Python Platform

I. INTRODUCTION

The growth in complexity and scale of macro-finance models over the past couple of decades, aided by computational advances, cannot be overstated. On the software side, specialized applications like DYNARE, the IRIS Macroeconomic Modelling Toolboxes, and TROLL have been developed to provide economists with an integrated platform for inputting their models, importing data, performing desired computational tasks (such as solving, simulating, calibrating, or estimating),

and obtaining well-formatted post-processed output in the form of tables, graphs, etc. Each application has its own advantages. The ease of use through a user-friendly interface, combined with the capability to handle a variety of models, has led to the immense popularity of DYNARE among general equilibrium modelers. However, DYNARE can only handle stationary DSGE models and requires users to write models in a stationary format by introducing variable deflators. The IRIS macroeconomic toolbox is another excellent tool that has gained popularity among economists for analyzing non-stationary DSGE models. TROLL, on the other hand, specializes in efficiently solving and simulating large systems of equations. All these applications, however, are either commercial or rely on commercial software that requires expensive licensing costs. To our knowledge, there is no integrated software package that is flexible enough to handle a wide range of models and available for free under the GNU General Public License agreements. This framework, built entirely on Python, is intended to fill that void. Additionally, the platform can read, parse model files developed by IRIS, DYNARE, TROLL, and Sirius software and run simulations. Users can also specify model variables, equations, and parameters on the fly that are not necessarily defined in a model file.

The Python platform is designed to analyze and estimate the following classes of theory-based dynamic models: New Keynesian DSGE, Real Business Cycle, Overlapping Generations, and Computable General Equilibrium. It also provides a toolbox to estimate time series models that can be cast into linear and non-linear state-space form. Like the other applications mentioned above, this framework is designed to be a fully integrated platform. To this end, users have the option to input their models in a human-readable format via a YAML (human-readable) file that also includes the data source. Options include directly exporting results to a CSV file, a Python SQLite database, and obtaining graphs and tables for the desired set of variables and parameters. Further details of these processing and output options are described in the following sections.

Below are the highlights of this framework:
- The framework is written in Python and uses only Python libraries that are available by installing the Python distribution from the Software Center.
- It can be run as a batch process, in a Jupyter notebook, or in a Spyder interactive development environment (Scientific Python Development environment).
- It is platform-agnostic and does not require adaptation for Windows, Linux, Unix, or Mac platforms.
- It is parallelized and can run on CPU/GPU cores.
- The model file incorporates complete details of the model: variables, parameters, their starting values for simulation or initialization in the case of calibration or estimation, equations linking the two, parameter bounds and prior distributions, data sources (Haver, ECOS, EDI, and World Bank databases), and options related to sample size, subsample choices, etc.
- It can estimate parameters of linear and non-linear DSGE models with rational expectations.
- Non-linear equations are solved iteratively using Newton's method. Two algorithms are implemented: the ABLR stacked matrices method and the LBJ forward-backward substitution method.
- Linear models are solved using Binder Pesaran's method, Anderson and More's method, and two generalized Schur's decomposition methods that reproduce calculations employed in DYNARE and IRIS software.
- Users can choose from various filtration methods (standard Kalman, unscented Kalman, diffuse Kalman, band pass, Hodrick–Prescott, LRX).
- Maximum likelihood estimates of parameters can be obtained.
- Bayesian estimation of parameters is available through MCMC-aided posterior sampling.
- Posterior sampling of parameters can be performed using Sequential Monte Carlo, affine invariant ensemble sampler algorithms, or the Random Walk Metropolis-Hastings algorithm.
- Non-linear models can be run with time-dependent parameters.
- The framework allows for the solving, calibration, and simulation of RBC and OLG models.
- It can generate forecasts and impulse response functions.

II. INSTALLATION

To install the package "snowdrop," navigate to the directory containing the tar file and execute the following command in the command line,

pip install snowdrop-0.1.0-py3-none-any.whl –user

III.   HARDWARE REQUIREMENTS

This code can run on both CPU and GPU across Windows, macOS, and Linux operating systems. For tasks that require high computational power, we recommend installing the CuPy library. CuPy is an open-source library designed for GPU-accelerated computing, utilizing NVIDIA's CUDA toolkit. This library will be employed by the platform's non-linear solver, provided that CuPy is installed and configured.

IV.   STRUCTURE OF THE PROGRAMS

This framework is developed in Python and utilizes several libraries, including numpy, pandas, scipy, sympy, matplotlib, ruamel.yaml, and ast. The first five libraries are primarily used for scientific computing and visualizing results, while the last two serve the purpose of parsing model files and representing mathematical expressions, as well as the overall structure of the code.
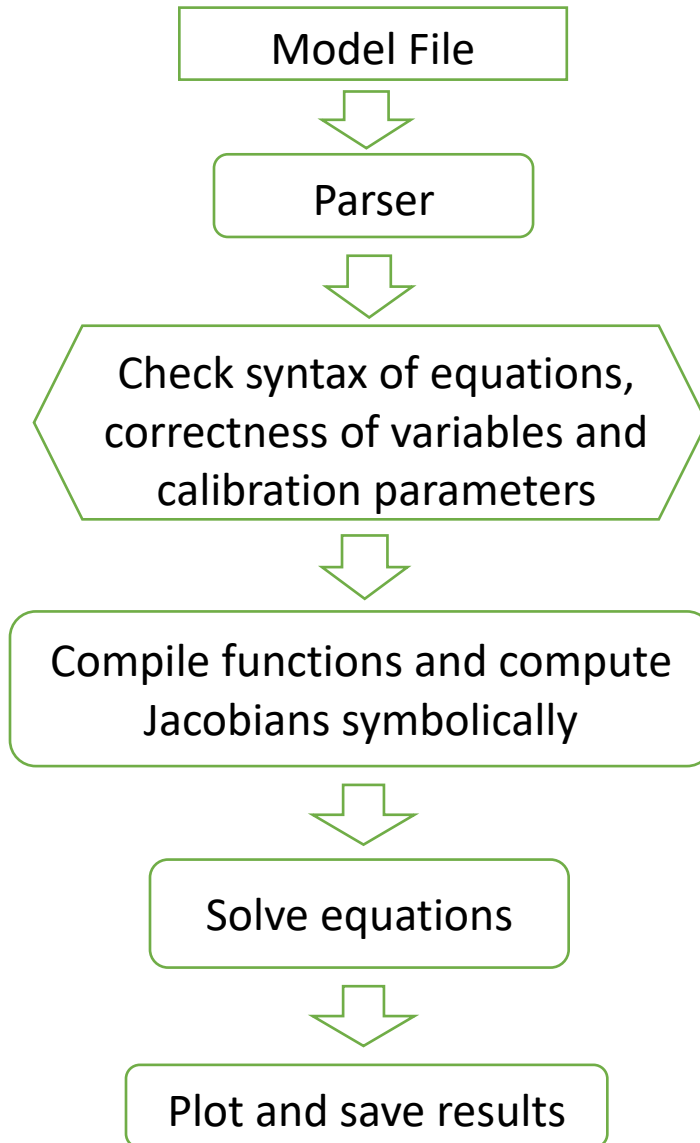

Fig. 1. Flowchart of Framework execution.

Each step of this flowchart is managed by a dedicated Python module. The code is written in an object-oriented programming style and is modularized to enhance reusability.

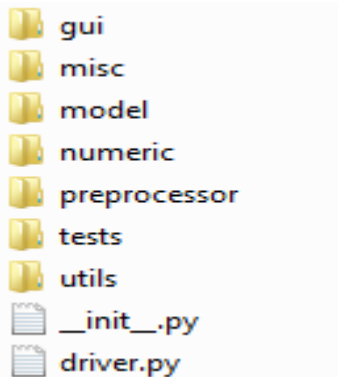The framework's *src* folder contains several subfolders, as illustrated below:

Fig. 2. The content of *src* folder.

The names of the subfolders are self-explanatory. For example, the **gui** folder contains a graphical user interface that assists users in editing equations, specifying endogenous and exogenous variables, and defining shocks. The **misc** folder includes modules for checking the syntax of model files, ensuring the correctness of equations, variables, and parameters.

The **model** folder contains modules that define the model class blueprint and facilitate model object creation.

The **numeric** folder comprises modules that solve equations and implement filters such as the Kalman filter, LRX filter, and band-pass filter.

The **preprocessor** folder contains modules that compile equation functions and compute partial derivatives of the first order (Jacobian), the second order (Hessian), and the third order. While most macroeconomic models require only the first-order derivatives, nonlinear models that employ rational expectations may necessitate higher-order derivatives.

The **tests** folder includes modules that can be used to run different economic models. Lastly, the **utils** folder contains modules that can read and parse model files from IRIS, Dynare, and TROLL software.

Examples of model files in YAML, IRIS, and DYNARE formats are located in the **models** folder.

V.   MODEL FILE AND RULES

Here, we describe the specifications of a model file written in YAML format. YAML is a human-readable data serialization language. The following is a brief list of rules that must be followed when creating a model file:
- Use whitespace identification to denote structure.
- Tab characters are not allowed as indentation.
- List members are denoted by a leading hyphen (-), with one member per line, or enclosed in square brackets and separated by a comma and space.
- Associative arrays are represented using the colon space (:) in the form variable: value.
- Use the hash symbol for comments.

This example illustrates the correct and incorrect tab rules for indentation in YAML:
- Correct (using spaces)

```
        calibration:
            param1: 0.1
            param2: 0.2
    • Incorrect (using tabs):
        calibration:
            [TAB] param1: 0.1
            [TAB] param2: 0.2
```

The model file must include the following sections: ***name, symbols, equations***, and ***calibration***.
- The ***name*** section describes the economic model.
- The ***symbols*** section consists of subsections for ***variables, shocks***, and ***parameters***. The ***variables*** subsection lists the names of the endogenous variables, the ***shocks*** section lists the names of the shocks, and the ***parameters*** subsection lists the names of parameters and exogenous variables.
- The ***equations*** section lists the model equations, while the ***calibration*** section specifies the starting values of endogenous variables, parameters, and exogenous variables.
- Additionally, the model file can contain an ***options*** section, which can be used to specify the simulation time range, the value and timing of shocks, and parameters of a multivariate normal distribution for shocks.

Below is an example of a model file:

name: Monetary policy model example (for details, see https://link.springer.com/article/10.1057/imfsp.2008.11)

symbols:
  variables: [PDOT,RR,RS,Y]
  shocks: [e]
  parameters: [g,p_pdot1,p_pdot2,p_pdot3,p_rs1,p_y1,p_y2,p_y3]

equations:
  - PDOT = p_pdot1*PDOT(+1) + (1-p_pdot1)*PDOT(-1) + p_pdot2*(g^2/(g-Y) - g) + p_pdot3*(g^2/(g-Y(-1)) - g)
  - RR = RS - p_pdot1*PDOT(+1) - (1-p_pdot1)*PDOT(-1)
  - RS = p_rs1*PDOT + Y
  - Y = p_y1*Y(-1) - p_y2*RR - p_y3*RR(-1) + e

calibration:
  # parameters
  g: 0.049
  p_pdot1: 0.414
  p_pdot2: 0.196
  p_pdot3: 0.276
  p_rs1: 3.000
  p_y1: 0.304
  p_y2: 0.098
  p_y3: 0.315
  # initial values
  PDOT: 0.0
  RR: 0.0
  RS: 0.0
  Y: 0.0
  e: 0.0

options:
  T : 14
  periods: [2]

shock_values: [0.02]

The lead and lag variables enter these equations with positive and negative signed integers in parentheses. In this example, the simulation range is set to 14 periods, and the shock *e* to output *Y* occurs at period 2. To implement multiple shocks, a user can specify multiple periods and corresponding shock values. For example:

options:
  T : 14
  periods: [2,4]
  shock_values: [0.02,-0.01]

If the steady state is known, it can be specified in a model file by entering the steady state values under the **steady_state** section of the YAML file:

steady_state:
  PDOT: 0.0
  RR: 0.0
  RS: 0.0
  Y: 0.0

In many cases, the steady state is unknown and is determined as part of the model solution.

To run stochastic simulations, please specify the number of paths, the probability density function of random shocks, and the parameters of this distribution, such as the mean of the shocks and their covariances. An example of a Real Business Cycle (RBC) model is demonstrated below:

name: Simple Real Business Cycle Model

symbols:
  variables: [Y,C,K,r,A]
  shocks: [ea]
  parameters: [beta,delta,gamma,rho,a]

equations:
  *- 1/C = 1/C(1) * beta * (1 + r)*
  *- Y = C + K - (1-delta) * K(-1)*
  *- Y = K(-1)^gamma * A^(1-gamma)*
  *- gamma*Y(1)/K = r + delta*
  *- log(A) = rho*log(A(-1)) + (1-rho)*log(a) + ea*

calibration:
    # parameters
    beta : 0.99
    gamma : 0.50
    delta : 0.03
    rho : 0.80
    a : 0.1
    # initial values
    C : 0.8
    K : 15.0
    Y : 1.2
    r : 0.01
    A : a

std : 0.05

options:
  T : 101
  Npaths : 10
  distribution: !MvNormal
    mean: [-0.05]
    cov: [[std^2]]

The calibration section specifies the values of parameters and the starting values of endogenous variables. Parameters of non-linear models can vary over time. For example, the parameter *delta* can be represented as a vector: [0.03, 0.03, 0.01, 0.02]. This indicates that *delta* equals 0.03 during periods 1 and 2, 0.01 during period 3, and 0.02 thereafter.

The framework implements simple macro language concepts such as **include** and **sets.** The example below illustrates how a model file can reference other YAML files, making it more compact and easier to read. Python parses these model files, inserts the content of the referred files, and generates a master model file.

symbols:
  # Endogenous variables
  variables: [@include endog_vars.yaml]
  # Exogenous variables
  shocks : [@include exog_vars.yaml, ex_y, vartheta, ex_sick]
  parameters : [@include params.yaml]

 # Parameter values
calibration:
  @include calibration.yaml

 # Model equations
equations:
  @include model_eqs.yaml

 # Variables labels
labels:
  @include labels.yaml

options:
  frequency: 0 # yearly

Another example illustrates the use of sets in modeling a three-country economy, specifically the United States (US), the European Union (EU), and Japan. The outputs of these countries are aggregated with specified weights to compute the world output.

name: Three countries economy

symbols:
  variables: [Y_WORLD,PDOT_WORLD]
  sets:
    countries: [US,EU,JP]
  shocks: [e]

equations:
  *- Y_WORLD = 0.5*Y_US+0.4*Y_EU+0.1*Y_JP*
  *- PDOT_WORLD = 0.5*PDOT_US+0.4*PDOT_EU+0.1*PDOT_JP*
  - Inflation:

7

set: countries
index: c
endo: PDOT_{c}
eq: PDOT_{c} = p_pdot1*PDOT_{c}(+1) + (1-p_pdot1)*PDOT_{c}(-1) + p_pdot2*(g^2/(g-Y_{c}) - g) + p_pdot3*(g^2/(g-Y_{c}(-1)) - g)
eq_ss: PDOT_{c} = 0
  - Real_Interest_Rate:
    set: countries
    index: c
    endo: RR_{c}
    eq: RR_{c} = RS_{c} - p_pdot1*PDOT_{c}(+1) - (1-p_pdot1)*PDOT_{c}(-1)
    eq_ss: RR_{c} = 0
  - Short_Term_Interest_Rate:
    set: countries
    index: c
    endo: RS_{c}
    eq: RS_{c} = p_rs1*PDOT_{c} + Y_{c}
    eq_ss: RS_{c} = 0
  - Output_Gap:
    set: countries
    index: c
    endo: Y_{c}
    eq: Y_{c} = p_y1*Y_{c}(-1) - p_y2*RR_{c} - p_y3*RR_{c}(-1) + e
    eq_ss: Y_{c} = 0

parameters:
  parameters: [g,p_pdot1,p_pdot2,p_pdot3,p_rs1,p_y1,p_y2,p_y3]
  #file: [params.yaml]

calibration:
  # shocks
  e: 0.0
  # parameters and initial values
  file: [files/calibration.yaml,files/initial_values.yaml]

options:
  T : 14
  periods: [2]
  shock_values: [0.02]

The framework parses this template file and generates equations for each of the three countries.

name: "Three countries economy"

Non-Linear Model

Transition Equations:
----

1   0.000 : Y_WORLD = 0.5*Y_US+0.4*Y_EU+0.1*Y_JP
2   0.000 : PDOT_WORLD = 0.5*PDOT_US+0.4*PDOT_EU+0.1*PDOT_JP
3   0.000 : Y_US = p_y1*Y_US(-1) - p_y2*RR_US - p_y3*RR_US(-1) + e
4   0.000 : RS_US = p_rs1*PDOT_US + Y_US
5   0.000 : RR_US = RS_US - p_pdot1*PDOT_US(+1) - (1-p_pdot1)*PDOT_US(-1)

8

6 0.000 : PDOT_US = p_pdot1*PDOT_US(+1) + (1-p_pdot1)*PDOT_US(-1) + p_pdot2*(g**2/(g-Y_US) - g) + p_pdot3*(g**2/(g-Y_US(-1)) - g)

7 0.000 : Y_EU = p_y1*Y_EU(-1) - p_y2*RR_EU - p_y3*RR_EU(-1) + e

8 0.000 : RS_EU = p_rs1*PDOT_EU + Y_EU

9 0.000 : RR_EU = RS_EU - p_pdot1*PDOT_EU(+1) - (1-p_pdot1)*PDOT_EU(-1)

10 0.000 : PDOT_EU = p_pdot1*PDOT_EU(+1) + (1-p_pdot1)*PDOT_EU(-1) + p_pdot2*(g**2/(g-Y_EU) - g) + p_pdot3*(g**2/(g-Y_EU(-1)) - g)

11 0.000 : Y_JP = p_y1*Y_JP(-1) - p_y2*RR_JP - p_y3*RR_JP(-1) + e

12 0.000 : RS_JP = p_rs1*PDOT_JP + Y_JP

13 0.000 : RR_JP = RS_JP - p_pdot1*PDOT_JP(+1) - (1-p_pdot1)*PDOT_JP(-1)

14 0.000 : PDOT_JP = p_pdot1*PDOT_JP(+1) + (1-p_pdot1)*PDOT_JP(-1) + p_pdot2*(g**2/(g-Y_JP) - g) + p_pdot3*(g**2/(g-Y_JP(-1)) - g)

This model printout displays a listing of equations, each prepended with its residuals. These residuals are derived by plugging the initial conditions of endogenous variables into the equations. A value of zero indicates that the initial condition corresponds to the steady state of this model.

Users can create model files for both dynamic and static steady-state equations. The example below illustrates a simplified version of a model file that includes only dynamic equations for five regions:

name: Five regions economy

symbols:
  sets:
    countries c: [US,EU,JP,EA,RC]
  variables: [Y_WORLD,PDOT(c),RR(c),RS(c),Y(c)]
  shocks: [e]
  parameters: [g,p_pdot1,p_pdot2,p_pdot3,p_rs1,p_y1,p_y2,p_y3]

equations:

  # World output
  - Y_WORLD = 0.18*Y_US+0.14*Y_EU+0.05*Y_JP+0.4*Y_EA+0.16*Y_RC
  # PDOT(c): Inflation
  - PDOT(c) = p_pdot1*PDOT(c)(+1) + (1-p_pdot1)*PDOT(c)(-1) + p_pdot2*(g^2/(g-Y(c)) - g) + p_pdot3*(g^2/(g-Y(c)(-1)) - g)
  # RR(c): Real Interest Rate
  - RR(c) = RS(c) - p_pdot1*PDOT(c)(+1) - (1-p_pdot1)*PDOT(c)(-1)
  # RS(c): Short Term Interest Rate
  - RS(c) = p_rs1*PDOT(c) + Y(c)
  # Y(c): Output Gap
  - Y(c) = p_y1*Y(c)(-1) - p_y2*RR(c) - p_y3*RR(c)(-1) + e

parameters:
  parameters: [g,p_pdot1,p_pdot2,p_pdot3,p_rs1,p_y1,p_y2,p_y3]
  #file: [params.yaml]

calibration:
  # shocks
  e: 0.0
  # parameters and initial values
  file: [files/calib.yaml]

options:

9

T : 14
periods: [2]
shock_values: [0.02]

This setup expands the model equations for the five regions, detailing the relationships among their outputs. The expanded equations demonstrate how the world output is influenced by the outputs of each region, thereby reflecting the interconnected dynamics of the model.

name: "Five regions economy"

Non-Linear Model

Transition Equations:
----

1   0.000 : Y_WORLD = 0.18*Y_US+0.14*Y_EU+0.05*Y_JP+0.4*Y_EA+0.16*Y_RC
   # Y US: Output Gap
2   0.000 : Y_US=p_y1*Y_US(-1)-p_y2*RR_US-p_y3*RR_US(-1)+e
   # RS US: Short Term Interest Rate
3   -3.000 : RS_US=p_rs1*PDOT_US+Y_US
   # RR US: Real Interest Rate
4   1.000 : RR_US=RS_US-p_pdot1*PDOT_US(+1)-(1-p_pdot1)*PDOT_US(-1)
   # PDOT US: Inflation
5   0.000 : PDOT_US=p_pdot1*PDOT_US(+1)+(1-p_pdot1)*PDOT_US(-1)+p_pdot2*(g**2/(g-Y_US)-g)+p_pdot3*(g**2/(g-Y_US(-1))-g)
   # Y EU: Output Gap
6   0.000 : Y_EU=p_y1*Y_EU(-1)-p_y2*RR_EU-p_y3*RR_EU(-1)+e
   # RS EU: Short Term Interest Rate
7   -0.600 : RS_EU=p_rs1*PDOT_EU+Y_EU
   # RR EU: Real Interest Rate
8   0.200 : RR_EU=RS_EU-p_pdot1*PDOT_EU(+1)-(1-p_pdot1)*PDOT_EU(-1)
   # PDOT EU: Inflation
9   0.000   :   PDOT_EU=p_pdot1*PDOT_EU(+1)+(1-p_pdot1)*PDOT_EU(-1)+p_pdot2*(g**2/(g-Y_EU)-g)+p_pdot3*(g**2/(g-Y_EU(-1))-g)
   # Y JP: Output Gap
10   0.000 : Y_JP=p_y1*Y_JP(-1)-p_y2*RR_JP-p_y3*RR_JP(-1)+e
   # RS JP: Short Term Interest Rate
11   0.000 : RS_JP=p_rs1*PDOT_JP+Y_JP
   # RR JP: Real Interest Rate
12   0.000 : RR_JP=RS_JP-p_pdot1*PDOT_JP(+1)-(1-p_pdot1)*PDOT_JP(-1)
   # PDOT JP: Inflation
13   0.000 : PDOT_JP=p_pdot1*PDOT_JP(+1)+(1-p_pdot1)*PDOT_JP(-1)+p_pdot2*(g**2/(g-Y_JP)-g)+p_pdot3*(g**2/(g-Y_JP(-1))-g)
   # Y EA: Output Gap
14   0.000 : Y_EA=p_y1*Y_EA(-1)-p_y2*RR_EA-p_y3*RR_EA(-1)+e
   # RS EA: Short Term Interest Rate
15   -3.000 : RS_EA=p_rs1*PDOT_EA+Y_EA
   # RR EA: Real Interest Rate
16   1.000 : RR_EA=RS_EA-p_pdot1*PDOT_EA(+1)-(1-p_pdot1)*PDOT_EA(-1)
   # PDOT EA: Inflation
17   0.000 : PDOT_EA=p_pdot1*PDOT_EA(+1)+(1-p_pdot1)*PDOT_EA(-1)+p_pdot2*(g**2/(g-Y_EA)-g)+p_pdot3*(g**2/(g-Y_EA(-1))-g)
   # Y RC: Output Gap

18   0.000 : Y_RC=p_y1*Y_RC(-1)-p_y2*RR_RC-p_y3*RR_RC(-1)+e
   # RS RC: Short Term Interest Rate
19   -9.000 : RS_RC=p_rs1*PDOT_RC+Y_RC
   # RR RC: Real Interest Rate
20   3.000 : RR_RC=RS_RC-p_pdot1*PDOT_RC(+1)-(1-p_pdot1)*PDOT_RC(-1)
   # PDOT RC: Inflation
21   0.000:PDOT_RC=p_pdot1*PDOT_RC(+1)+(1-p_pdot1)*PDOT_RC(-1)+p_pdot2*(g**2/(g-Y_RC)-g)+p_pdot3*(g**2/(g-Y_RC(-1))-g)

The model file structure is quite generic. In certain cases, users may seek a solution to a minimization or maximization problem of an objective function, given linear or non-linear constraints. The example below illustrates a transportation expenses minimization model file. Here is a description of this problem:

Two plants, located in San Diego and Seattle, deliver goods to three markets in Chicago, New York, and Topeka. The supply side of the factories is limited by specific upper bounds *a(i),* while the demand side amounts *b(j)* for the markets are limited from below. The distances from the factories to the markets *d(i)(j)* are provided in the calibration section, along with the cost of transportation *cost(i)(j)* per mile. The objective function is defined as the sum of costs multiplied by shipment quantities.

name: Transportation expenses minimization model

sets:
  plants i: [Seattle, SanDiego]
  markets j: [NewYork, Chicago, Topeka]

symbols:
  variables: [x(i)(j)]
  parameters: [f, a(i), b(j), d(i)(j), cost(i)(j)]

equations:
  - Supply(i): sum(j, x(i)(j))
  - Demand(j): sum(i, x(i)(j))

calibration:
  f: 90
  a(i): [350, 600]
  b(j): [325, 300, 275]
  x(i)(j): [[0,0,0],[0,0,0]]
  d(i)(j): [[2.5, 1.7, 1.8],
        [2.5, 1.8, 1.4]]
  *cost(i)(j): f*d(i)(j)/1000 # Transport cost in 1000s of dollars per case*

objective_function:
  *- sum(i;j, cost(i)(j)*x(i)(j)) # Total shipment cost*

constraints:
  - Supply(i) .lt. a(i)
  - Supply(i) .ge. 0
  - Demand(j) .gt. b(j)
  - x(i)(j)   .ge. 0

11

labels:
   x: Shipment quantities in cases
   f: Freight in dollars per case per thousand miles

Solver: 'SLSQP'   # 'SLSQP' #'trust-constr'
Method: 'Minimize'

This setup generates the following output:

Number of declared equations: 2, variables: 1, constraints: 4
Number of expanded equations: 5, parameters: 18

Model:
------
name: "Transportation expenses maximization model"

Linear Model

Transition Equations:

 Supply_Seattle : (x_Seattle_NewYork+x_Seattle_Chicago+x_Seattle_Topeka)
 Supply_SanDiego : (x_SanDiego_NewYork+x_SanDiego_Chicago+x_SanDiego_Topeka)
 Demand_NewYork : (x_Seattle_NewYork+x_SanDiego_NewYork)
 Demand_Chicago : (x_Seattle_Chicago+x_SanDiego_Chicago)
 Demand_Topeka : (x_Seattle_Topeka+x_SanDiego_Topeka)

Objective Function:
func   =
(cost_Seattle_NewYork*x_Seattle_NewYork+cost_SanDiego_NewYork*x_SanDiego_NewYork+cost_Seattle_Chicago*x_Seattle_Chi
cago+cost_SanDiego_Chicago*x_SanDiego_Chicago+cost_Seattle_Topeka*x_Seattle_Topeka+cost_SanDiego_Topeka*x_SanDiego
_Topeka)

Constraints:
   Supply_Seattle < a_Seattle
   Supply_SanDiego < a_SanDiego
   Supply_Seattle >= 0
   Supply_SanDiego >= 0
   Demand_NewYork > b_NewYork
   Demand_Chicago > b_Chicago
   Demand_Topeka > b_Topeka
   x_Seattle_NewYork >= 0
   x_Seattle_Chicago >= 0
   x_Seattle_Topeka >= 0
   x_SanDiego_NewYork >= 0
   x_SanDiego_Chicago >= 0
   x_SanDiego_Topeka >= 0

Objective function value: 1.54e+02

Solution status: success
Number of function calls: 161
Elapsed time: 0.01 (seconds)

Var Name	Var Value	Var Name	Var Value
x_Seattle_NewYork	50	x_Seattle_Chicago	300
x_Seattle_Topeka	0	x_SanDiego_NewYork	275
x_SanDiego_Chicago	0	x_SanDiego_Topeka	275

Table.1. Shipment amounts from factories to markets. The locations of the factories and markets are indicated in the "Var Name" column.

The example below presents a model file for Armington trade equilibrium with iceberg costs. This type of model falls under the umbrella of Computable General Equilibrium (CGE) models. CGE models are typically solved using commercial software such as GAMS, and GEMPACK.

name: Armington Trade Equilibrium with Iceberg Costs

sets:
  regions r: [R1, R2, R3]
  goods j: [G1]
  regions alias s: r

symbols:
  variables: [Q(j)(r),P(j)(r),c(j)(r),Y(j)(r)]
  parameters: [sig,eta,mu,Q0(j)(r),P0(j)(r),Y0(j)(r),c0(j)(r),tau(j)(r)(s),vx0(j)(r)(s),zeta(j)(r)(s)]

equations:
 # Eq.1 Aggregate demand
 *- DEM(j)(r):   Q(j)(r) - Q0(j)(r) * (P0(j)(r) / P(j)(r))**eta*

 # Eq.2 Armington unit cost function
 - ARM(j)(s):   sum(r, zeta(j)(r)(s)**sig*tau(j)(r)(s)*c(j)(r))**(1/(1-sig)) - P(j)(s)

 # Eq.3 Market clearance
 - MKT(j)(r):   Y(j)(r) = sum(s, tau(j)(r)(s)*Q(j)(s)*zeta(j)(r)(s)*P(j)(s)/(tau(j)(r)(s)*c(j)(r)))**sig

 # Eq.4 Input supply (output)
 *- SUP(j)(r):   Y0(j)(r) * (c(j)(r)/c0(j)(r)) ** mu - Y(j)(r)*

calibration:
  # Parameters
  sig   : 3
  eta   : 1.5
  mu   : 0.5
  P0(j)(r)   : 1
  c0(j)(r)   : 1
  vx0(j)(r)(s) : 1
  vx0(j)(r)(r) : 3
  Q0(j)(r)   : sum(s, vx0(j)(s)(r)) / P0(j)(r)
  Y0(j)(r)   : sum(s, vx0(j)(s)(r)) / c0(j)(r)
  # Variables

Q(j)(r)   : Q0(j)(r)
Q(G1)(R3)   : 7.0
P(j)(r)   : P0(j)(r)
P(G1)(R2)   : 1.5
P(G1)(R3)   : 0.5
c(j)(r)   : c0(j)(r)
c(G1)(R2)   : 0.5
Y(j)(r)   : Y0(j)(r)
Y(G1)(R1)   : 3.5

constraints:
  # Positive Variables
  - Q(j)(r) .ge. 5.1
  - P(j)(r) .ge. 0
  - c(j)(r) .ge. 0
  - Y(j)(r) .ge. 0
  # Positive LHS of equations
  - DEM(j)(r) .ge. 0
  - ARM(j)(s) .ge. 0
  - MKT(j)(r) .ge. 0
  - SUP(j)(r) .ge. 0

labels:
  # Variables
  Q: Composite Quantity
  P: Composite Price Index
  c: Composite input price (marginal cost)
  Y: Composite input supply (output)
  # Parameters
  sig: Elasticity of substitution
  eta: Demand elasticity
  mu: Supply elasticity
  Q0:   Benchmark aggregate quantity
  P0:   Benchmark price index
  c0:   Benchmark input cost
  Y0:   Benchmark input supply
  tau: Iceberg transport cost factor
  vx0: Arbitrary benchmark export values
  zeta: Bilateral preference weights
  # Equations
  DEM: Aggregate demand
  ARM: Armington unit cost function
  MKT: Input market clearance
  SUP: Input supply (output)

Model: [DEM.Q,ARM.P,MKT.c,SUP.Y]
Solver: 'CONSTRAINED_OPTIMIZATION'   # 'MCP', 'ROOT'

The platform parses this model file and generates equations for each region and goods in the set:

Number of declared equations: 4, variables: 4, constraints: 4

14

Number of expanded equations: 12, parameters: 42

Model:
------
name: "Armington Trade Equilibrium with Iceberg Costs"
file: "c:\work\platform\examples\models\OPT\armington.yaml"

Non-Linear Model

Transition Equations:

DEM_G1_R1 : Q_G1_R1-Q0_G1_R1*(P0_G1_R1/P_G1_R1)**eta
DEM_G1_R2 : Q_G1_R2-Q0_G1_R2*(P0_G1_R2/P_G1_R2)**eta
DEM_G1_R3 : Q_G1_R3-Q0_G1_R3*(P0_G1_R3/P_G1_R3)**eta
MKT_G1_R1   *:*
Y_G1_R1=(tau_G1_R1_R1*Q_G1_R1*zeta_G1_R1_R1*P_G1_R1/(tau_G1_R1_R1*c_G1_R1)+tau_G1_R1_R2*Q_G1_R2*zeta_G1_R1_R2*P_G1_R2/(tau_G1_R1_R2*c_G1_R1)+tau_G1_R1_R3*Q_G1_R3*zeta_G1_R1_R3*P_G1_R3/(tau_G1_R1_R3*c_G1_R1))
**sig
MKT_G1_R2   *:*
Y_G1_R2=(tau_G1_R2_R1*Q_G1_R1*zeta_G1_R2_R1*P_G1_R1/(tau_G1_R2_R1*c_G1_R2)+tau_G1_R2_R2*Q_G1_R2*zeta_G1_R2_R2*P_G1_R2/(tau_G1_R2_R2*c_G1_R2)+tau_G1_R2_R3*Q_G1_R3*zeta_G1_R2_R3*P_G1_R3/(tau_G1_R2_R3*c_G1_R2))
**sig
MKT_G1_R3   *:*
Y_G1_R3=(tau_G1_R3_R1*Q_G1_R1*zeta_G1_R3_R1*P_G1_R1/(tau_G1_R3_R1*c_G1_R3)+tau_G1_R3_R2*Q_G1_R2*zeta_G1_R3_R2*P_G1_R2/(tau_G1_R3_R2*c_G1_R3)+tau_G1_R3_R3*Q_G1_R3*zeta_G1_R3_R3*P_G1_R3/(tau_G1_R3_R3*c_G1_R3))
**sig
SUP_G1_R1 : Y0_G1_R1*(c_G1_R1/c0_G1_R1)**mu-Y_G1_R1
SUP_G1_R2 : Y0_G1_R2*(c_G1_R2/c0_G1_R2)**mu-Y_G1_R2
SUP_G1_R3 : Y0_G1_R3*(c_G1_R3/c0_G1_R3)**mu-Y_G1_R3
ARM_G1_R1   *:*
(zeta_G1_R1_R1**sig*tau_G1_R1_R1*c_G1_R1+zeta_G1_R2_R1**sig*tau_G1_R2_R1*c_G1_R2+zeta_G1_R3_R1**sig*tau_G1_R3_R1*c_G1_R3)**(1/(1-sig))-P_G1_R1
ARM_G1_R2   *:*
(zeta_G1_R1_R2**sig*tau_G1_R1_R2*c_G1_R1+zeta_G1_R2_R2**sig*tau_G1_R2_R2*c_G1_R2+zeta_G1_R3_R2**sig*tau_G1_R3_R2*c_G1_R3)**(1/(1-sig))-P_G1_R2
ARM_G1_R3   *:*
(zeta_G1_R1_R3**sig*tau_G1_R1_R3*c_G1_R1+zeta_G1_R2_R3**sig*tau_G1_R2_R3*c_G1_R2+zeta_G1_R3_R3**sig*tau_G1_R3_R3*c_G1_R3)**(1/(1-sig))-P_G1_R3

Constraints:
Q_G1_R1 >= 5.1
Q_G1_R2 >= 5.1
Q_G1_R3 >= 5.1
P_G1_R1 >= 0
P_G1_R2 >= 0
P_G1_R3 >= 0
c_G1_R1 >= 0
c_G1_R2 >= 0
c_G1_R3 >= 0
Y_G1_R1 >= 0

15

Y_G1_R2 >= 0
Y_G1_R3 >= 0

CONSTRAINED_OPTIMIZATION solver
Solution status: success
Number of function calls: 39
Elapsed time: 0.05 (seconds)

This setup produces the results shown in the table below:

Var Name	Var Value	Var Name	Var Value
Q_G1_R1	5.4	Q_G1_R2	5.4
Q_G1_R3	7.1	P_G1_R1	1.1
P_G1_R2	1.3	P_G1_R3	0.7
c_G1_R1	1.1	c_G1_R2	0.7
c_G1_R3	0.9	Y_G1_R1	3.5
Y_G1_R2	3	Y_G1_R3	5

Table.2. Equilibrium values of the composite quantity index, price index, and output of the Armington trade model with iceberg costs.

VI. COMPUTING FUNCTIONS AND JACOBIANS

Once the YAML file is read, the program caches the equations into memory. An abstract syntax tree (AST) of these mathematical expressions is built using the parse method from the Python AST package. These AST expressions are then converted into symbolic expressions with the help of the Sympify method. The Sympy package is utilized for symbolic mathematics in Python. To find the partial derivatives of the symbolic expressions of functions, the diff method is employed. The result of these steps is the generation of a function in Python. The parameters of these functions are vectors of endogenous variables, model parameters, and the order of function differentiation. The output of these functions consists of symbolic expressions for equations and their partial derivatives, up to the third order.

Below, we provide examples of the Python code that is automatically generated for linear equations:

Linear Example:

$y1 = y2$
$y1 + y2 = x1$
$x1 = 2$

Jacobian:

1	-1
1	1

Generated Function:

```
def f_dynamic(x, p, order=1):

    import numpy
```

```
y1__ = x[0]
y2__ = x[1]
x1 = p[0]

# Function
function= numpy.zeros(2)
function [0] = y1__ - y2__
function [1] = -x1 + y1__ + y2__

if order == 0:
    return function

# Jacobian
jacobian= numpy.zeros((2,2))
jacobian [0,0] = 1
jacobian [0,1] = -1
jacobian [1,0] = 1
jacobian [1,1] = 1

if order == 1:
    return [function, jacobian]
```

VII. MODEL SETTINGS

The platform utilizes several numerical algorithms to solve model equations. The first two in the table below are solvers for non-linear models, while the last four are designed for linear models. LBJ solver is named after Laffargue, Boucekkine, and Juillard, who developed this forward-backward substitution algorithm, which is applied to solve systems of equations with dense matrices. The ABLR solver addresses systems of stacked equations using sparse matrix algebra. The next two algorithms employ Generalized Schur matrix decomposition and mimic the algorithms used by DYNARE and IRIS software.

For more details on the solvers, please refer to the accompanying documentation file titled "Numerical Algorithms."

Solver Algorithms	Description
LBJ	Juillard, Laxton, McAdam, Pioro algorithm (mimics DYNARE Toolbox perfect foresight solver)
ABLR	Armstrong, Black, Laxton, Rose algorithm
Villemot	Villemot Sebastien (mimics DYNARE Toolbox perturbations method solver)
Klein	Paul Klein (mimics IRIS Toolbox perturbations method solver)
BinderPesaran	Binder and Pesaran algorithm
AndersonMoore	Anderson and Moore algorithm

The default boundary condition is a ***non-reflective*** condition, indicating that the first derivative of the variables remains constant at the right boundary of the computational domain. This contrasts with a fixed boundary condition, which can lead to disturbances in the solution at the right boundary.

Boundary Values Condition	Description
FixedBoundaryCondition	Fixed condition at right boundary
ZeroDerivativeBoundaryCondition	Floating zero derivative condition at right boundary

The following Kalman filter and smoother algorithms have been implemented:

Kalman Filter Algorithms	Description
Diffuse	Diffuse Kalman filter (multivariate and univariate) with missing observations
Durbin_Koopman	Non-diffusive variant of Durbin-Koopman Kalman filter
Non_Diffuse_Filter	Non diffuse Kalman filter (multivariate and univariate) with missing observations
Unscented	Unscented Kalman filter
Particle	Particle filter

Kalman smoother algorithms:

Kalman Smoother Algorithms	Description
BrysonFrazier	Bryson, Frazier
Diffuse	Diffuse Kalman Smoother (multivariate and univariate) with missing observations
Durbin_Koopman	Non-diffuse variant of Kalman smoother (multivariate)

The Kalman filter requires the setting of initial conditions for the filtered variables and their error variance-covariance matrix. The two tables below present the coded algorithms:

Variables	Description
StartingValues	Model starting values are used
SteadyState	Steady-state values are used as starting values
History	Starting values are read from a history file

These are the algorithms used for setting the starting values of the Kalman filter error covariance matrix:

Error Covariance Matrix	Description
Diffuse	Diffuse prior for covariance matrices (Pinf and Pstar)
StartingValues	Starting values for covariance matrices (Pinf with diagonal values of 1.E6 on diagonal and Pstar=T*Q*T')
Equilibrium	Equilibrium error covariance matrices obtained by discrete Lyapunov solver by using stable part of transition and shock matrices
Asymptotic	Asymptotic values for error covariance matrices; it is obtained by solving time discrete Riccati equation

Finally, when estimating and sampling model parameters, users can select from several Markov Chain Monte Carlo (MCMC) sampling algorithms:

Sampling Algorithms	Description
Emcee	Affine Invariant Markov Chain Monte Carlo Ensemble sampler
Pymcmcstat	Adaptive Metropolis based sampling techniques include
Pymcmcstat_mh	Metropolis-Hastings (MH): Primary sampling method
Pymcmcstat_am	Adaptive-Metropolis (AM): Adapts covariance matrix at specified intervals.

Pymcmcstat_dr	Delayed-Rejection (DR): Delays rejection by sampling from a narrower distribution. Capable of n-stage delayed rejection.
Pymcmcstat_dram	Delayed Rejection Adaptive Metropolis (DRAM): DR + AM
Pymc3	Markov Chain Monte Carlo (MCMC) and variational inference (VI) algorithms
Particle_pmmh	Particle Marginal Metropolis Hastings sampling
Particle_smc	Particle Sequential Quasi
Particle_gibbs	Particle Generic Gibbs sampling

This environment offers a rich modeling setting for users to experiment with.

VIII. GRAPHICAL REPRESENTATION OF MODEL EQUATIONS

By raising the flag of the *graph_info* parameter in the run function of the **driver** module, a directional graph of model variables is produced. An example of this graph for the US potential output model is shown below. The green ellipses represent endogenous variable nodes that appear on the left side of the model equations, while the yellow nodes indicate the variables that appear on the right side. The arrows illustrate the dependence of the left-side variables on their counterparts in the equation expressions. The equations of the MVF potential output model and the generated graph are displayed below:

equations:
  # Transition equations
  #Eq.1 Potential output definition
  - LGDP = LGDP_BAR + Y
  #Eq.2 Stochastic process for potential output level
  - LGDP_BAR = LGDP_BAR(-1) + DLGDP + RES_LGDP_BAR
  #Eq.3 Stochastic process for growth rate of potential
  - DLGDP = (1-theta)*DLGDP(-1) + theta*growth_ss + RES_DLGDP
  #Eq.4 Stochastic process for output gap
  - Y = phi*Y(-1) + RES_Y
  #Eq.5 Philips curve
  - PIE = lmbda*PIE(+1) + (1-lmbda)*PIE(-1) + beta*Y + RES_PIE
  #Eq.6 Growth definition
  - GROWTH = LGDP - LGDP(-1)
  #Eq.7 Potential growth definition
  - GROWTH_BAR = LGDP_BAR - LGDP_BAR(-1)
  #Eq.8 NAIRU definition
  - UNR_BAR = UNR + UNR_GAP
  #Eq.9 Dynamic Okun's law
  - UNR_GAP = tau2*UNR_GAP(-1) + tau1*Y + RES_UNR_GAP
  #Eq.10 Stochastic process for NAIRU
  - UNR_BAR = (1-tau4)*UNR_BAR(-1) + G_UNR_BAR + tau4*unr_ss + RES_UNR_BAR
  #Eq.11 Stochastic process for the change in NAIRU
  - G_UNR_BAR = (1-tau3)*G_UNR_BAR(-1) + RES_G_UNR_BAR
  #Eq.12 One-year ahead model consistent inflation expectations (reporting variable)
  - PIE_BAR = PIE(+1)
  #Eq.13 One-year ahead model consistent growth expectations (reporting variable)
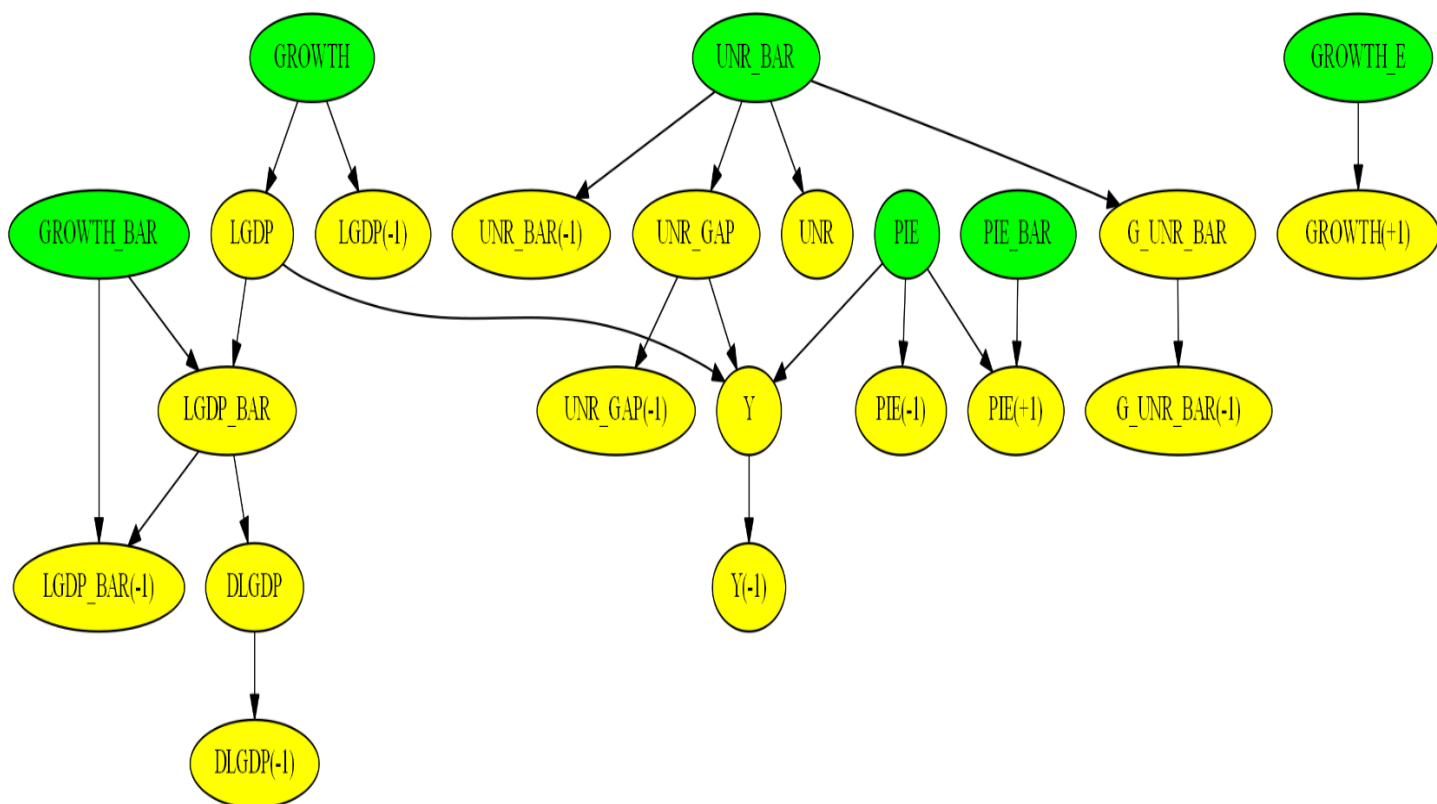 *- GROWTH_E = GROWTH(+1)*

Fig. 3. Graphical representation of the dependencies among model variables.

IX. PDF REPORTING

By raising the flag of the *model_info* parameter in the **run** function, a PDF report of the model object is generated. This PDF document includes several sections that describe the model's endogenous and exogenous variables, parameters, shocks, as well as transient and measurement equations, as illustrated below:

<div align="center">

Model File

Generated by Python Framework

June 13, 2024

</div>

1 Model Information

name: *Multivariate Filter of Potential Output for US Economy*
file: *c:\work\framework\models\TOY\MVF_US.yaml*

1.1 Endogenous Variables Values

DLGDP = 5.0, GROWTH = 5.0, GROWTH_BAR = 5.0, GROWTH_E = 5.0, G_UNR_BAR = 0.0, LGDP = 800.0, LGDP_BAR = 800.0, PIE = 7.0, PIE_BAR = 7.0, UNR = 7.2, UNR_BAR = 7.2, UNR_GAP = 0.0, Y = 0.0

1.2 Measurement Variables

OBS_GROWTH, OBS_LGDP, OBS_PIE, OBS_UNR

1.3 Parameters

beta = 0.25, growth_ss = 0.00, lmbda = 0.25, phi = 0.75, tau1 = 0.30, tau2 = 0.30, tau3 = 0.10, tau4 = 0.10, theta = 0.10, unr_ss = 5.00

1.4 Shocks

RES_DLGDP, RES_G_UNR_BAR, RES_LGDP_BAR, RES_PIE, RES_UNR_BAR, RES_UNR_GAP, RES_Y

1.5 Measurement Shocks

RES_OBS_LGDP, RES_OBS_PIE, RES_OBS_GROWTH, RES_OBS_UNR

1.6 Equations

1 : LGDP = LGDP_BAR + Y

2 : LGDP_BAR = LGDP_BAR(-1) + DLGDP + RES_LGDP_BAR

<div align="center">

1

</div>

X. PROGRAMMING WITH FRAMEWORK

Creating a Model Object

A model can be instantiated in two ways:

1. Passing a path to a model file in the *importModel()* method:

<div align="center">

</div>

from driver import importModel
model=importModel(path_to_model_file, Solver="Benes")

The parameters for this function include:
- Path to the model file
- Path to a file containing historical data
- List of exogenous variables
- Path to a file containing shock values
- Path to a file containing steady-state values
- Path to a calibration file

2. Passing a list of endogenous variables names, equations, parameters, etc., to the *getModel*() method:

from model.importModel import getModel
model=getModel(name=name,eqs=eqs,meas_eqs=measEqs,variables=variables,parameters=params,shocks=shoc
ks,meas_variables=measVar,calibration=calibration,var_labels=var_labels,options=options, infos=infos)

The parameters for this function include:
- Model name
- List of transition equations
- List of endogenous variables
- List of model parameters
- List of shock names
- Calibration dictionary with starting values of endogenous variables and parameters

Importing Model Files

The framework can read, and parse model files developed by macroeconomic modeling software such as DYNARE, IRIS, TROLL, and SIRIUS. Below, we present an example of a YAML model file that incorporates DYNARE *.mod files, describing endogenous and exogenous variables, parameters, and equations:

name: DIGNAD model

symbols:
  # Endogenous variables
  variables: [@include end_vars.mod]
  # Exogenous variables
  shocks : [@include exo_vars.mod, shock_s,tfp_adj]

  # Parameters
  parameters : [@include params.mod]

 # Model equations
equations: @include model_eqs.mod

options:
  T: 200
  frequency: 0 # yearly

In this example, the YAML model file outlines the structure for the model, including the definitions of variables, parameters, and the equations that govern their relationships.

Upon parsing this model file, the framework generates a Python model object. This translation is a work in progress and may require further development to accommodate more complex model files.

Setting Starting Values

Starting values of endogenous variables can be specified in the calibration section of the model's YAML file. They can also be read from a file containing historical data. The framework reads these variable values at the start of the simulation range. If the endogenous variables have lags and leads, the timing of these lags and leads is determined, and the corresponding values are utilized. The historical data will overwrite the starting values specified in the model file.

However, the data file may not contain historical values for some variables. In such cases, the missing variable values can be estimated by solving the model's steady-state equations and minimizing their residuals. This is accomplished by raising the flag of the parameter *bTreatMissingObs.*

The excerpt of the code below illustrates an example of setting the starting values of variables:

from model.model import setStartingValues
setStartingValues(model=model, hist=path_to_data_history_file, bTreatMissingObs=True)

Setting Parameters

Parameters can be set in four ways:

- Defining parameters in the calibration section of the yaml model file

- Passing a dictionary of parameters names and values when creating the model object. For example,
  model = import_model(model_file_path, calibration={ "b1": 0.7, ...})

- Calling setCalibration() and setParameters() functions of model object. For example, *model.setCalibration("b1",0.7)* and *model.setParameters({ "b1":[1,2,3]}).* The latter function is used when passing time-dependent parameters.

- Passing the path of a file with defined values of parameters. These files can be in YAML, text or excel format. For example, *model = importModel(model_file_path,shocks_file_path=shocks_file_path, steady_state_file_path=steady_state_file_path, calibration_file_path=calibration_file_path).*

  Below is the content of the fcalibration.txt file, which defines parameters that could be missing in the JLMP98 model file:

  g = 0.049
  p_pdot1 = 0.414
  p_pdot2 = 0.196
  p_pdot3 = 0.276
  p_rs1 = 3.000
  p_y1 = 0.304
  p_y2 = 0.098
  p_y3 = 0.315

Defining Shocks

Shocks can be set by:

1. Passing dictionary object to *setShocks()* method where shocks names are keys of this dictionary and values are either the list of tuples of shock time and shock values or python Pandas' time series. The example below sets shock SHK_DLA_CPIE to values of 1, 5, and 3. Please note that index 0 corresponds to the starting time of simulations.

Shocks can be set in the following way:

1. Passing a dictionary object to the *setShocks()* method, where the shock names are the keys of this dictionary, and the values are either a list of tuples containing shock times and shock values, or Python Pandas time series. The example below sets the SHK_DLA_CPIE variable shock to values of 1, 5, and 3. Please note that index 0 corresponds to the starting time of the simulations.

   import pandas as pd
   from model.util import setShocks

   #d = {"SHK_DLA_CPIE": [(0,1),(1,5),(2,3)]}
   d = {"SHK_DLA_CPIE": pd.Series([1,5,3],pd.date_range(start=start_date,end=end_date,freq='QS'))}
   setShocks(model,d)

2. Defining a list of shocks**.** In the example below, four shocks occur in the first quarter of 1998. The shock values are set to 1. Variables are shocked sequentially, one by one, in a loop, and the impulse-response functions are computed:

   model.options["periods"] = [[1998,1,1]]
   shock_names = model.symbols["shocks"]
   shocks = [1,1,1,1]
   num_shocks = len(list_shocks)
   for i in range(num_shocks):
   *   shock_name = list_shocks[i]*
   *   ind = shock_names.index(shock_name)*
   *   shock_values = np.zeros(n_shocks)*
   *   shock_values[ind] = shocks[i]*
   *   model.options["shock_values"] = shock_values*

   *   # Find solution*
   *   y,rng_date,epsilonhat,etahat,s,rng,periods,model = run(model=model,irf=True)*

   In this example, each shock is defined to occur at the specified time, and the impulse-response functions are calculated accordingly.

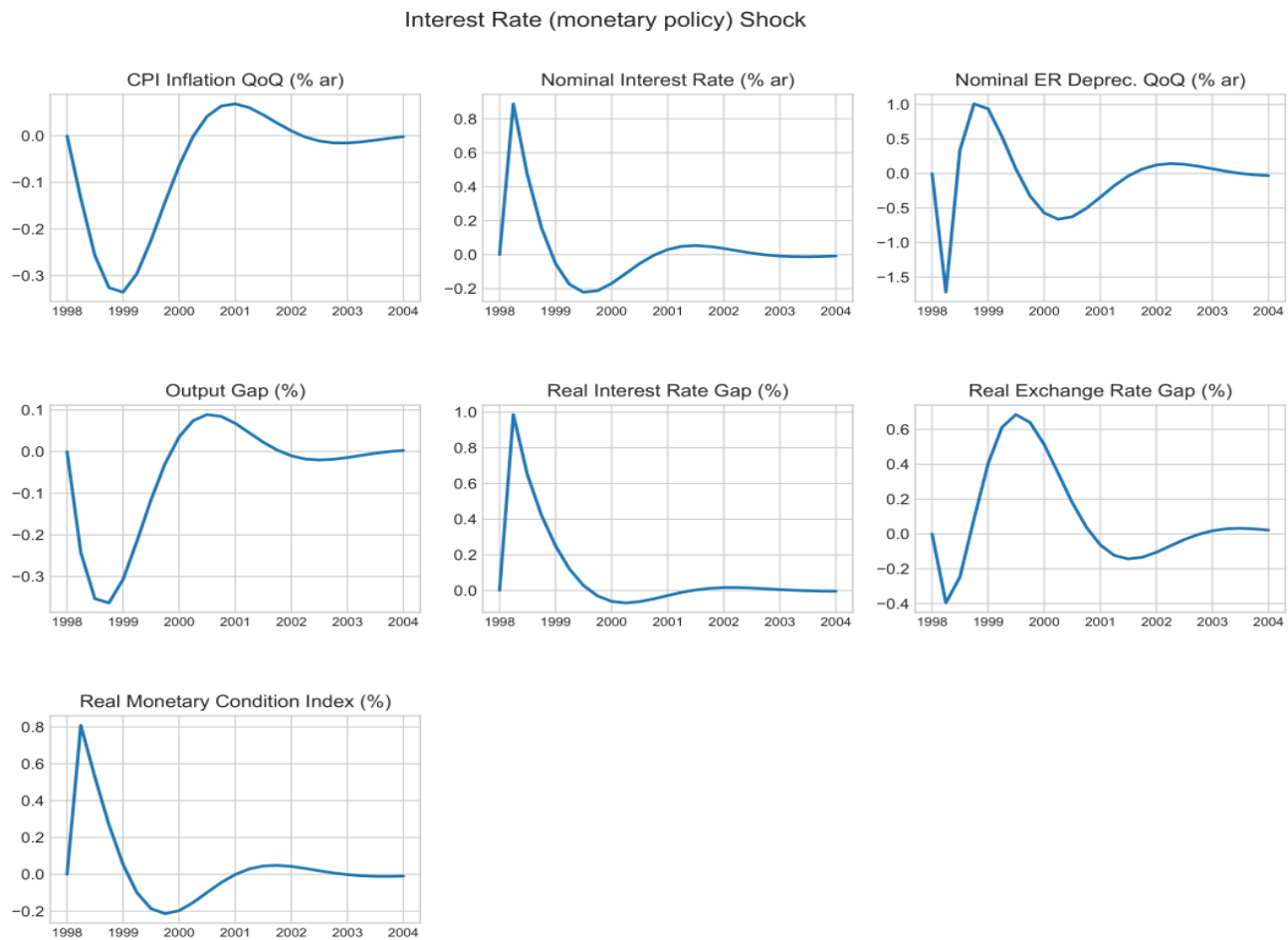   Below, we present the plots of the Impulse Response Functions.

Fig. 4. Impulse response functions of the GAP model.

XI. RUNNING SIMULATIONS

The **driver** module is the workhorse of the framework. It serves multiple purposes, including stochastic calculations and forecasts, sampling of parameters, as well as Kalman filtering and smoothing.

y,rng_date = run(model=model, irf=True)

This function returns forecast dates and forecast results.

The parameters of this function are:
- *Model* object: The instance of the model being used.
- Variable *Solver*: The name of the numerical method used to solve the system of equations.
- List of output variable names: Specifies which variables to plot graphs for.
- Variable *shocks_file_path:* The path to the shock file.
- Variable *steady_state_file_path: The* path to the steady-state file.
- Variable *calibration_file_path:* The path to the calibration file or files.
- Boolean flag *Plot:* If raised, this will plot graphs.
- Boolean flag *Output:* If raised, this will output simulation results to an Excel file.
- String *output_dir:* The path to the output folder where results will be saved.

If *output_dir* parameter is not set, the platform will query the user-defined environment variable *PLATFORM_OUTPUT_FOLDER***.** This variable defines a path to output, for example: C:/temp/out. If neither *output_dir* nor this environment variable is set, the output directory will default to the *Framework* folder.

Additionally, a call to the Kalman filter function will return the filtered and smoothed shocks:

y,rng_date, epsilonhat,etahat = kalman_filter(model=model)

The parameters of this function are:
- Variable *meas***:** The path to a file containing measurement data.
- Variable *Prior***:** Used to set the initial values of the error covariance matrix for the Kalman Filter.
- Variable *Filter***:** The name of the Kalman filter algorithm.
- Variable Smoother**:** The name of the Kalman smoother algorithm.

Finally, the estimate(model=model) function estimates the model parameters and runs MCMC sampling.

Test Program

The **Tests** folder contains modules with examples for running several models. Each module requires the user to provide a path to a model file. Users can also specify which output variables they wish to output or plot. If the *output_variables* parameter is not set, all variables will be output or plotted by default.

Results will be stored in Excel files or in a Python SQLite database within the **data** folder, while plots will be saved in the **graphs** folder. Below, we present the test.py file.

```
from driver import run

fname = 'models/TOY/JLMP98.yaml'   # Simple monetary policy example
fout = 'data/test.csv' # Results are saved in this file
decomp = ['PDOT','RR','RS','Y'] # List of variables for which decomposition plots are produced
output_variables = None

# Path to model file
file_path = os.path.abspath(os.path.join(working_dir, '..', fname))

# Function that runs simulations, model parameters estimation, MCMC sampling, etc...
y,rng_date = run(fname=file_path,fout=fout,decomp_variables=decomp,
    output_variables=output_variables,
    Output=True,Plot=True,Solver="LBJ",
    #InitCondition="SteadyState",
    graph_info=False,use_cache=False,Sparse=False)
```

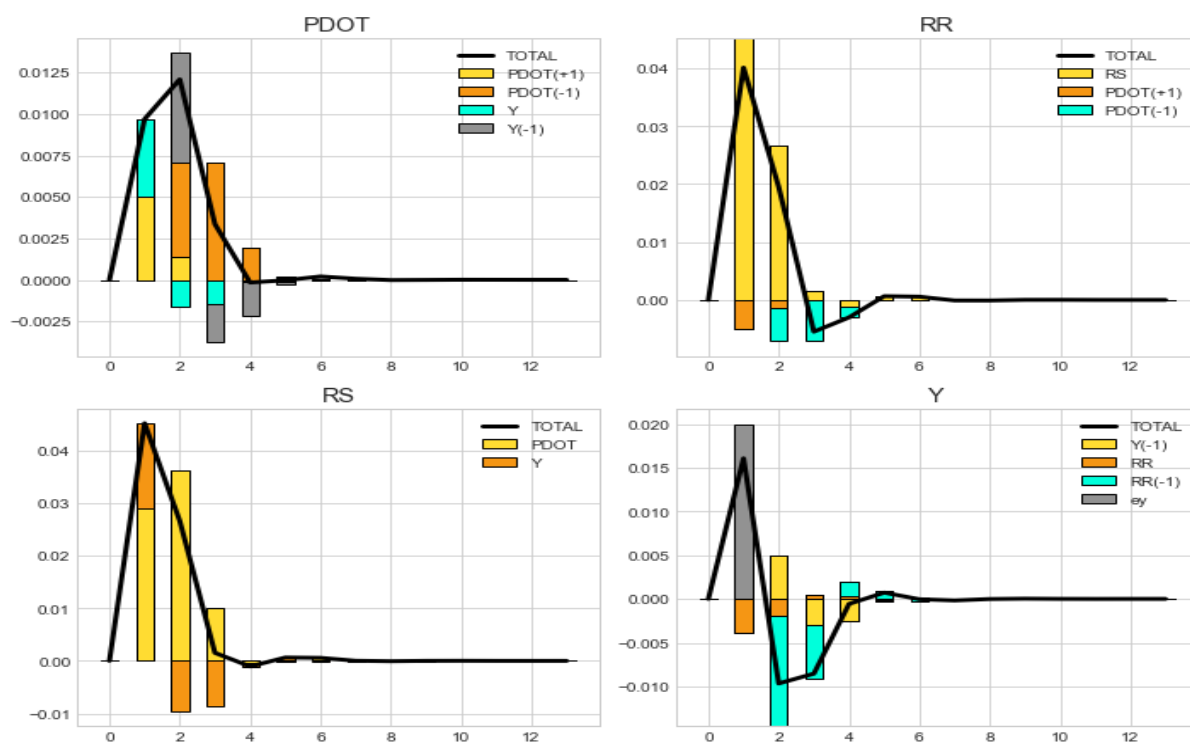Running this script generates the graphs shown below.

Fig. 5. The Impulse Response Functions of inflation (PDOT), real interest rates (RR), nominal interest rates (RS), and output (Y). The output variable is shocked by 2% in period 1.

Below, we present the model file and the results of the forecast for a simple Real Business Cycle (RBC) model. The shocks to output (Y) and total factor productivity (A) are stochastic and are described by a multivariate normal distribution. The mean and covariance matrix for these shocks are specified in the **options** section of this model file.

RBC model with stochastic shocks and 10 realization paths

name: Simple Real Business Cycle Model
symbols:
  variables: [Y,C,K,r,A]
  shocks: [ea,ey]
  parameters: [beta,delta,gamma,rho,a]

equations:
  *- 1/C = 1/C(1) * beta * (1 + r)*
  *- Y = C + K - (1-delta) * K(-1)*
  *- Y = K(-1)^gamma * A^(1-gamma) + ey*
  *- gamma*Y(1)/K = r + delta*
  *- log(A) = rho*log(A(-1)) + (1-rho)*log(a) + ea*

calibration:
    # parameters
    beta : 0.99
    cov : 0.0001
...

options:
  T : 51

Npaths : 10
distribution: !MvNormal
  mean: [-0.05,0.05]
  cov: [[std^2, cov],[cov, std^2]]

The run of this test program generates plots of variables shown below.
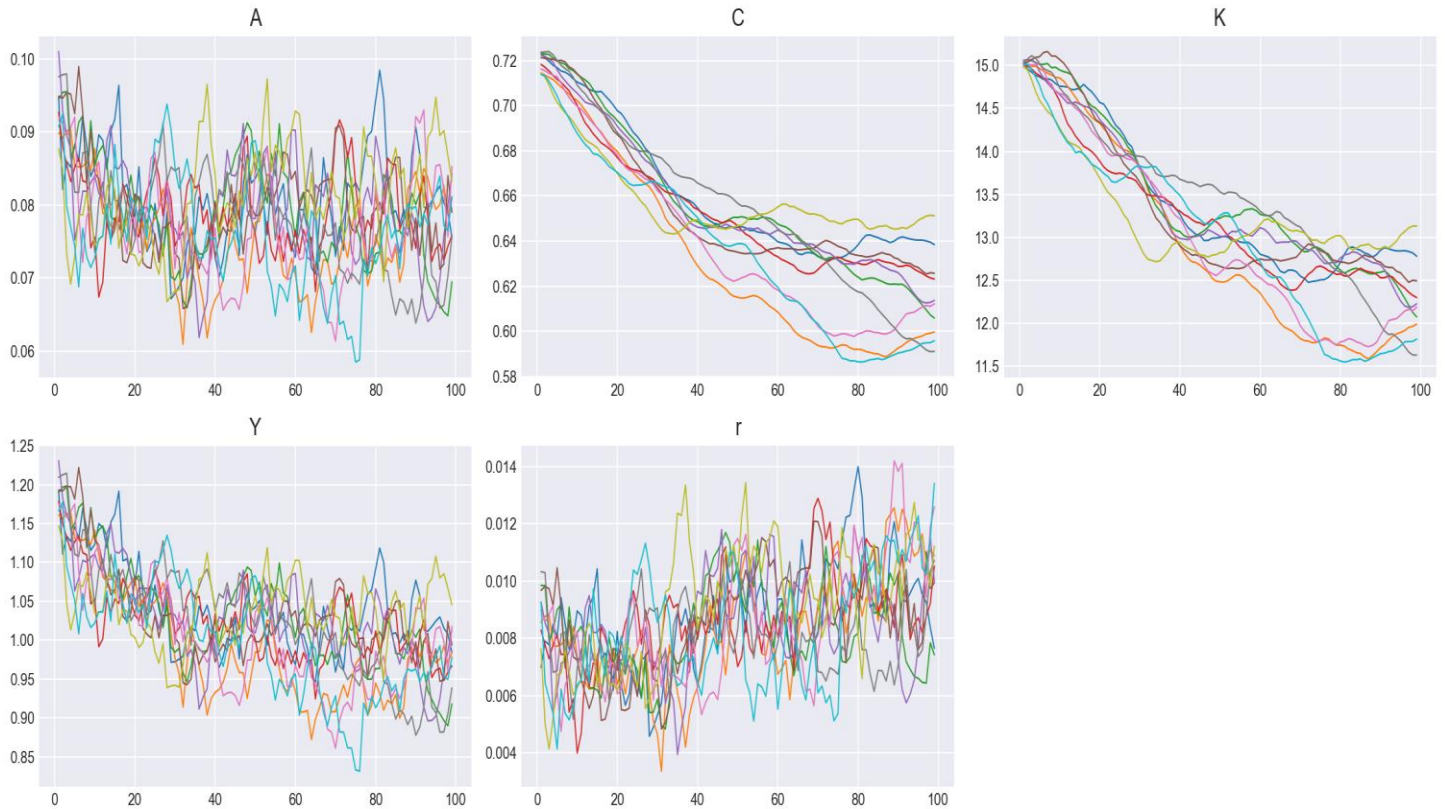
Fig. 6. Ten realization paths of macroeconomic variables. Here A is the total factor productivity, C is the consumption, K is the capital, Y is the output, and r is the interest rate.

Kalman Filter and Smoother

The next example illustrates the testing of the US potential output model. In this scenario, the user specifies paths to the model file, the measurement data, and the results output file. The *importModel* function parses the model file and constructs a model object, which is then passed as a parameter to the Kalman filter.

```
from driver import importModel
   from driver import kalman_filter
   fname = 'TOY/MVF_US.yaml' # Multivariate Kalman filter example
   fout = 'data/results.csv' # Results are saved in this file
   output_variables = None #['DLGDP','LGDP','LGDP_BAR','PIE','UNR','UNR_GAP']    # List of variables that will be plotted or
displayed
   decomp = None #['pie','r']
```

28

```
# Path to measurement data
meas = os.path.abspath(os.path.join(working_dir, '../data/dataForKalman.csv'))

# Path to model file
file_path = os.path.abspath(os.path.join(working_dir, '../models', fname))

# Instantiate model object
model = importModel(fname=file_path, Solver="Benes",
            Filter="Durbin_Koopman",Smoother="Durbin_Koopman",Prior="Equilibrium",
            measurement_file_path=meas,model_info=True)

### Run Kalman filter
y,rng_date,epsilonhat,etahat = kalman_filter(model=model,meas=meas,fout=fout,Output=False,Plot=True)
```

A call to the Kalman filter function returns a list of dates, *rng_date*, the results of the Kalman filter and smoother wrapped in a list **y**, as well as the filtered shocks, *epsilonhat* and smoothed shocks, *etahat.* The observation variables are read from the measurement data file. They can also be sourced from databases such as HAVER, ECOS, EDI, and the World Bank.

The example below demonstrates how to source GDP, CPI index, growth rate, and unemployment observation variables from the HAVER database. In this case, the name of the observation variable is followed by a ticker name and the operation to be performed on this time series:

Model file:

name: Multivariate Filter of Potential Output for US Economy

...
equations:
    # Transition equations
    #Eq.1 Potential output definition
    - LGDP = LGDP_BAR + Y
    ...

measurement_equations:
    - OBS_LGDP = LGDP + RES_OBS_LGDP
    - OBS_PIE = PIE + RES_OBS_PIE
    - OBS_GROWTH = GROWTH + RES_OBS_GROWTH
    - OBS_UNR = UNR + RES_OBS_UNR

calibration:
  # parameters:
  beta: 0.25
...
  # initial values and starting values for endogenous variables:
  LGDP: 800
...
  # Standard deviation of shocks:
  std_RES_LGDP_BAR: 0.1
...
  # Standard deviations of measurement variables:
  # Any standard deviation that is not listed below is treated as zero.
  std_RES_OBS_LGDP : 1.0
...

data_sources:
  # Frequencies : Annually,Quarterly,Monthly,Weekly,Daily
  frequency : 'AS'
  HAVER:
    OBS_LGDP : 'GDPA@USECON,log' # quarterly SAAR GDP, Bill. USD
    OBS_PIE : 'CTGA@USECON,difflog' # monthly core CPI
    OBS_GROWTH : 'GDPA@USECON,difflog'
    OBS_UNR : 'USRA@EMPLR' # unemployment rate
  # ECOS:
  #   OBS_LGDP : 'WEO_WEO_PUBLISHED@111_NGDP'
  # EDI:
  #   OBS_LGDP : '111_NGDP'
  # WORLD_BANK:
  #   OBS_LGDP : 'USA_NGDP,log'

The execution of this test produces the plots shown below. The solid blue lines represent the filtered endogenous variables, while the green and yellow lines illustrate the results of applying the LRX and HP filters, respectively. The dots indicate the actual data points. By default, the RX and HP filters are applied to measurement variables with names that end in "_BAR."
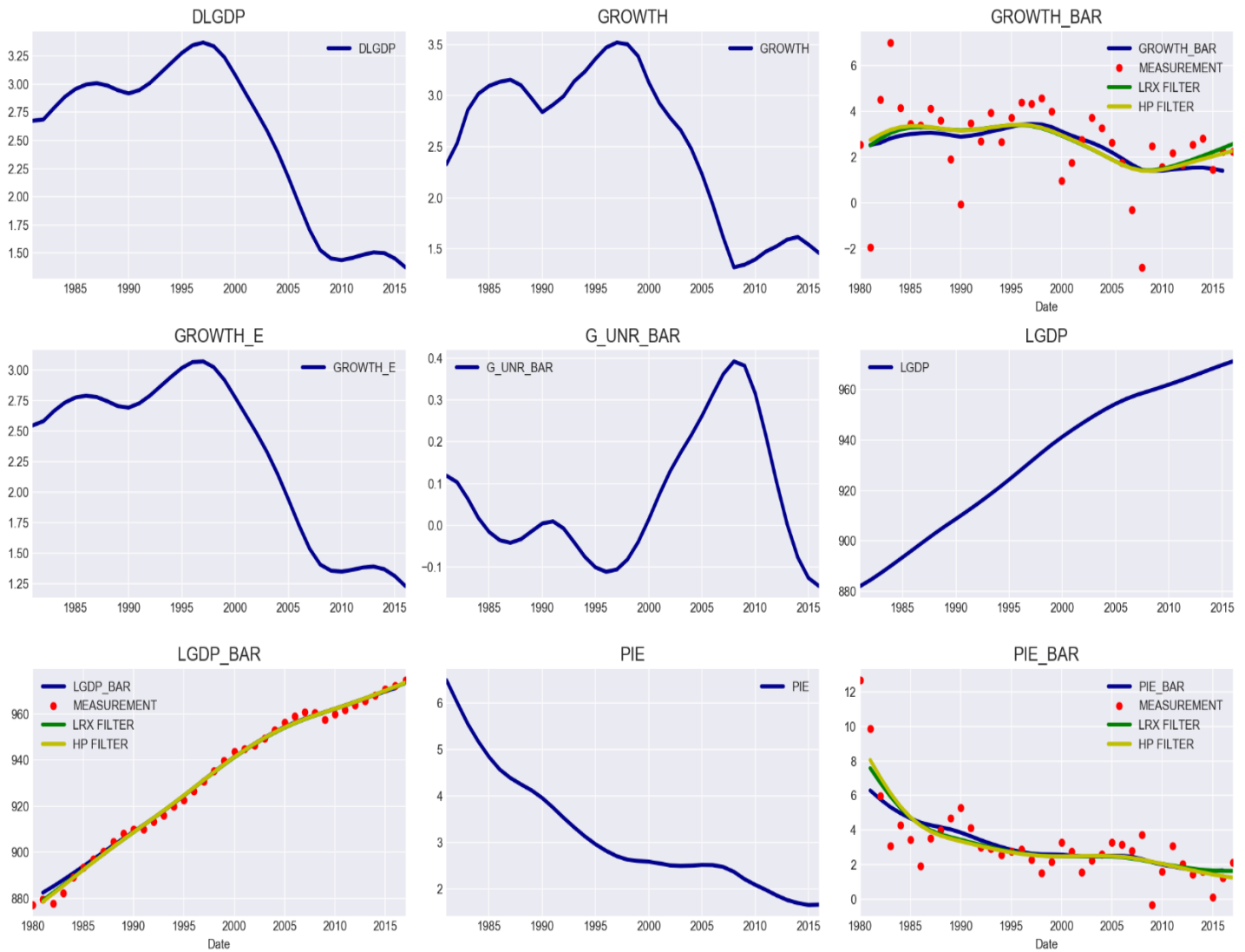
Fig. 7. The HP, LRX, and Kalman filters yield similar results for the potential output model.

30

Estimating Model Parameters

Users can estimate model parameters given measurement data. This is achieved by finding parameters that maximize the likelihood of the model fit. One can choose all or a subset of parameters by selecting initial values along with the lower and upper bounds of the model parameters. Below is an excerpt from a model file:

estimated_parameters:
   # Please choose one of the following distributions:
   # normal_pdf,lognormal_pdf,beta_pdf,gamma_pdf,t_pdf,weibull_pdf,inv_gamma_pdf,inv_weibull_pdf,
   # wishart_pdf,inv_wishart_pdf
   # PARAM NAME, INITVAL, LB, UB, PRIOR_SHAPE, PRIOR_P1, PRIOR_P2, PRIOR_P3, PRIOR_P4, PRIOR_P5
   # The first parameter is the parameter name, the second is the initial value, the third and
   # the fourth are the lower and the upper bounds, the fifth is the prior shape, and
   # the sixth to tenth are prior parameters (mean, standard deviation, shape, etc...).
   - beta, 0.25, 0, 10, normal_pdf, 0.25, 0.01
   - lmbda, 0.25, 0, 1., normal_pdf, 0.25, 0.01
   - phi,   0.75, 0, 1., normal_pdf, 0.75, 0.01
   - theta, 0.1, 0, 0.5, normal_pdf, 0.1, 0.01

This framework attempts to find optimal parameters when the *estimate* flag is raised in the estimate function. Sampling of parameters can be accomplished by passing the parameter *sample* equal to true. Markov Chain Monte Carlo (MCMC) methods are utilized to sample model parameters from probability distributions. The names of the four parameters listed above are followed by their starting values, lower and upper bounds, the type of probability density function distribution, and

the parameters' means and standard deviations. An example of parameter sampling for 300 draws is shown below:
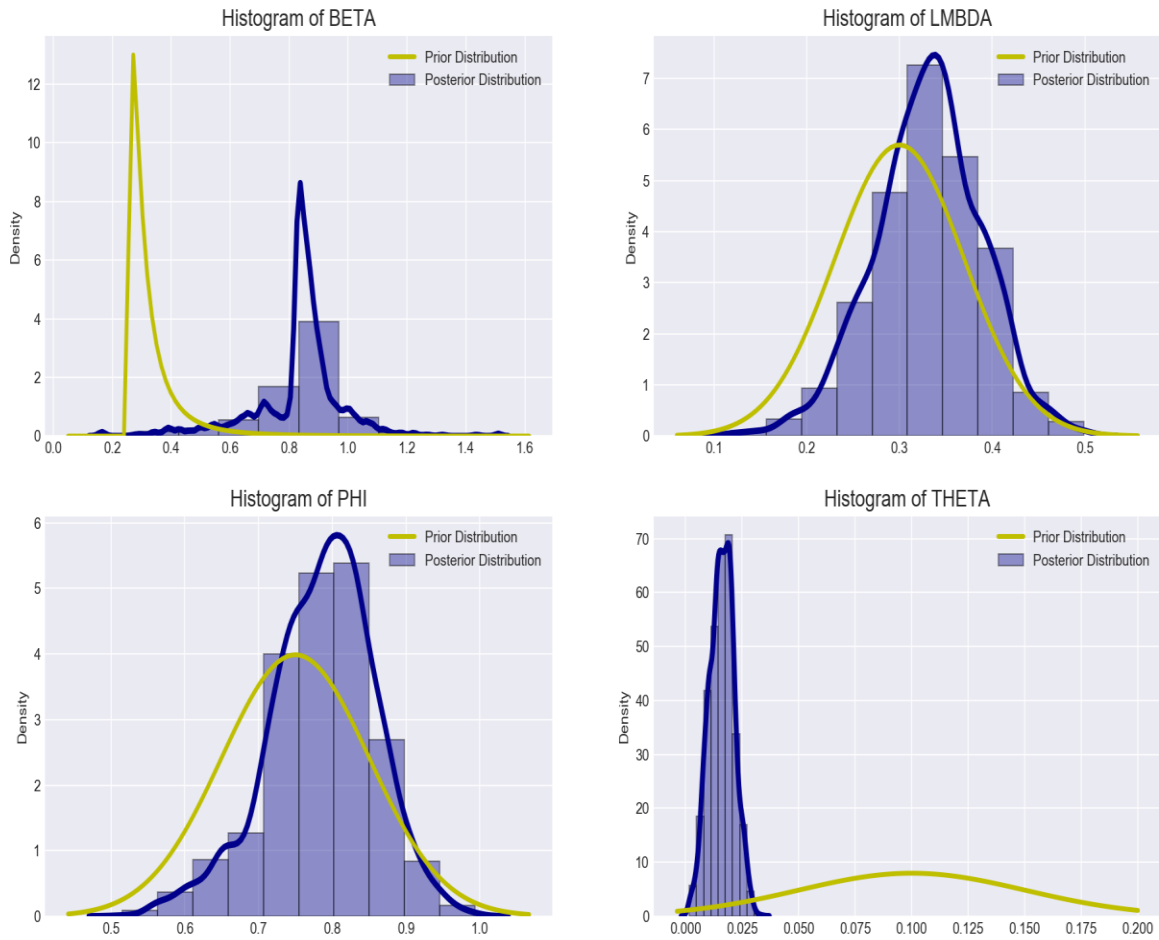

Fig. 8. The distribution of the US MVF potential output model is illustrated below. The yellow lines represent the prior distribution, while the blue lines indicate the posterior distribution.

These sampling algorithms utilize the sum of the prior and posterior logarithms of probabilities. The number of draws is controlled by the parameter *Ndraws* in the estimate function.

Additionally, two-dimensional projections of the multidimensional sample covariances are presented below. The blue vertical and horizontal lines display the means of the samples, while the red lines indicate the optimal values of the parameters.
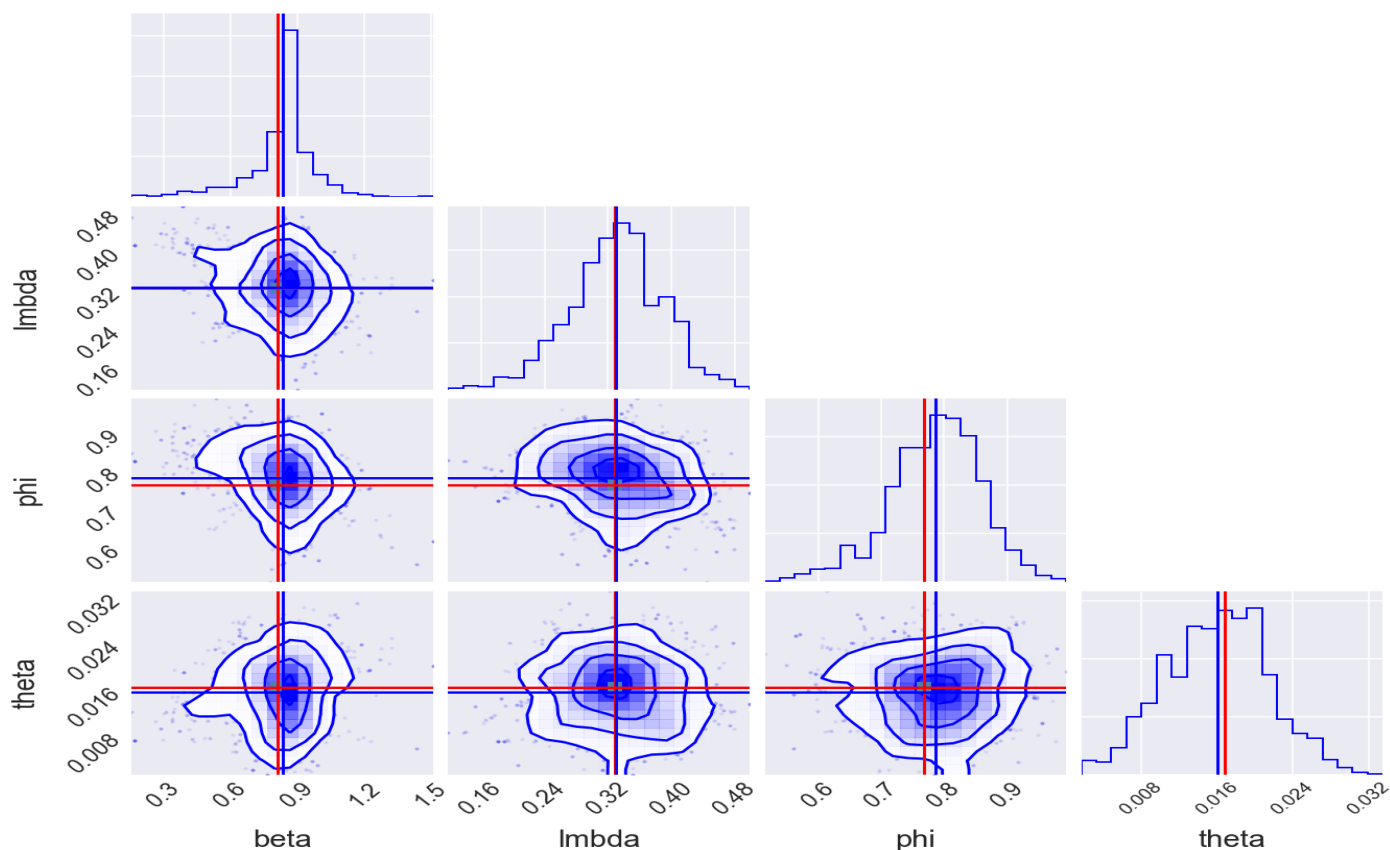
Fig. 9. Two dimensional projections of model parameters.

Judgmental Adjustments

Users may have a specific perspective on the future paths of endogenous variables. This can be programmed by "exogenizing" endogenous variables and "endogenizing" exogenous shock variables. The example below demonstrates judgmental adjustments to the nominal interest rate *RS*. The framework identifies the shock values *SHK_RS* that will adjust the path of *RS* to the desired level. This shock is endogenized by calling the model's swap method:

```
rng = pd.date_range(start=start_date, end=end_date, freq='QS')
m = {''RS': pd.Series([5,5],rng)}
shock_names  = ['SHK_RS']
model.swap(var1=m,var2=shock_names,reset=False)
```

Another example of users' *tunes* is illustrated below. This example demonstrates the imposition of *soft* and *hard* tunes on macroeconomic variables. Seven scenarios are included: no tunes, soft tunes, hard tunes, a combination of soft and hard tunes, anticipated tunes, and conditional tunes.

```
## Define the time range of the forecast
   start_fcast = '2017-1-1'
   end_fcast = '2020-12-31'
   start    = dt.strptime(start_fcast,"%Y-%m-%d")
   start_m1  = start - rd.relativedelta(months=1*3)
```

33

```
start_m3  = start - rd.relativedelta(months=3*3)
start_p1  = start + rd.relativedelta(months=1*3)
start_p2  = start + rd.relativedelta(months=2*3)
#start_p3  = start + rd.relativedelta(months=3*3)
start_p4  = start + rd.relativedelta(months=4*3)
start_p7  = start + rd.relativedelta(months=7*3)
end = dt.strptime(end_fcast,"%Y-%m-%d")


### 1. No tunes
y1,rng_date = run(model=model)


### 2. Soft tune
#d = {"SHK_DLA_CPIE": [(0,1),(1,2),(2,3)]}
d = {"SHK_DLA_CPIE": pd.Series([1,2,3],pd.date_range(start=start,end=start_p2,freq='QS'))}
model.setShocks(d)
y2,rng_date = run(model=model)


### 3. Hard tune
rng_3 = pd.date_range(start=start_p1,end=start_p2,freq='QS')
m = {}
m['RS'] = pd.Series([5,5],rng_3)
#m['RS'] = y2['RS'][rng_3]
shock_names  = ['SHK_RS']
model.swap(var1=m,var2=shock_names,reset=False)
y3,rng_date = run(model=model)


### 4. Combination of soft tune and hard tune
d = {"SHK_L_GDP_GAP": [(3,1)]}
#d = {"SHK_L_GDP_GAP": pd.Series([1],pd.date_range(start=start_p3,end=start_p3,freq='QS'))}
model.setShocks(d)
 rng_4 = pd.date_range(start=start,end=start_p2,freq='QS')
m['L_GDP_GAP'] = pd.Series([-1.0, -1.0, -1.0],rng_4)
 shock_names  = ['SHK_L_GDP_GAP']
model.swap(var1=m,var2=shock_names)
y4,rng_date = run(model=model)


### 5. Anticipated soft tune
d = {"SHK_L_S": [(3,10*i)]}
model.setShocks(d)
y5,rng_date = run(model=model)


### 6. Anticipated hard tune
rng_6 = pd.date_range(start=start_p2,end=start_p4,freq='QS')
m['DLA_CPIF'] = pd.Series([5*i,5*i,5*i],rng_6)
shock_names  = ['SHK_DLA_CPIF']
model.swap(var1=m,var2=shock_names)
y6,rng_date = run(model=model)


### 7. Conditional shock
rng_7 = pd.date_range(start=start,end=start_p4,freq='QS')
m = {}
m['DLA_GDP'] = pd.Series([10.0, 10.0, 10.0, 10.0, 10.0],rng_7)
model.condition(m)
```

34

y7,rng_date = run(model=model)

This setup produces the results of the seven scenarios, as shown below.
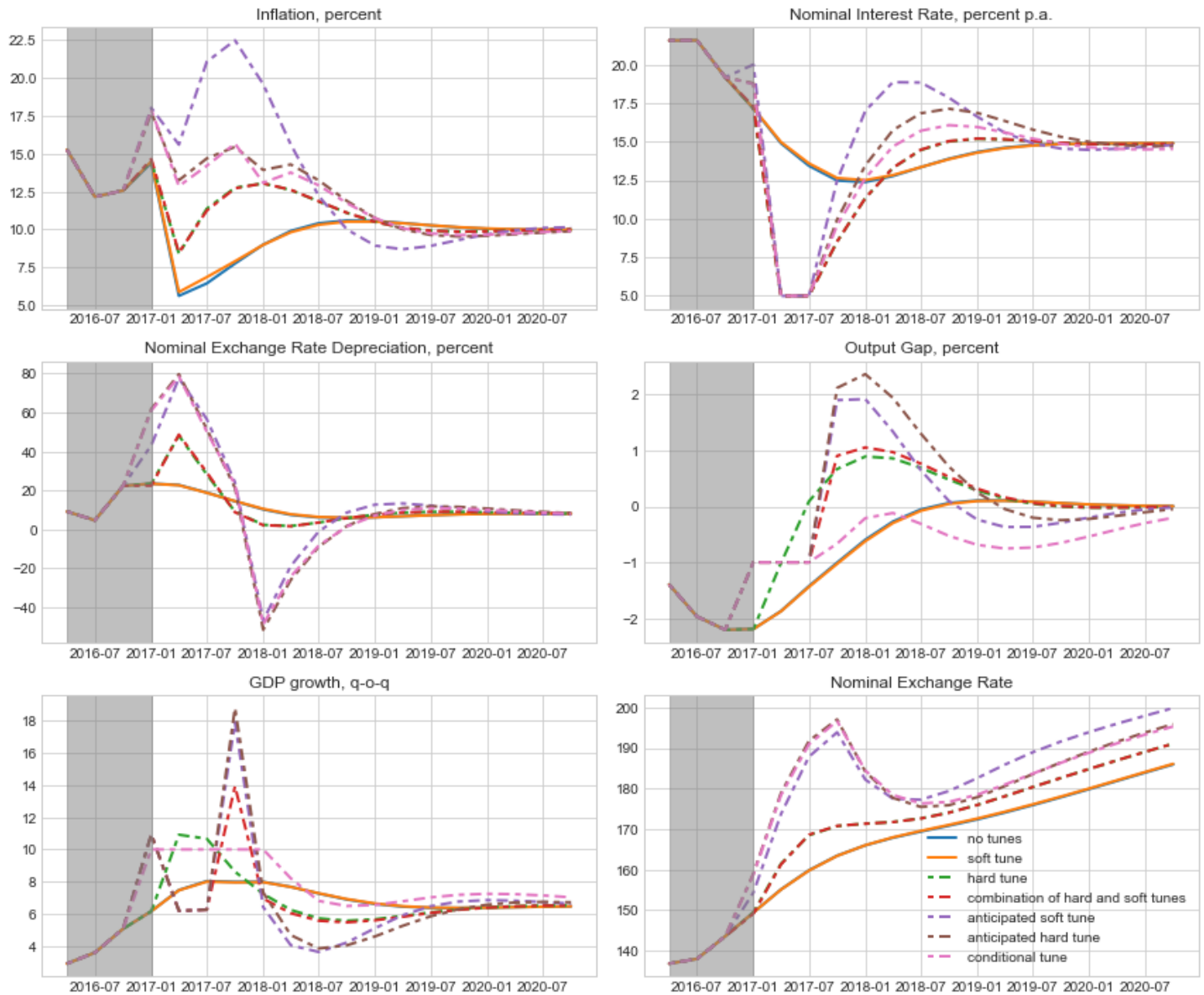
Forecast - Main Indicators

Fig. 10. Forecast of macroeconomic variables for different sets of user judgments on future path of these variables.

DSGE Modelling in Jupyter Notebook

The Python framework can be executed in a Jupyter notebook, which is a web-based computational environment. The example below illustrates a code snippet and the results of the forecast for economic variables of Chile using the Sirius XML model.

In [2]:

```
import os, sys
import numpy as np
from tkinter import filedialog as fd
from tkinter import Tk
# Open file dialog
root = Tk(); root.update(); root.withdraw(); model_file = fd.askopenfilename(); root.destroy()
print('Model file path: ',model_file)

path  = os.path.abspath(model_file+'\\..\\..\\..\\..\\')
working_dir = os.path.abspath(path+'\\src\\')
print('Working directory: ',working_dir)
```

Model file path: D:/Data/agoumilevski/Framework/examples/siriusModels/xml/chile_model.xml

Working directory: D:\Data\agoumilevski\Framework\src

Parse XML file and instantiate model. Enter time and shock values.

In [3]:

```
from utils.util import importModel

#shocks = { '2017/1/1': {'e_dot_cpi':1, 'e_lgdp_gap':-1} }

model = importModel(file_path=model_file,startDate='2015/1/1',endDate='2020/1/1',shocks=None)
print(model)
```
Model:

------

name: "Sirius Model"

file: "D:/Data/agoumilevski/Framework/examples/siriusModels/xml/chile_model.xml"

Linear Model

Equations:

----

 1 : 0.0000 : a1*lgdp_gap(-1)-a2*mci+a3*lx_gdp_gap+e_lgdp_gap-(lgdp_gap)

 2 : 0.0000 : a4*(rr_gap+cr_prem)+(1-a4)*(-lz_gap)-(mci)

 3 : 0.0000 : a5*cr_prem(-1)+(1-a5)*(prem-prem(-1))+e_cr_prem-(cr_prem)

 ...

Find steady-state solution

In [4]:

```
from numeric.solver.nonlinear_solver import find_steady_state

steady_state, eigen_values = find_steady_state(model)
eigen_values = np.array([e for e in eigen_values if abs(e) < 10])
```

Plot eigen values of system of equations

In [5]:

from driver import plot,plotDecomposition,plotEigenValues
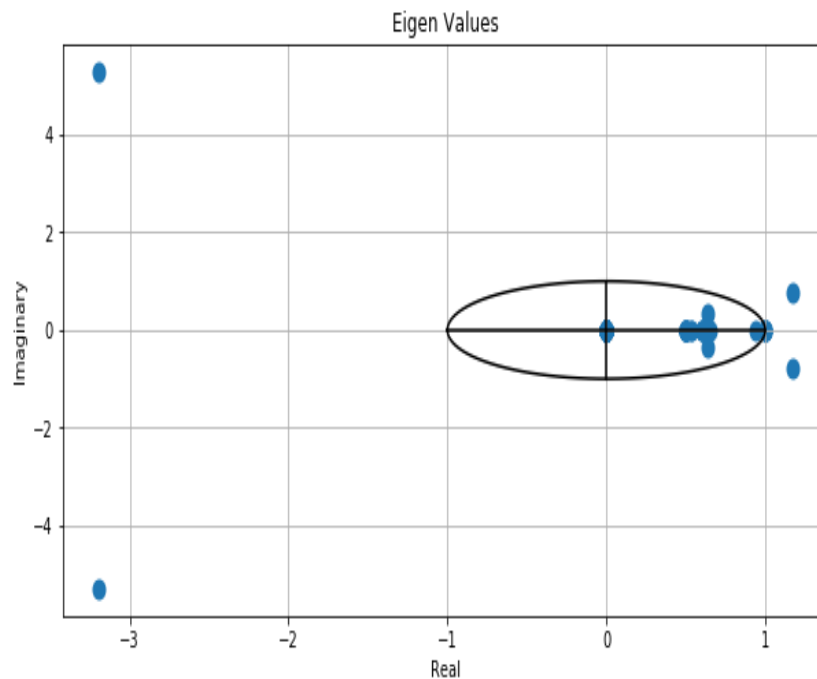
plotEigenValues(eigen_values,save=False)

Fig. 11. Eigen values of Chile model.

Run simulations. Plot graphs of endogenous variables and these variables decomposition.
In [6]:
from driver import run

output_variables = ['dot4_cpi','dot4_cpi_x','dot4_gdp','lgdp_gap','lx_gdp_gap','mci','rmc','rr','rr_gap']
time,data,variable_names,rng,periods,model = run(model=model,output_variables=output_variables,decomp_variables=None,Plot=True)

Finding a steady-state solution. It uses starting variable values.

Steady-State Solution:
['cr_prem=-0.014351544049', 'cum_gap=0.0', 'dot4_cpi=1.72829839923', ...]

Eigen Values:
[0.00000000e+00 +0.00000000e+00j -0.00000000e+00 -0.00000000e+00j
..
* 2.61188334e+01 +1.09534293e+08j]*

Number of eigen values greater than one: 14
Number of forward-looking endogenous variables: 4
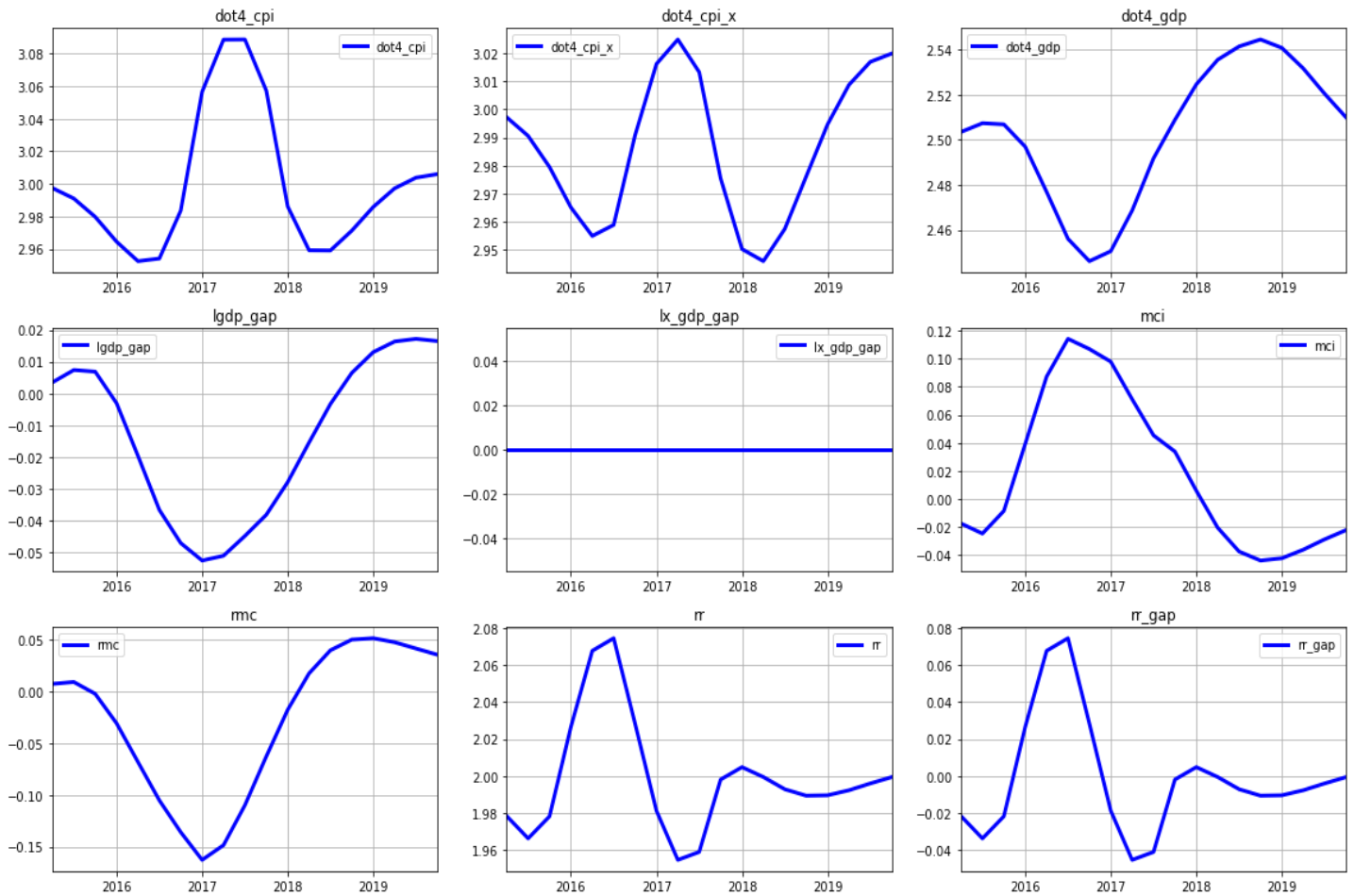
Plot Evolution of Macro Variables

Fig. 12. Forecast of Chile country macroeconomic variables. Shocks to inflation and GDP are imposed in year 2015.

XII. FORECASTING ECONOMIC IMPACT OF COVID 19 PANDEMIC

Lastly, we illustrate the application of the Python platform to forecast the impact of the COVID-19 virus. We utilize the model developed by Eichenbaum, Rebelo, and Trabandt (ERT), which integrates the New Keynesian framework with sticky prices and wages, alongside the epidemiological Susceptible-Infected-Recovered (SIR) model of virus transmission.

The original Omicron virus strain, which emerged in December 2020, was followed about six months later by the Delta strain, which proved to be more contagious and aggressive. While the Omicron variant prompted significant lockdown measures by the US government and led to a substantial recession in economic activity, the impact of the Delta strain was much milder. Consequently, we focus solely on the economic impact of the Omicron strain. An excerpt of the model file is displayed below.

name: Eichenbaum, Rebelo and Trabandt Model with Resistant Virus Strain.
....

# #
 # equilibrium equations: actual (sticky price) economy
# #

 # Eq.1. Production
- y: y=pbreve*A*k(-1)**(1-alfa)*n**alfa

38

 # Eq.2. Marginal cost
- mc: mc=1/(A*alfa**alfa*(1-alfa)**(1-alfa))*w**alfa*rk**(1-alfa)

 # Eq.3. Cost mininizing inputs
- w: w=mc*alfa*A*n**(alfa-1)*k(-1)**(1-alfa)

 # Eq.4. Law of motion capital
- k: k=x+(1-delta)*k(-1)

 # Eq.5. Aggregate resources
- x: y=c+x+g_ss

 # Eq.6. Aggregate hours
- n: n = s1(-1)*ns+i1(-1)*ni+r1(-1)*nr

 # Eq.7. Aggregate consumption
- c: c = s1(-1)*cs+i1(-1)*ci+r1(-1)*cr

Two-strain SIR model with vaccination
 # Eq.8. New infections
 # Strain #1
- tau1: tau1 = (pi1*s1(-1)*cs*i1(-1)*ci + pi2*s1(-1)*ns*i1(-1)*ni + pi3*s1(-1)*i1(-1))*(1-theta_lockdown*lockdown_policy)**2
 # Strain #2
- tau2: tau2 = (pi1*s2(-1)*cs*i2(-1)*ci + pi2*s2(-1)*ns*i2(-1)*ni + mult*pi3*s2(-1)*i2(-1))*virus_resistant_strain*(1-theta_lockdown*lockdown_policy)**2
 # Total new infection
- tau: tau = tau1 + tau2

 # Eq.9. Total susceptible
- s1: s1 = s1(-1) - tau1 - v
- s2: s2 = s2(-1) - tau2
- s: s = IfThenElse(s(-1)-tau-v,s(-1)-tau-v,0)

 # Eq.10. Total infected
 # Strain #1
- i1: i1 = i1(-1) + tau1 - (pir+pid)*i1(-1) + ei1
 # Strain #2
- i2: i2 = i2(-1) + (tau2 - (pir+pid/mult2)*i2(-1))*virus_resistant_strain + ei2
 # Total infected
- i: i = i1 + i2

 # Eq.11. Total recovered
- r1: r1 = r1(-1) + pir*i1(-1) + v
- r2: r2 = r2(-1) + pir*i2(-1)
- r: r = r1 + r2

 # Eq.12. Newly vaccinated
- v: v = vaccination_rate*s1(-1)

 # Eq.13. Total deaths
- dd: dd = dd(-1) + pid*i1(-1) + pid/mult2*i2(-1) + ed

 # Eq.14. Total population
- pop: pop = pop(-1) - pid*i1(-1) - pid/mult2*i2(-1)

….
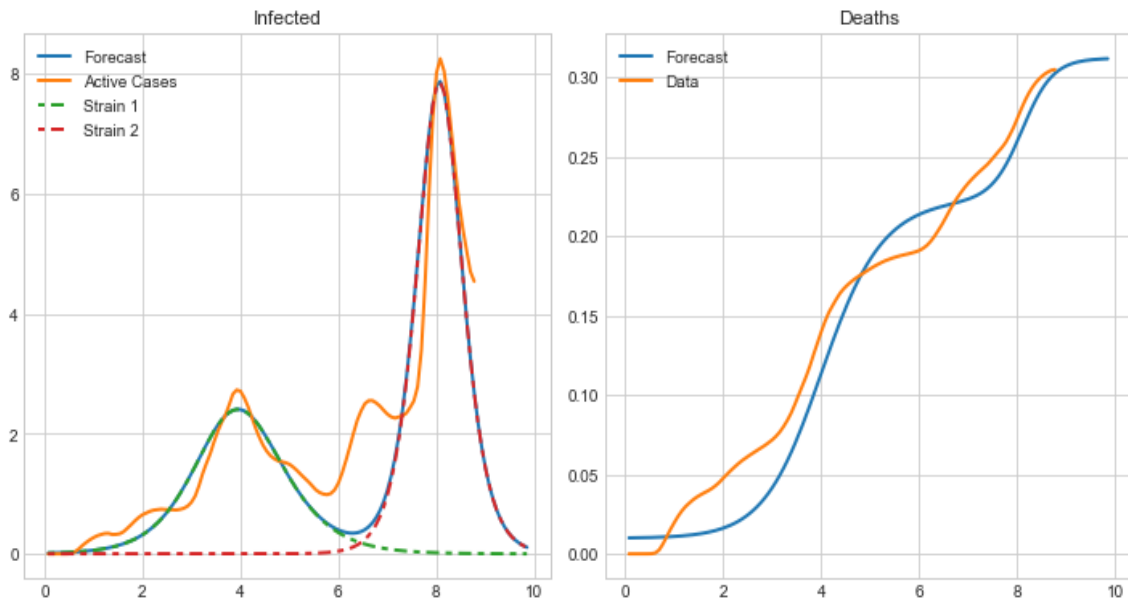Results of forecasts are shown below.

Fig. 13. The impact of the Omicron and Delta virus strains on the number of infected individuals and the number of deaths is illustrated below. The vertical axes represent the percentage of the population. The green and red lines indicate the transmission rates of the first and second virus strains, respectively, while the orange lines display the actual data.

Sticky and Flexible Price Economies
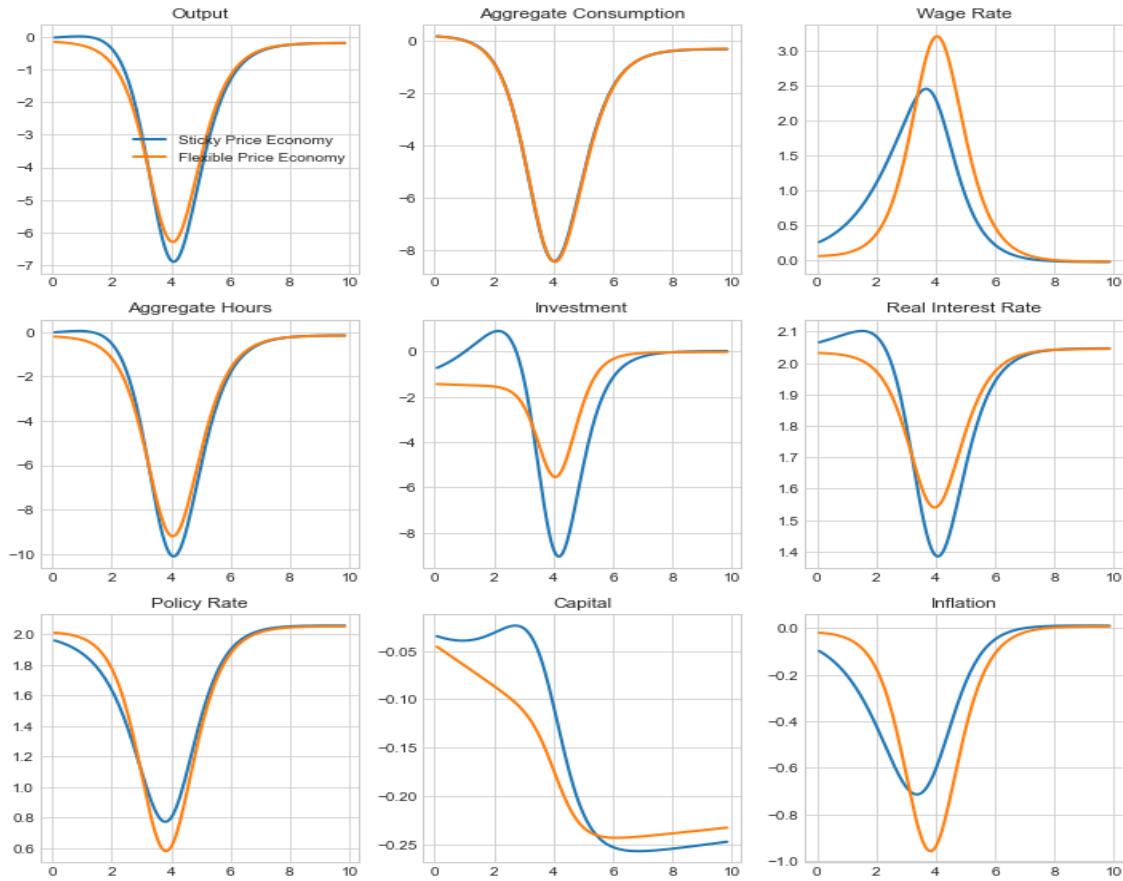
Fig. 14. Detrimental effects of the first COVID-19 virus strain on economic activity.

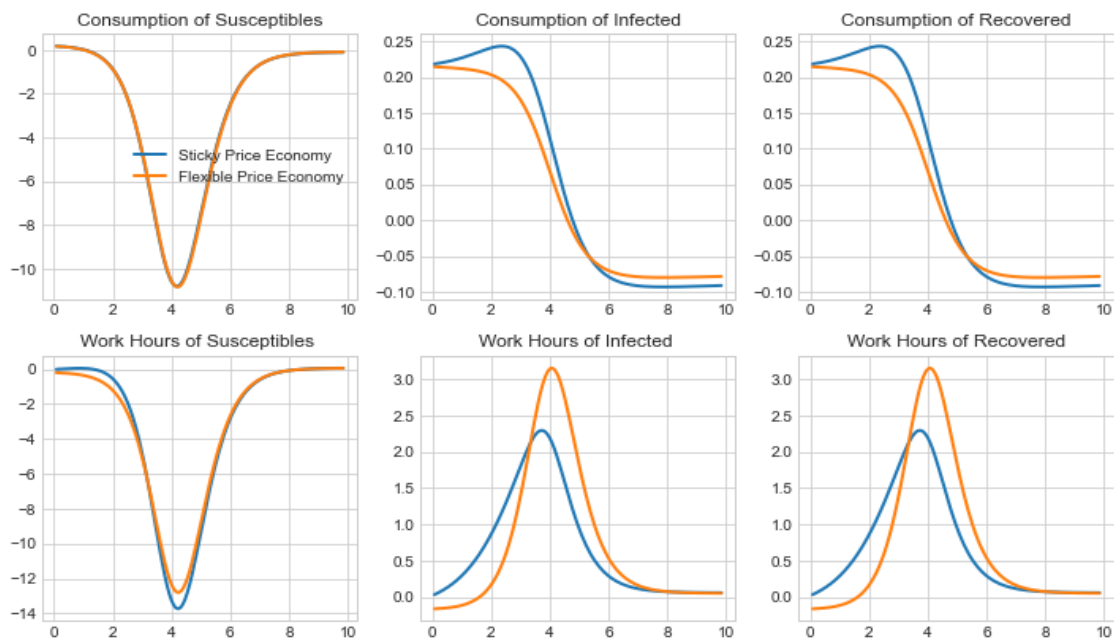## Sticky and Flexible Prices Economies (continued)

41

Fig. 15. The COVID-19 pathogen leads to a significant reduction in consumption among susceptible, infected, and recovered individuals. It results in a substantial decrease in working hours for the susceptible population, while there is only a minor increase in working hours for the infected and recovered individuals.

XIII. EXAMPLES

Below, we present four examples: running simple toy DSGE models, executing the Kalman filter, performing optimization, estimating the parameters of the Peter Ireland DSGE model, and running the South Africa Reserve Bank DSGE model.

Toy Model

```
from snowdrop.src.driver import run

def test(fname='models/JLMP98.yaml'):

    #fname = 'models/TOY/JLMP98.yaml'   # Simple monetary policy examplez
    #fname = 'models/Toy/RBC.yaml'      # Simple RBC model
    #fname = 'models/Toy/RBC1.yaml'      # Simple RBC model
    fout = 'data/test.csv' # Results are saved in this file
    decomp = ['PDOT','RR','RS','Y'] # List of variables for which decomposition plots are produced
    output_variables = None #['pie','r','y','ystar'] #['PDOT','RR','RS','Y','PIE','LGDP','G','L_GDP','L_GDP_GAP']

    # Function that runs simulations, model parameters estimation, MCMC sampling, etc...
    y,rng_date = \
    run(fname=fname,fout=fout,decomp_variables=decomp,
          output_variables=output_variables,
          Output=True,Plot=True,Solver="LBJ",
          #output_dir="C:/temp/out",
          graph_info=False,use_cache=False,Sparse=False)

if __name__ == '__main__':
    """ The main test program. """
    test()
```

Kalman Filter

```
from snowdrop.src.driver import importModel, kalman_filter


def test(fname='models/Ireland2004.yaml',fmeas='data/gpr_1948.csv'):

    #fname = 'TOY/Ireland2004.yaml' # Multivariate Kalman filter example
    #fname = 'TOY/MVF_US.yaml' # Multivariate Kalman filter example
    fout = 'data/results.csv' # Results are saved in this file
    output_variables = None #['DLGDP','LGDP','LGDP_BAR','PIE','UNR','UNR_GAP']   # List of variables that will be plotted or
displayed
    decomp = None #['pie','r']
```

```
# Instantiate model object
model = importModel(fname=fname,
            Solver="BinderPesaran",
            #Solver="Benes,AndersonMoore,LBJ,ABLR,BinderPesaran,Villemot
            #Filter="Particle",Smoother="Durbin_Koopman",
            #Filter="Unscented", Smoother="BrysonFrazier",
            Filter="Durbin_Koopman",Smoother="Durbin_Koopman",
            #Filter="Diffuse",Smoother="Diffuse",
            Prior="Diffuse", #Prior="StartingValues",  Prior="Diffuse",
            measurement_file_path=fmeas,model_info=True)


### Run Kalman filter
y,rng_date,epsilonhat,etahat = kalman_filter(model=model,meas=fmeas,fout=fout,Output=False,Plot=True)



if __name__ == '__main__':
    """ The main test program. """
    test()
```

Model Estimation

```
from snowdrop.src.driver import importModel, estimate


def test(fname='models/Ireland2004.yaml',fmeas='data/gpr_1948.csv'):

    fout = 'data/results.csv' # Results are saved in this file
    output_variables = None  # List of variables that will be plotted or displayed

    # Create model object
    model = importModel(fname=fname,Solver="Benes",
                # Available Kalman filters: Durbin_Koopman,Particle
                Filter="Durbin_Koopman",Smoother="Durbin_Koopman",
                #Filter="Non_Diffuse_Filter",Smoother="BrysonFrazier",
                # Available priors: StartingValues,Diffuse,Equilibrium
                Prior="Diffuse",
                # Available methods: pymcmcstat-mh,pymcmcstat-am,pymcmcstat-dr,pymcmcstat-dram,emcee,particle-pmmh,particle-
smc
                SamplingMethod="pymcmcstat-dr",
                measurement_file_path=fmeas,use_cache=False)

    # Estimate model parameters
    estimate(model=model,output_variables=output_variables,
            #estimate_Posterior=True,
            estimate_ML=True,
            # Available algorithms: Nelder-Mead, L-BFGS-B, TNC, SLSQP, Powell, and trust-constr
            algorithm="SLSQP",
            sample=True,Ndraws=2000,Niter=100,burn=100,Parallel=False,
            fout=fout,Output=False,Plot=True)
```

```python
if __name__ == '__main__':
    """ The main test program. """
    test()
```

South Africa Reserve Bank Model

```python
import os
import numpy as np
import datetime as dt
import pandas as pd
from snowdrop.src.utils.merge import merge
from snowdrop.src.misc.termcolor import cprint
from snowdrop.src.driver import run
from snowdrop.src.driver import findSteadyStateSolution
from snowdrop.src.utils.getIrisData import getIrisModel
from snowdrop.src.driver import kalman_filter
from snowdrop.src.numeric.solver.util import find_residuals
from snowdrop.src.model.settings import SolverMethod
from snowdrop.src.numeric.solver.linear_solver import solve
from snowdrop.src.utils.equations import getVariablesPosition as getTopology
from snowdrop.src.graphs.util import plotTimeSeries


def test(file_path = 'models/model_Lshocks.model',
         calib_path = 'data/QPMcalibration_dict.xlsx',
         hist_path = 'data/history.csv'):

    fout = 'data/results.csv'    # Results are saved in this file
    output_variables = ['lgdp_gap','lgdp','lcpi','lcpi_core','dot4_cpi','rr','rn','dot_w','dot4_emp']      # List of variables that will be displayed
    decomp_variables = ['lz_gap','lgdp_gap','lemp_gap','rr_gap']  # List of variables for which decomposition plots are produced
    filtered = None

    # Path to model file
    fpath = os.path.abspath(os.path.join(os.path.dirname(__file__),file_path))

    # Path to model calibration file
    df = pd.read_excel(calib_path,"params")
    var = [x.strip() for x in df["var"]]
    val = [float(x) for x in df["value"]]
    calibration = dict(zip(var,val))

    #Path to shock std dev
    df = pd.read_excel(calib_path,"std")
    var = [x.strip() for x in df["var"]]
    val = [float(x) for x in df["value"]]
    stddev = dict(zip(var,val))
    calibration = {**calibration, **stddev}

    #---------- 1. Instantiate model
    # Please use parameter 'use_cache' with caution...
    # If set to True it will read a model from a file neglecting calibrations and conditions settings.
```

```
# If set to False it will save (aka serialize) this model in a file with the new set of conditions and calibration parameters.
cprint("Parsing model file...\n","blue")
model = getIrisModel(fpath,calibration=calibration,conditions={"fiscalswitch":False,"wedgeswitch":True},
            use_cache=True,check=False,debug=False)
#print(model)
variables_names = model.symbols["variables"]
n = len(variables_names)
variables_values = model.calibration["variables"]
var_labels = model.symbols["variables_labels"]
param_names = model.symbols["parameters"]
param_values = model.calibration["parameters"]
params = dict(zip(param_names,param_values))
#print(params)
model.calibration["variables"] = np.zeros(len(variables_names))
shock_names = model.symbols["shocks"]

model.options["frequency"] = 1 # Quarterly
# Steady state is computed as the numerical solution at the end of this time interval
model.options["ss_interval"] = 400
is_linear = model.isLinear

#---------- 2. Compute steady state
cprint("Computing steady state...\n","blue")
ss_values, ss_growth = findSteadyStateSolution(model=model)
ss = dict(zip(variables_names,ss_values))
ss_gr = dict(zip(variables_names,ss_growth))
print("Steady state:")
print("-"*13)
i = 0
for x in sorted(variables_names):
    if not "_plus_" in x and not "_minus_" in x:
        i += 1
        lbl = None #var_labels[x] if x in var_labels else ""
        if bool(lbl):
            print(f"{i}: {lbl} - {x} = ({ss[x]:.3f}, {ss_gr[x]:.3f})")
        else:
            print(f"{i}: {x} = ({ss[x]:.3f}, {ss_gr[x]:.3f})")
print()

# Set starting values
ss_vars = [ss[x] if x in ss else variables_values[i] for i,x in enumerate(variables_names)]
model.calibration["variables"] = ss_vars

#---------- 3. Solve linear model
# Compute reduced form transition and shock matrices.
# If model is non-linear, then use steady state.

cprint("Solving linearized model","blue")
getTopology(model)
model.SOLVER = SolverMethod.Benes
solve(model,steady_state=ss_values)
```

```python
# --------------- 4. Run IRFs
if True:
    cprint("\nRunning IRFs...\n","blue")
    model.options["range"] = ["2020-1-1","2030-1-1"]
    model.options["periods"] = ["2021-1-1"]

    shock_names = model.symbols["shocks"]
    n_shocks = len(shock_names)

    model.calibration["variables"] = ss_vars
    model.isLinear = is_linear

    # Define shocks
    shocks = [0.1]
    list_shocks = ['e_lgdp_gap']
    num_shocks = len(list_shocks)

    for i in range(num_shocks):
        # Set shocks
        shock_name = list_shocks[i]
        ind = shock_names.index(shock_name)
        shock_values = np.zeros(n_shocks)
        shock_values[ind] = shocks[i]
        model.options["shock_values"] = shock_values

        y1,rng_date1 = \
        run(model=model,decomp_variables=decomp_variables,
            output_variables=output_variables,Solver="LBJ",
            fout=fout,Output=True,Plot=True,irf=True,MULT=2)


# -----------5.  Run Kalman Filter and Smoother
if True:
    cprint("Running Kalman filter and smoother...\n","blue")
    # Set simulation and filter ranges
    simulation_range = [[1990,1,1],[2023,1,1]]
    filter_range = [[2001,1,1],[2023,1,1]]
    model.options['range'] = simulation_range
    model.options['filter_range'] = filter_range
    model.options["shock_values"] = np.zeros(len(shock_names))
    start_filter = dt.date(*filter_range[0])
    end_filter = dt.date(*filter_range[1])
    start_simulation = dt.date(*simulation_range[0])
    end_simulation = dt.date(*simulation_range[1])

    # set path to a file with filtered results
    smoother_path = 'data/QPM/smoother_results.csv'

    #Set variables starting values from historical data
    calib = model.calibration["variables"].copy()
    ss = dict(zip(variables_names,calib))
    model.setStartingValues(hist=hist_path,bTreatMissingObs=True,debug=False)
    #starting_values = dict(zip(variables_names,model.calibration["variables"]))
```

```python
# Get filtered and smoothed endogenous variables
# We apply Kalman filter for linearized model
model.isLinear = True
y2,rng_date2,epsilonhat,etahat = \
    kalman_filter(model=model,Output=True,Plot=False,fout=smoother_path,meas=hist_path,
            #Filter="Diffuse",Smoother="Diffuse",Prior="Diffuse",
            Filter="Durbin_Koopman",Smoother="Durbin_Koopman",Prior="Equilibrium")
# filtered results
filtered = y2[0]
# smoothed results
smoothed = y2[-1]
results = smoothed
rows,columns = results.shape

# Save filtration results
dct = {}
for j in range(columns):
    n = variables_names[j]
    data = results[:,j]
    m = min(len(data),len(rng_date2))
    ts = pd.Series(data[:m], rng_date2[:m])
    dct[n] = ts[start_filter:end_simulation]

# Get shocks and residuals
shocks = model.symbols['shocks']
n_shk = len(shocks)
res = find_residuals(model,results)
m = min(len(rng_date2),len(data))
if etahat is None:
    for j in range(n_shk):
        n = shocks[j]
        data = res[:,j]
        ts = pd.Series(data[:m], rng_date2[:m])
        dct[n] = ts[start_filter:end_simulation]
        dct[n+"_other"] = pd.Series(np.zeros(m), rng_date2[:m])[start_filter:end_simulation]
else:
    for j in range(n_shk):
        n = shocks[j]
        data = etahat[:,j]
        m = min(len(data),len(rng_date2))
        ts = pd.Series(data[:m], rng_date2[:m])
        dct[n] = ts[start_filter:end_simulation]
        data = res[:,j]
        m = min(len(data),len(rng_date2))
        ts2 = pd.Series(data[:m], rng_date2[:m])
        dct[n+"_other"] = 0*(ts2-ts)[start_filter:end_simulation]

date = dt.datetime(*filter_range[1])
filtered = [dct[n][date] for n in variables_names]

# Read historical data
ext = hist_path.split(".")[-1].lower()
```

```
if ext == 'xlsx' or ext == 'xls':
    df = pd.read_excel(hist_path,header=0,index_col=0,parse_dates=True)
else:
    df =
pd.read_csv(filepath_or_buffer=hist_path,sep=',',header=0,index_col=0,parse_dates=True,infer_datetime_format=True)


### Plot results
output_variables = [x for x in variables_names if "_gap" in x and x in var_labels]
#output_variables = variables_names
series = []; labels = []
header = "Kalman_Filter"
titles = [var_labels[k] for k in variables_names if k in output_variables and k in var_labels]
for k in dct:
    if k in output_variables:
        arr = []
        lbls = ["filter ("+k+")"]
        if k in df.columns:
            lbls.append("data")
            val = df[k][start_filter:end_simulation]
            # diff = (dct[k]-val)[:end_filter]
            # shift = np.nanmean(diff)
            # arr.append(dct[k]-shift)
            arr.append(val)
        else:
            arr.append(dct[k])
        series.append(arr)
        labels.append(lbls)


plotTimeSeries(path_to_dir='graphs',header=header,titles=titles,labels=labels,series=series,sizes=[3,1],fig_sizes=(6,8),save=True)


    files = []
    outputFile = "graphs/Gap Variables.pdf"
    list_names = ["graphs/"+x for x in os.listdir("graphs") if x.startswith(header) and x.endswith(".pdf")]
    for f in list_names:
        files.append(f)
    merge(outputFile,files)

# ---------------- 5. Impose user judgements
#### Run forecast with tunes
if True:
    cprint("Running forecasts with tunes...\n","blue")
    # Set simulation range
    model.options['range'] = ["2020-1-1","2028-1-1"]
    model.options["periods"] = ["2021-1-1"]
    rng = pd.date_range(start="2022-1-1",end="2028-1-1",freq='QS')
    model.isLinear = True
    model.anticipate = False

    # Set starting values
    model.calibration["variables"] = ss_vars if filtered is None else filtered

    # Swap endogenous and exogenous variables
```

```python
    m = {}
    m['w_food'] = pd.Series([0.1714]*len(rng),rng)
    m['w_petr'] = pd.Series([0.0482]*len(rng),rng)
    m['w_elec'] = pd.Series([0.0363]*len(rng),rng)
    m['w_goodsx'] = pd.Series([0.2308]*len(rng),rng)
    m['w_serv'] = pd.Series([0.5132]*len(rng),rng)
    m['w_bfp'] = pd.Series([0.5094]*len(rng),rng)

    exog_shocks = ["e_w_food","e_w_petr","e_w_elec","e_w_goodsx","e_w_serv","e_w_bfp"]
    model.swap(var1=m,var2=exog_shocks)

    output_variables = ['w_food','w_petr','w_elec','w_goodsx','w_serv','w_bfp']     # List of variables that will be displayed
    decomp_variables = ['lgdp_gap','lemp_gap','w_petr','w_bfp']  # List of variables for which decomposition plots are produced

    # Define shocks
    shocks = [0.1]
    list_shocks = ['e_lgdp_gap']
    num_shocks = len(list_shocks)

    for i in range(num_shocks):
        # Set shocks
        shock_name = list_shocks[i]
        ind = shock_names.index(shock_name)
        shock_values = np.zeros(n_shocks)
        shock_values[ind] = shocks[i]
        model.options["shock_values"] = shock_values

        y3,rng_date = run(model=model,output_variables=output_variables,
                    decomp_variables=decomp_variables,
                    fout=fout,Output=True,Plot=True,output_dir="out")

    print("\nDone!")

if __name__ == '__main__':
    """ The main test program."""
    test()
```

Optimization Example

```python
from snowdrop.src.driver import optimize

def test(fname='models/armington.yaml'):
    #fname = 'models/melitz.yaml' # Melitz model example
    #fname = 'models/krugman.yaml' # Krugman model example
    #fname = 'models/armington.yaml' # Armington model example
    #fname = 'models/transport.yaml' # Transportation expenses minimization example

    plot_variables = ["c","Y","Q","P"]

    # Run optimization routine.
    optimize(fpath=fname,Output=True,plot_variables=plot_variables,model_info=False)
```

```
if __name__ == '__main__':
    """ The main test program. """
    test()
```

Peter Ireland's Model File

name: Technology Shocks in the New Keynesian Model
 # Peter Ireland, 2004, TECHNOLOGY SHOCKS IN THE NEW KEYNESIAN MODEL
 # http://ireland   com/pubs/tshocksnk.pdf

symbols:

  variables : [y, x, r, g, pie, a, e]

  measurement_variables : [obs_g, obs_pie, obs_r]

  shocks : [epsa, epse, epsz, epsr]

  measurement_shocks : [res_obs_g, res_obs_pie, res_obs_r]

  parameters : [beta, psi, omega, alphax, alphapie,
          rhopie, rhog, rhox, rhoa, rhoe]

  # measurement_parameters : []

equations:
    # Linear Model (equations 5,11,14,13,15,2,9)
    *- a = rhoa*a(-1) + epsa*
    *- e = rhoe*e(-1) + epse*
    *- x = alphax*x(-1) + (1-alphax)*x(+1) - (r-pie(+1)) + (1-omega)*(1-rhoa)*a*
    - pie = beta(alphapie*pie(-1)+ (1-alphapie)*pie(+1)) + psi*x - e*
    *- x = y - omega*a*
    - g = y - y(-1) + epsz
    *- r = r(-1) + rhopie*pie + rhog*g + rhox*x + epsr*

measurement_equations:
  - obs_g   = g   + res_obs_g
  - obs_pie = pie + res_obs_pie
  - obs_r   = r   + res_obs_r

calibration:
  # parameters:
  beta :   0.99
  psi :   0.1
  omega :   0.0617
  alphax : 0.0836
  alphapie : 0.0001
  rhopie : 0.3597
  rhog :   0.2536
  rhox :   0.0347

50
```

*rhoa :      0.9470*
*rhoe :      0.9625*


*# starting values for endogenous variables:*
*y:   0.0*
*pie: 0.0*
*r: 0.0*
*g: 0.0*
*x: 0.0*
*a: 0.0*
*e: 0.0*


*#Standard deviation of shocks:*
*std_epsa: 0.0405*
*std_epse: 0.0012*
*std_epsz: 0.0109*
*std_epsr: 0.0031*


*#Standard deviations of observation errors:*
*std_res_obs_g:   0*
*std_res_obs_pie: 0*
*std_res_obs_r:   0*


*estimated_parameters:*
  *# Please choose one of the following distributions:*
  *# normal_pdf,lognormal_pdf,beta_pdf,gamma_pdf,t_pdf,weibull_pdf,inv_gamma_pdf,inv_weibull_pdf,wishart_pdf,inv_wishart_pdf*
  *# PARAM NAME, INITVAL, LB, UB, PRIOR_SHAPE, PRIOR_P1, PRIOR_P2, PRIOR_P3, PRIOR_P4, PRIOR_P5*
  *# The first parameter is the parameter name, the second is the initial value, the third and*
  *# the fourth are the lower and the upper bounds,the fifth is the prior shape,*
  *# the sixth to tenth are prior parameters (mean, standard deviation, shape, etc.).*
  *# Parameters*
  *- omega,    0.06, 1.e-7, 1,  normal_pdf, 0.20, 0.10*
  *- alphax,   0.08, 1.e-7, 1,  normal_pdf, 0.10, 0.05*
  *- alphapie, 0.00001, 1.e-7, 1,  normal_pdf, 0.10, 0.05*
  *- rhoa,     0.9, 1.e-7, 1,  normal_pdf, 0.85, 0.10*
  *- rhoe,     0.9, 1.e-7, 1,  normal_pdf, 0.85, 0.10*
  *- rhopie,   0.3, 1.e-7, 1,  normal_pdf, 0.30, 0.10*
  *- rhog,     0.2, 1.e-7, 1,  normal_pdf, 0.30, 0.10*
  *- rhox,     0.03, 1.e-7, 1,  normal_pdf, 0.25, 0.0625*


  *# Endogenous variables shocks*
  *# should we use inverse wishart distribution for sigma^-2: inverse wishart_pdf ?*
  *- std_epsa, 0.0405, 1.e-7, 1,  normal_pdf, 0.03,    1.80*
  *- std_epse, 0.0012, 1.e-7, 1,  normal_pdf, 0.0000727, 0.0000582*
  *- std_epsz, 0.0109, 1.e-7, 1,  normal_pdf, 0.005,    0.275*
  *- std_epsr, 0.0031, 1.e-7, 1,  normal_pdf, 0.005,    0.004417*


  *# # Measurement variables shocks*
  *# - std_res_obs_g,   0.01,  0, 10, normal_pdf, 0.03, 2*
  *# - std_res_obs_pie, 0.01,  0, 10, normal_pdf, 0.03, 2*
  *# - std_res_obs_r,   0.01,  0, 10, normal_pdf, 0.03, 2*

*labels:*
  *y:      "Output"*
  *y(-1):   "Lag of Output"*
  *x:       "Output Gap"*
  *x(-1):   "Lag Output Gap"*
  *r:       "Interest Rate"*
  *r(-1):   "Lag of Interest Rate"*
  *pie:      "Inflation"*
  *g:       "Output Growth"*
  *pie(+1): "Lead of Inflation"*
  *pie(-1): "Lag of Inflation"*
  *a:       "Total Factor Productivity"*
  *a(-1):   "Lag of Total Factor Productivity"*
  *e:       "Aggregate Technology AR(1) Process"*
  *e(-1):   "Lag of Aggregate Technology AR(1) Process"*
  *epsa:    "Preference Shock"*
  *epse:    "Cost-Push Shock"*
  *epsz:    "Shock to Output Gap"*
  *epsr:    "Shock to Interest Rate"*


*options:*
  *range : ["1948,1,1","2002,12,31"]*
  *filter_range : ["1948,4,1","2002,12,31"]*


# XIV.  APPENDICES

## Graphical User Interface

To enhance model development, we have designed a graphical user interface (GUI) that enables users to input model equations, endogenous and exogenous variables, parameters, and shocks. To launch the interface, execute the script located at *src/gui/clientGui.py*.

This GUI features multiple panels for entering model equations, parameters, exogenous and endogenous variables, shock values, and their timing. Users can specify the frequency of time series and the simulation time range. Instead of manually entering model equations, variables, and parameters, users can simply open a model file, which the GUI will automatically parse to display the model settings.

The buttons at the bottom of the GUI facilitate various operations, which are as follows:
- **CLOSE**: Closes the GUI application.
- **RESET**: Resets the GUI and clears all text boxes.
- **SAVE TEMPLATE**: Saves the user's model settings into a YAML file for later retrieval and restoration.
- **OPEN_MODEL_FILE**: Opens a model file and displays its settings in the GUI. It supports parsing DYNARE and IRIS model files, as well as files in YAML, XML, and TEXT formats.
- **FIND STEADY STATE**: Identifies and displays the model's steady state in the Steady State tab. If a range of parameters is specified, it calculates steady states at intervals of one-tenth of this range and generates plots in the tabs labeled Steady State Figure #1, Steady State Figure #2, etc.
- **IMPULSE RESPONSE FUNCTION**: Calculates and displays impulse response functions (IRFs) for user-specified shocks in tabs labeled Figure #1, Figure #2, etc.
- **RUN SIMULATION**: Executes forecasts based on the specified time range and initial values of the endogenous variables.

Fig.16. The equations editor enables users to input model equations along with the names of variables, parameters, and shocks, and to execute forecasts.

If the parameter range text box is left empty, the framework will calculate the model's steady state using the parameters specified in the *Parameters* text box. In this case, this text box should remain empty. Users can also define lower and upper bounds for some parameters in the *Parameters Range* text box, for example, $a = 0.6 - 0.7$. The remaining parameters will be fixed at the values set in the *Parameters* box. The format for specifying the range is as follows: parameter name followed by an equal sign and then followed by the lower and upper bounds.

The figure above illustrates $a$ parameter range from 0.6 to 0.7. In this scenario, ten steady states will be generated, as shown in Figure 17. Additionally, plots of steady state variables as functions of these parameters will be created (see Figure 18).

```
                          Steady-State Solution:
              Name                                Value
a                                      0.6
A                                      0.347
C                                      1.86
K                                      15.991
Y                                      2.341
r                                      0.036

a                                      0.611
A                                      0.35
C                                      1.871
K                                      16.006
Y                                      2.353
r                                      0.036

a                                      0.622
A                                      0.353
C                                      1.882
K                                      16.02
Y                                      2.364
r                                      0.036

a                                      0.633
A                                      0.357
C                                      1.892
K                                      16.035
Y                                      2.375
r                                      0.037

a                                      0.644
A                                      0.36
C                                      1.903
K                                      16.049
Y                                      2.386
r                                      0.037

a                                      0.656
A                                      0.363
C                                      1.913
K                                      16.064
Y                                      2.397
r                                      0.037
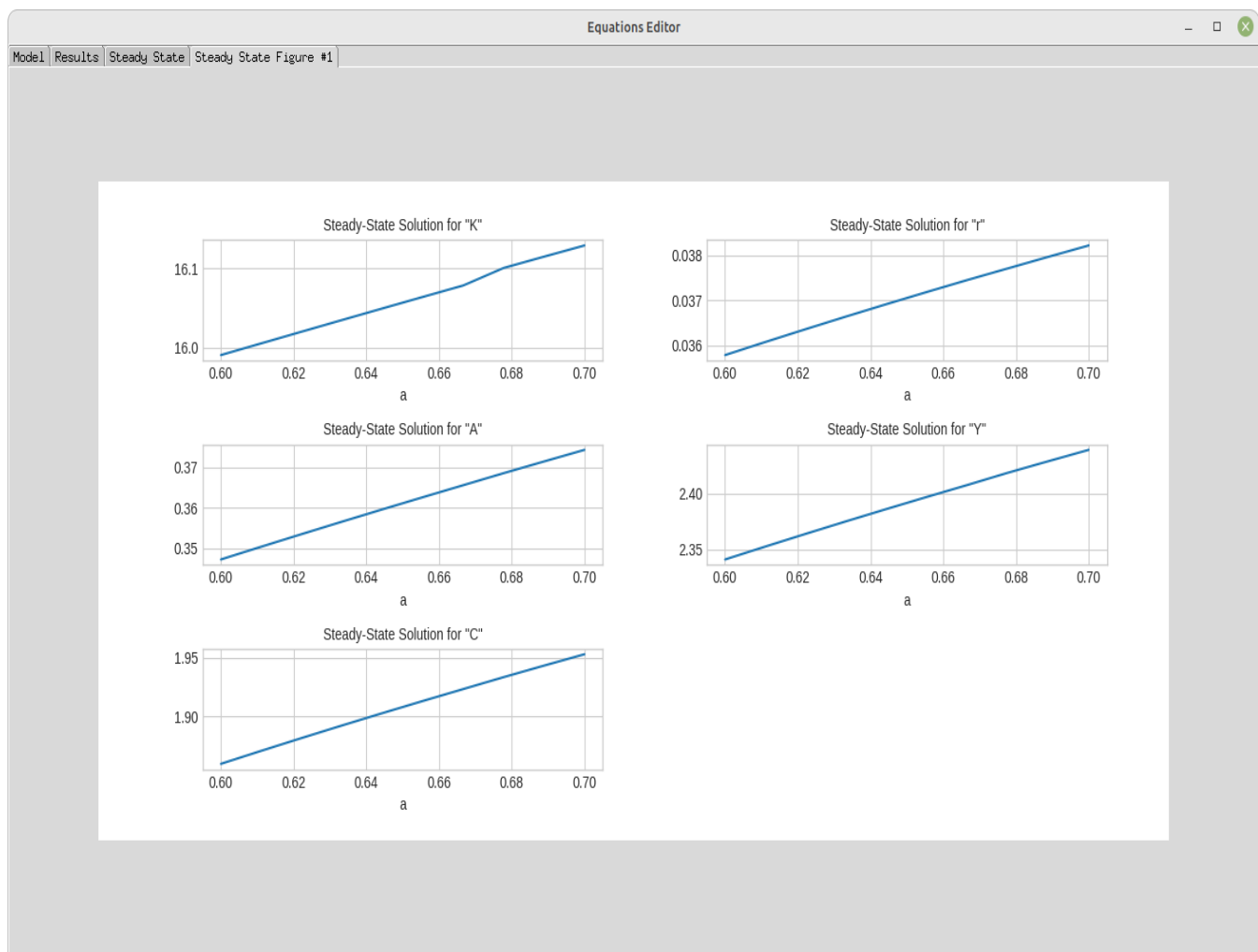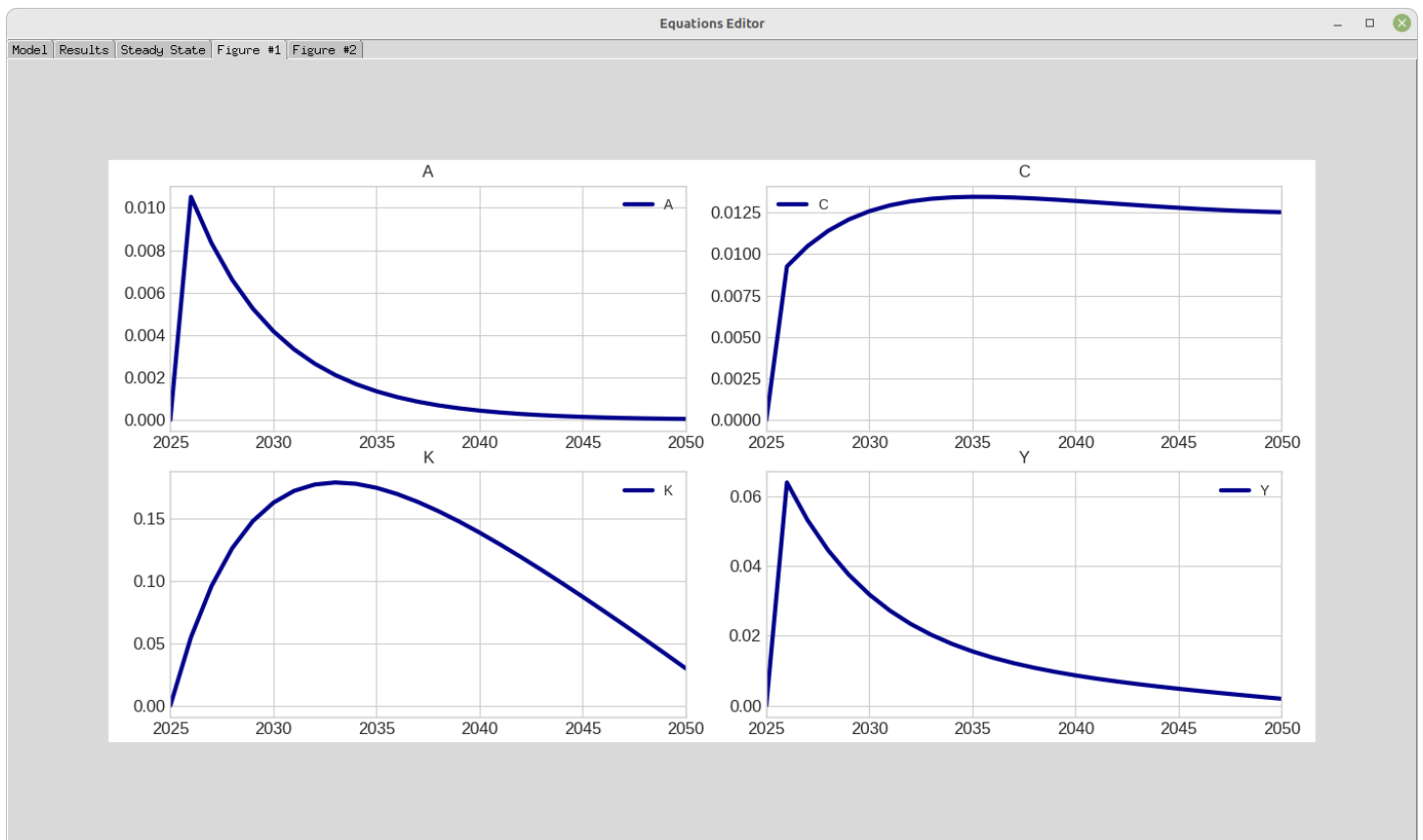
a                                      0.667
A                                      0.366
C                                      1.923
```

Fig.17. The steady state values are calculated for ten parameters within the range of a from 0.6 to 0.7.

Fig.18. The steady state plot for parameters range of a from 0.6 to 0.7.

Calculating steady states across a range of parameters can provide insights into the stability and properties of the model.

When users run simulations, the model specifications are saved into a temporary YAML model file for subsequent analysis. The results of these simulations are displayed in tabular format within the Results tab.

Fig.19. The forecast variables are displayed in tabular format.

In addition to this data, the plots are displayed in Figure #1, Figure #2, etc.

These plots, along with the impulse response function (IRF) and steady state plots, enable researchers to examine the characteristics and properties of the developed model.

Fig.19. Graphs illustrate the forecasted model variables.