a given guess on $\bar{x}$, the first equation can be solved for $\bar{v}$ via backwards iteration. Denote the function that solves this as $\bar{v} = \bar{V}(\bar{x})$. Given $\bar{x}$ and $\bar{v}$, the second equation can be solved via forward iteration (or via eigenvector, but eigenvectors are not yet available for automatic differentiation). Denote this solver as $\bar{d} = \bar{D}(\bar{x}, \bar{v})$. Define $\bar{f}$ equivalently.

Combining those, $\bar{x}$ must satisfy a function H defined by

$$H(\bar{x}) = \bar{f}\left(b(\bar{x}), \bar{D}\left(b(\bar{x}), \bar{V}(b(\bar{x}))\right), \bar{V}(b(\bar{x}))\right) = 0.$$

We can calculate the Jacobian of H using automatic differentiation and, starting with some guess X^j on $\bar{x}$, we can iterate on H using a Newton method. During iteration we use the Moore–Penrose inverse for the inversion of the Jacobian since $\bar{f}$ may not have full rank, in which case it is necessary that $K \neq \{\}$, from which it follows that $b(x)$ also does not have full rank.

4.2 Dynamic solution

Let, for the sake of generality, letters with a tilde denote the complete expected time series/sequence of a variable up to some distant point T in the future, e.g $\tilde{x} = \{x_t\}_{t=0}^{T}$. The above system can be written as

$$v_t = V(\tilde{x}, v_{t+1}) \tag{4.8}$$

$$d_t = D(\tilde{x}, \tilde{v}, d_{t-1}) \tag{4.9}$$

$$0 = F(\tilde{x}, \tilde{d}, \tilde{v}) \tag{4.10}$$

with $F = \{f(x_{t-1}, x_t, x_{t+1}, d_t, v_t)\}_{t=1}^{T}$.

V often takes the form of a value function. Note that the above assumes that V is independent of the distribution at any point in time. Assuming that after T we are back in the steady state, the sequence $\tilde{v}$ can be found by starting with

$$v_T = V(\tilde{x}, \bar{v}) \tag{4.11}$$

$$v_{T-1} = V(\tilde{x}, v_T) \tag{4.12}$$

$$\dots = V(\tilde{x}, v_{T-1}) \tag{4.13}$$

and iterating backwards until we arrive at v_0. This yields the sequence $\tilde{v}$.

Now taking v_0 as given and assuming we are also in the steady state in $t = 0$, we can start with $d_{-1} = \bar{d}$ and iterate D forward:

$$d_0 = D(\tilde{x}, \tilde{v}, \bar{d}) \tag{4.14}$$

$$d_1 = D(\tilde{x}, \tilde{v}, d_0) \tag{4.15}$$

$$\dots = D(\tilde{x}, \tilde{v}, d_2) \tag{4.16}$$

until we arrive at d_T. This yields $\tilde{d}$.

Denote the function that generates $\tilde{v}$ from $\tilde{x}$ as $\hat{V}$, and a function that generates $\tilde{d}$ from $\tilde{x}$ and $\tilde{v}$ as $\hat{D}$. Then we can define

$$G(\tilde{x}) = F\left(\tilde{x}, \hat{D}\left(\tilde{x}, \hat{V}(\tilde{x})\right), \hat{V}(\tilde{x})\right)$$

as a function that only depends on the sequence $\tilde{x}$.

We are looking for a sequence $\tilde{x}$ of aggregate variables such that $G(\tilde{x}) = 0$. Given that the jacobian J_G of $G(\cdot)$ is available via automatic differentiation, we can then use a Newton method to solve for $\tilde{x}$ directly without having to explicitely keep track of any distribution variable. Note that (a) J_G is sparse, so we can make use of sparse linear algebra when solving for the next Newton guess. Also note (b) that not all entries of f depend on aggregated heterogeneous inputs, which can be used to calculate most of the individual blocks of G_J directly from f instead of having to diffentiate the complete function F, which is computationally relatively expensive.

4.3 Implementation

The above functions are written dynamically during parsing (in `parsing.py`).

[1]:
```
import econpizza as ep
from econpizza import example_hank
```

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

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

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

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

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

The function `func_backw` corresponds to function $V(\cdot)$ above, `func_stst_dist` is $\bar{D}(\cdot)$, `func_dist` is $D(\cdot)$, `func_pre_stst` corresponds to $b(\cdot)$, and `func_eqns` is $f(\cdot)$. F and $\bar{f}$ are created dynamically in `stacking.py` and `steady_state.py`.

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, LLag, piLag, rLag, rnLag, rstarLag, TaxLag, vphiLag, wLag,
↪ YLag, YprodLag, ZLag, ) = XLag

 (B, beta, C, Div, L, pi, r, rn, rstar, Tax, vphi, w, Y, Yprod, Z, ) = X

 (BPrime, betaPrime, CPrime, DivPrime, LPrime, piPrime, rPrime, rnPrime, rstarPrime,␣
↪TaxPrime, vphiPrime, wPrime, YPrime, YprodPrime, ZPrime, ) = XPrime

 (BSS, betaSS, CSS, DivSS, LSS, piSS, rSS, rnSS, rstarSS, TaxSS, vphiSS, wSS, YSS,␣
↪YprodSS, ZSS, ) = XSS

 (eis, frisch, rho_e, sd_e, mu, theta, psi, phi_pi, phi_y, rho, rho_beta, rho_rstar, rho_
↪Z, ) = pars

 () = shocks

 (D, ) = distributions

 (a, c, ne, ) = decisions_outputs

 A = jnp.sum(D*a, axis=(0,1))
 NE = jnp.sum(D*ne, axis=(0,1))
 Caggr = jnp.sum(D*c, axis=(0,1))

 root_container0 = L - ( Yprod / Z)
 root_container1 = Div - ( - w * L + (1 - psi*(pi/piSS - 1)**2/2)*Yprod)
 root_container2 = C - ( (1 - psi*(pi/piSS - 1)**2/2)*Yprod)
 root_container3 = C - ( Y)
 root_container4 = C - ( Caggr)
 root_container5 = psi*(pi/piSS - 1)*pi/piSS - ( (1-theta) + theta*w + psi*betaPrime*C/
↪CPrime*(piPrime/piSS - 1)*piPrime/piSS*YprodPrime/Yprod)
 root_container6 = r - ( rnLag/pi)
 root_container7 = rn - ( (rstar*((pi/piSS)**phi_pi)*((Y/YLag)**phi_y))**(1-
↪rho)*rnLag**rho)
```

(continues on next page)

(continued from previous page)

```
root_container8 = Tax - ( (r-1) * B)
root_container9 = B - ( A)
root_container10 = NE - ( L)
root_container11 = vphi - ( vphiSS)
root_container12 = beta - ( betaSS*(betaLag/betaSS)**rho_beta)
root_container13 = rstar - ( rstarSS*(rstarLag/rstarSS)**rho_rstar)
root_container14 = Z - ( ZSS*(ZLag/ZSS)**rho_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]).ravel()
```

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

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

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

CHAPTER

FIVE

MODULE DOCUMENTATION

5.1 `econpizza.load`

Functions for model parsing yaml into a working model instance. Involves a lot of dynamic function definition…

`econpizza.parsing.`**`compile_init_values`**`(`*evars*, *pars*, *initvals*, *stst*`)`

Combine all available information in initial guesses.

`econpizza.parsing.`**`define_function`**`(`*func_str*, *context*`)`

Define functions from string. Writes the function into a temporary file in order to get meaningful debug traces.

`econpizza.parsing.`**`define_subdict_if_absent`**`(`*parent*, *sub*`)`

`econpizza.parsing.`**`eval_strs`**`(`*vdict*, *context={}*`)`

Evaluate a dictionary of strings into a given context

`econpizza.parsing.`**`get_exog_grid_var_names`**`(`*distributions*`)`

WIP. So far unused.

`econpizza.parsing.`**`initialize_context`**`()`

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

Load model from dict or yaml file.

Parameters

- **`model`** (`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 silently for debugging. Defaults to True
- **`verbose`** (`bool, optional`) – inform that parsing is done. Defaults to True

Returns

model – The parsed model

Return type

PizzaModel instance

`econpizza.parsing.`**`load_external_functions_file`**`(`*model*, *context*`)`

Load the functions file as a module.

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

parse from yaml file

`econpizza.parsing.`**`parse_external_functions_file`**`(`*model*`)`

Parse the functions file.

5.2 econpizza.find_stack

econpizza.stacking.**find_stack**(*model*, *x0=None*, *shock=None*, *init_path=None*, *horizon=250*, *tol=None*, *maxit=None*, *use_linear_guess=True*, *use_linear_endpoint=None*, *use_jacrev=True*, *verbose=True*, ***solver_kwargs*)

Find the expected trajectory given an initial state.

Parameters

- **model** (*dict*) – model dict or PizzaModel instance
- **x0** (*array*) – initial state
- **shock** (*tuple, optional*) – shock in period 0 as in *(shock_name_as_str, shock_size)*
- **init_path** (*array, optional*) – a first guess on the trajectory. Should not be necessary
- **horizon** (*int, optional*) – number of periods until the system is assumed to be back in the steady state. A good idea to set this corresponding to the respective problem. A too large value may be computationally expensive. A too small value may generate inaccurate results
- **tol** (*float, optional*) – convergence criterion. Defaults to 1e-8
- **maxit** (*int, optional*) – number of iterations. Default is 30.
- **use_linear_guess** (*bool, optional*) – whether to use the linear impulse responses as an initial guess. Defaults to True if the linear LOM is known
- **use_linear_endpoint** (*bool, optional*) – whether to use the linear impulse responses as the final state. Defaults to True if the linear LOM is known
- **use_jacrev** (*bool, optional*) – whether to use reverse mode or forward mode automatic differentiation. By construction, reverse AD is faster, but does not work for all types of functions. Defaults to True
- **verbose** (*bool, optional*) – degree of verbosity. 0/*False* is silent
- **solver_kwargs** (*optional*) – any additional keyword arguments will be passed on to the solver

Returns

- **x** (*array*) – array of the trajectory
- **x_lin** (*array or None*) – array of the trajectory based on the linear model. Will return None if the linear model is unknown
- **flag** (*bool*) – returns True if the solver was successful, else False

5.3 econpizza.find_pizza

econpizza.shooting.**find_path_linear**(*model*, *shock*, *T*, *x*, *use_linear_guess*)

Solves the expected trajectory given the linear model.

econpizza.shooting.**find_pizza**(*model*, *x0=None*, *shock=None*, *T=30*, *init_path=None*, *max_horizon=200*, *max_loops=100*, *max_iter=None*, *tol=1e-05*, *use_linear_guess=False*, *root_options={}*, *raise_error=False*, *verbose=True*)

Find the expected trajectory given an initial state. A good strategy is to first set *tol* to a low value (e.g. 1e-3) and check for a good max_horizon. Then, set max_horizon to a reasonable value and let max_loops be high.

Parameters

- **model** (*dict*) – model dict or PizzaModel instance
- **x0** (*array*) – initial state
- **shock** (*tuple, optional*) – shock in period 0 as in *(shock_name_as_str, shock_size)*
- **T** (*int, optional*) – number of periods to simulate
- **init_path** (*array, optional*) – a first guess on the trajectory. Normally not necessary
- **max_horizon** (*int, optional*) – number of periods until the system is assumed to be back in the steady state. A good idea to set this corresponding to the respective problem. Note that a horizon too far away may cause the accumulation of numerical errors.
- **max_loops** (*int, optional*) – number of repetitions to iterate over the whole trajectory. Should eventually be high.
- **max_iterations** (*int, optional*) – number of iterations. Default is *max_horizon*. It should not be lower than that (and will raise an error). Normally it should not be higher, better use *max_loops* instead.
- **tol** (*float, optional*) – convergence criterion
- **root_options** (*dict, optional*) – dictionary with solver-specific options to be passed on to *scipy.optimize.root*
- **verbose** (*bool, optional*) – degree of verbosity. 0/*False* is silent

Returns

- **x_fin** (*array*) – array of the trajectory
- **x_lin** (*array or None*) – array of the trajectory based on the linear model. Will return None if the linear model is unknown
- **fin_flag** (*int*) – error code

econpizza.shooting.**solve_current**(*model*, *shock*, *XLag*, *XLastGuess*, *XPrime*, *tol*)

Solves for one period.

5.4 econpizza.solve_stst

econpizza.steady_state.**solve_stst**(*model*, *tol=1e-08*, *tol_newton=None*, *maxit_newton=30*, *tol_backwards=None*, *maxit_backwards=2000*, *tol_forwards=None*, *maxit_forwards=5000*, *force=False*, *verbose=True*, ***newton_kwargs*)

Solves for the steady state.

Parameters

- **tol_newton** (*float, optional*) – tolerance of the Newton method, defaults to 1e-8
- **maxit_newton** (*int, optional*) – maximum of iterations for the Newton method, defaults to 30
- **tol_backwards** (*float, optional*) – tolerance required for backward iteration. Defaults to tol_newton
- **maxit_backwards** (*int, optional*) – maximum of iterations for the backward iteration. Defaults to maxit_newton

- **tol_forwards** (*float, optional*) – tolerance required for forward iteration. Defaults to tol_newton
- **maxit_forwards** (*int, optional*) – maximum of iterations for the forward iteration. Defaults to maxit_newton
- **force** (*bool, optional*) – force recalculation of steady state, even if it is already evaluated. Defaults to False
- **verbose** (*bool, optional*) – level of verbosity. Defaults to True
- **newton_kwargs** (*keyword arguments*) – keyword arguments passed on to the Newton method

Returns
res – results dictionary from the Newton method

Return type
dict

`econpizza.steady_state.`**`solver`**(*jval*, *fval*)

A default solver to solve indetermined problems.

PYTHON MODULE INDEX

INDEX