Python Framework

Table of Contents

I. INTRODUCTION

The growth in complexity and scale of macro-finance models over the past couple of decades aided by computational advances cannot be understated. On the software side, specialized applications like DYNARE and IRIS Macroeconomic Modeling Toolboxes, TROLL software etc. were developed to provide economists with an integrated platform to input their models, import data, perform the desired computational tasks (solve, simulate, calibrate or estimate) and obtain well formatted post process output in the form of tables, graphs etc. Each application has its own advantages. Ease of use via a user-friendly interface, combined with the capability to handle variety of models have led to the immense popularity of DYNARE among general equilibrium modelers. However, DYNARE can handle only stationary DSGE models and requires user to write models in a stationary way by introducing variables deflators, for example. IRIS macroeconomic toolbox is another great tool and it gained popularity among economists to analyze non-stationary DSGE models. Troll,

on the other hand, is specialized to efficiently solve and simulate large models very large systems of equations. All these applications, however, are either commercial, or rely on commercial software to run, that require expensive licensing costs. There is no integrated software package to our knowledge that is both flexible to handle a wide class of models and is available for free under the GNU General Public Licensing agreements. This framework, built entirely on Python, is intended to fill that void. Additionally, Platform can read, and parse model files developed by Iris/Dynare/Troll/Sirius software and run simulations. User may specify model variables, equations, parameters on fly that are not necessary defined in a model file.

Python platform is built to analyze and estimate the following classes of theory based dynamic models: New Keynesian DSGE, Real Business Cycle and Overlapping Generations, 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 are provided with the option to input their models in human readable format via a YAML (human readable) file that also includes the data source. options include directly exporting results to a CSV file, to python sqlite database, as well as obtaining graphs and tables to the desired set of variables and parameters. Further details of these processing and output options are described in the following sections.

Below are highlights of this Framework:

- Framework is written in Python language and uses only Python libraries that are available by installing Python distribution from 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 to Windows, Linux, Unix, Mac platforms.
- It is parallelized and can be run on CPU/GPU cores.
- 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 source (Haver, ECOS, EDI, and World Bank databases) and options relating to sample size, subsample choices, etc.
- Estimate parameters of linear and non-linear DSGE models with rational expectations.
- Non-linear equations are solved by iterations by Newton's method. Two algorithms are implemented: ABLR stacked matrices method and LBJ forward-backward substitution method.
- Linear models are solved with Binder Pesaran's method, Anderson, More's method and two generalized Schur's decomposition method that reproduce calculations employed in DYNARE and IRIS software.
- Choice of filtration (standard Kalman, unscented Kalman, diffuse Kalman, band pass, Hodrick–Prescott, LRX).
- Maximum likelihood estimates of parameters.
- Bayesian estimation of parameters by MCMC aided posterior sampling.
- Posterior sampling of parameters using Sequential Monte Carlo, affine invariant ensemble sampler algorithm or Random Walk Metropolis-Hastings algorithm.
- Non-linear models can be run with time dependent parameters.
- Solve, calibrate, and simulate RBC and OLG models.

- Generate forecasts and impulse response functions.

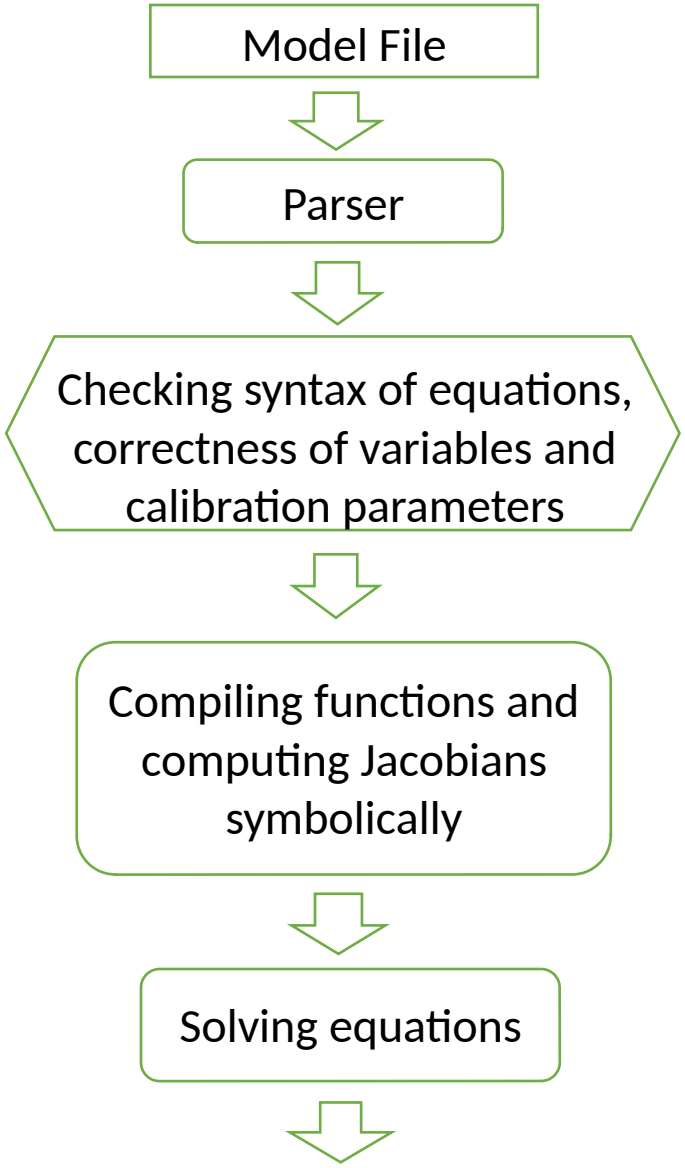
II. INSTALLATION

To install package “snowdrop” change directory to tar file and run in a command line:

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

III. STRUCTURE OF THE PROGRAMS

This framework is written in Python language and uses several Python libraries such as numpy, pandas, scipy, sympy, matplotlib, ruamel.yaml, ast. While the first five libraries are used for scientific computing and plotting the results, the last two are employed for parsing model files and representing mathematical expressions and the structure of the code. The diagram below shows the flowchart of code execution.

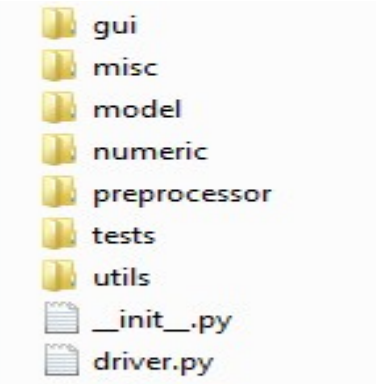


Each step of this flowchart is hand

Plotting and saving results

The code is written by using object-oriented programming style and is modularized for reusab

The framework "*src*" folder contains several subfolders as shown below:



The names of the subfolders are self-explanatory. For example, the "*gui*" folder contains graphical user interface which helps user to edit equations, specify endogenous and exogenous variables, and define shocks. The "*misc*" folder contains modules to check syntax of model file including correctness of equations, variables and parameters. The "*model*" folder contains modules defining model class blueprint and model object creation. The "*numeric*" folder is comprised of modules which solve equations and modules that implement Kalman filter, LRX filter, band-pass filter.

The "*preprocessor*" folder contains modules that compile equations functions and compute partial derivatives of the first order (Jacobian), the second order (Hessian), and the third order. While for most macroeconomic model only the first order derivatives are sufficient, for nonlinear models, which employ rational expectations econimic concept, the higher order derivatives may be necessary.

The "*tests*" folder contains modules which can be used to run different economic models. Finally, the utils folder, contains modules which can read and parse model files of IRIS/Dynare/TROLL software.

Examples of model files in Yaml/Iris/DYNARE format are located under models folder.

IV. Model File and Rules

Below we describe specifications of a model file written in YAML format. YAML is the human-readable data serialization language. Below is the short list of rules that are required when creating model file:

- Use white space 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 comma space.
- Associative arrays are represented using the colon space (:) in the form *variable: value*.
- Use hash symbol (#) for comments.

The model file needs to have the following sections: *name*, *symbols*, *equations*, and *calibration*. The *name* section describes the economic model. The *symbols* section is comprised of *variables*, *shocks*, and *parameters* subsections.

The *variables* subsection lists names of the endogenous variables, the *shocks* section lists the names of the shocks, and the *parameters* subsection list names of parameters and exogenous variables. The *equations* section lists model equations, and the *calibration* section lists the starting values of endogenous variables, parameters, and exogenous variables values. Additionally, model file can contain *options* section. It can be used to specify simulation time range, the value and timing of shocks, and parameters of multivariate normal distribution of shocks.

Below we show 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 14 periods and the shock *e* to output *Y* occurs at period 2. To set multiple shock user can specify multiple periods and multiple shock values. For example,

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

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

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

In many cases steady state is unknown and is found as part of 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, for example, mean of the shocks and their covariances. An example of 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 lists values of parameters and starting values of endogenous variables. Parameters of non-linear models can change with time. For example, parameter delta can be given by vector, [0.03,0.03,0.01,0.02]. This means that at period 1,2 delta = 0.03, at period 3 delta = 0.01, and thereafter delta = 0.02.

Framework implements simple macro language concepts such “include” and “sets”. Example below illustrates that a model file may include references to other YAML files. This makes this model file compact and easy to read. Python parses model files and inserts 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_yy, 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 usage of sets for five countries economy. These are US, EU, EA, Japan and est of the world. The output of these countries is aggregated with weights to compute world output.

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

This setup expands model equations for five regions:

```
====================
Five regions economy
====================
Model:
------
name: "Five regions economy"
file: "C:\work\Platform\examples\templates\countries.yaml

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)*

Model file structure is quite generic. In certain cases, user may want to find a solution to minimization or maximization problem of an objective function given linear or non-linear constraints. Example below illustrates 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 cities. Supply side of factories is limited by certain amounts a(i) from above while demand side amounts b(j) for markets is limited from below. Distances from factories to markets d(i)(j) are given in calibration section as well as cost of transportation cost(i)(j) per mile. The objective function is 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, 100]
    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

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 output below:

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

Model:
------
name: "Transportation expenses maximization model"
file: "C:\work\Platform\examples\models\OPT\transport.yaml

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_Chicago+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: 1.32e+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_Chicago	300	x_Seattle_Topeka	0
x_Seattle_Topeka	0	x_Seattle_Dallas	0
x_Seattle_Dallas	0	x_SanDiego_NewYork	275
x_SanDiego_NewYork	275	x_SanDiego_Chicago	0
x_SanDiego_Chicago	0	x_SanDiego_Topeka	100
x_SanDiego_Topeka	100	x_SanDiego_Dallas	200
x_SanDiego_Dallas	200		

Table.1. Shipment amounts from factories to markets. Factories and market's locations are shown in "Var Name" column.

Example below presents model file for Armington trade equilibrium with iceberg costs. This type of models falls under umbrella of Computable General Equilibrium (CGE) models. CGE models are solved by 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'

Platform parses this model file and generates equations for each region and good in the set:

Number of declared equations: 4, variables: 4, constraints: 4
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
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 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_R2	5.4	Q_G1_R3	7.1
Q_G1_R3	7.1	P_G1_R1	1.1
P_G1_R1	1.1	P_G1_R2	1.6
P_G1_R2	1.6	P_G1_R3	0.7
P_G1_R3	0.7	c_G1_R1	1.1
c_G1_R1	1.1	c_G1_R2	0.7
c_G1_R2	0.7	c_G1_R3	1.1
c_G1_R3	1.1	Y_G1_R1	3.5
Y_G1_R1	3.5	Y_G1_R2	5
Y_G1_R2	5	Y_G1_R3	5
Y_G1_R3	5		

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

V. COMPUTING FUNCTIONS AND JACOBIANS

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

Below we show 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]
```

VI. MODEL SETTINGS

Platform uses several numerical algorithms to solve model equations. The first two in the table below are solvers for non-linear models, and the last four – for linear models. The LBJ solver is named after Laffargue, Boucekkine, and Juillard who invented this forward-backward substitution algorithm. It is applied to solve system of equations with dense matrices. The ABLR solver solves system of stacked equations with sparse matrices algebra. Next two algorithms use Generalized Schur matrix decomposition and mimic algorithms employed by DYNARE and IRIS software.

For details on solvers please see accompanying documentation file 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 meaning that the first derivative of variables is constant at right boundary of computational domain. This is in a contrast to fixed boundary condition that can lead to solution disturbances 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 are programmed:

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)

Kalman filter requires to set initial conditions for filtered variables and its error variance-covariance matrix.
Two tables below show 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 Kalman filter error covariance matrix starting values algorithms:

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 user can select several Markov Chain Monte Carlo 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 provides a rich model setting that user can experiment with.

VII. Graphical Representation of Model Equations

Raising flag of *graph_info* parameter of run function of *driver* module produces a directional graph of model variables. An example of this graph for US potential output model is shown below. Green color ellipses mark endogenous variables nodes that appear on the left side of model equations. And the yellow color marks variables nodes that appear on the right side of these equations. The arrows show dependence of the left side variables on their counterparts in their equation expression. Equations of 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)

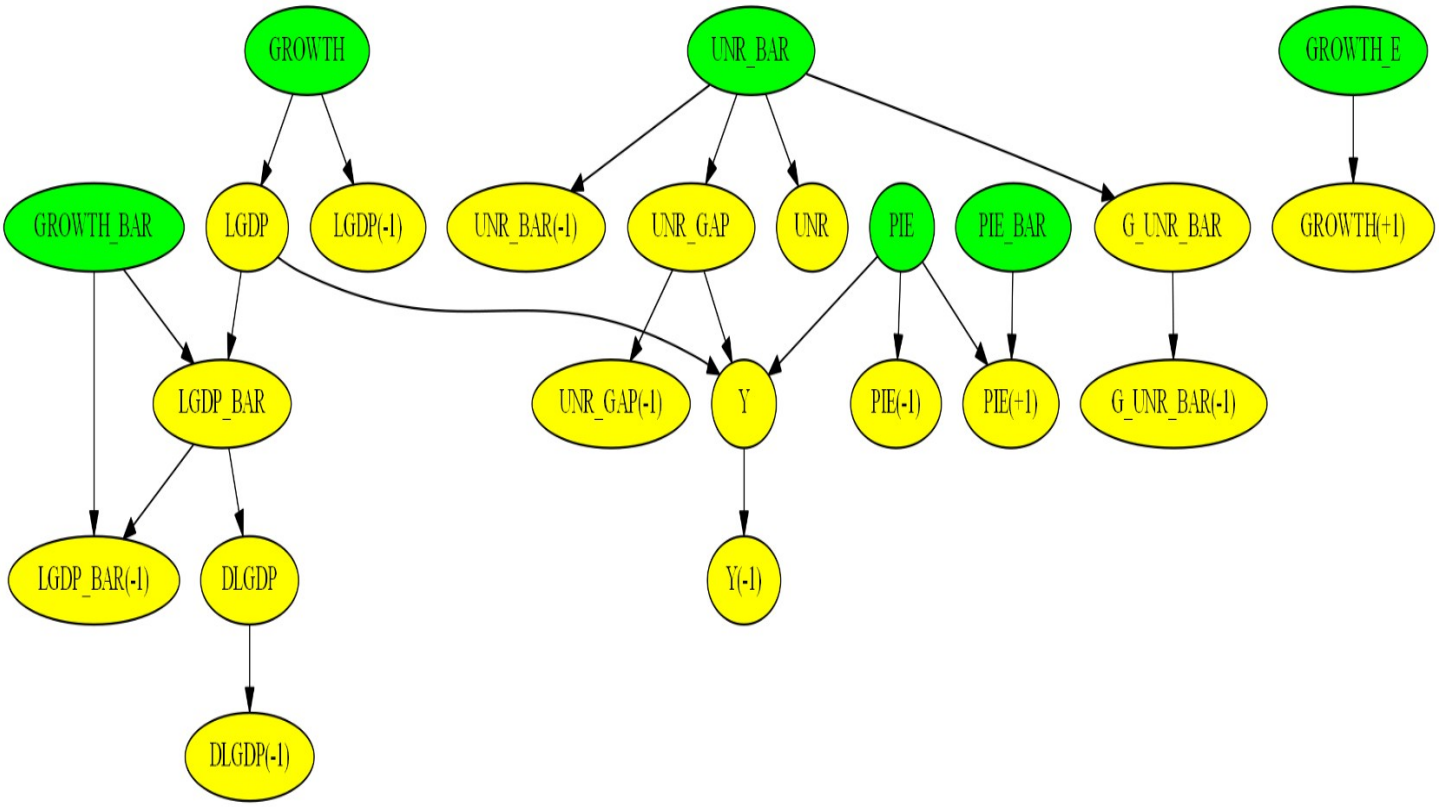


Fig.1. Graphical representation of model variables dependencies.

VIII. PDF REPORTS

Raising flag of *model_info* parameter in *run* function produces PDF report of model object. This PDF document contains several sections that describe model endogenous and exogenous variables, parameters, shocks, transient and measurement equations, as shown below:

Model File

Generated by Python Framework

June 13, 2024

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

1

IX. PROGRAMMING WITH FRAMEWORK

Creating a Model Object

Model can be instantiated by:

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

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

   This function parameters are:
   - path to 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 getModel() method:

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

   Function parameters include:
   - model name,
   - list of transition equations,
   - list of endogenous variables.,
   - list of model parameters,
   - list of shocks names,
   - calibration dictionary with starting values of endogenous variables and parameters.

Importing Model Files

Framework can read and parse model files developed by macroeconomic modelling software such as DYNARE, IRIS, TROLL, and SIRIUS. Below we show example of YAML model file which includes files describing endogenous and exogenous variables, parameters, and equations:

```
name:  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
```

Upon parsing this model file Framework generates Python model object. This translation is a work in progress and may need further development for complex model files.

Setting Starting Values

Starting values of endogenous variables can be set in a calibration section of model YAML file. They can also be read from a file containing historical data. Framework reads variables values at the start of simulation range. If endogenous variables have lags and leads, then the timing of these lags/leads is determined, and the corresponding values of these variables are used. These historic data overwrite starting values of model file. However, data file may not contain history for some of variables. These missing variables values can be determined by solving model steady-state equations and minimizing their residuals. This is achieved by raising flag of parameter, *bTreatMissingObs.* The excerpt of code below shows example of setting 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 calibration section of yaml model file
- Passing dictionary of parameters names and values when creating 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 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 fcalibration.txt file defining parameters that could be missed in 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. Example below sets "SHK_DLA_CPIE shock to values of 1, 5, and 3. Please note that index 0 corresponds to the starting time of 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 list of shocks. In the example below four shocks occur at the first quarter of 1998. The shock values are set to 1. Variables are shocked sequentially one by one in a loop and 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
    rng_date,yy,epsilonhat,etahat,s,rng,periods,model = run(model=model,irf=True)
```

Below we present plots of IRF functions.

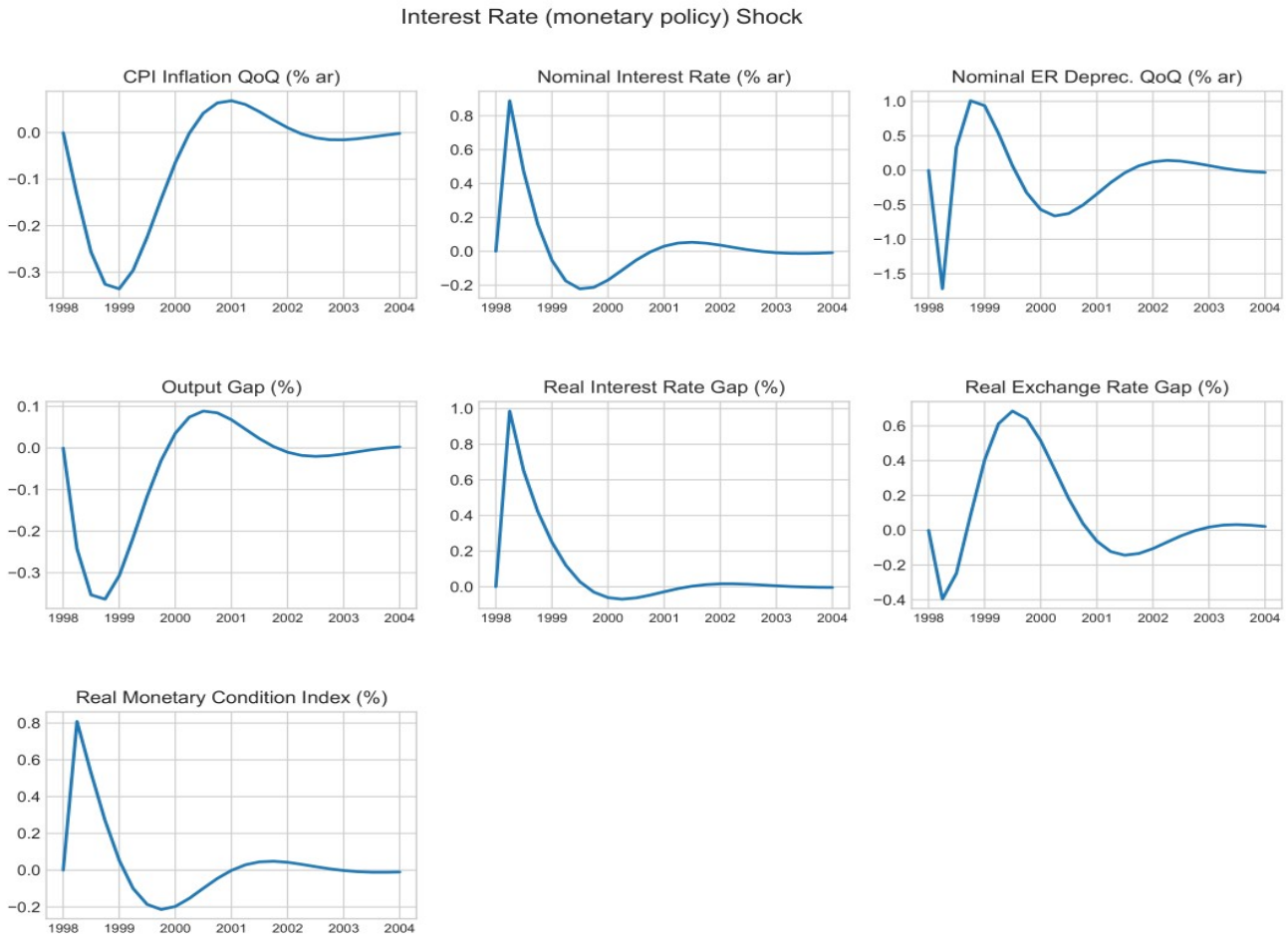


Fig.2. Impulse response functions of GAP model.

X. RUNNING SIMULATIONS

The **driver** module is a work horse of Framework. It serves many purposes including stochastic calculations and forecasts, sampling of parameters, and Kalman filtering and smoothing.

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

This function returns forecast dates and forecast results.

Parameters of this function are:

- model object,
- variable “Solver” which is a name of numerical method to solve the system of equations,
- list of output variables names to plot graphs,
- variable “shocks_file_path” which is a path to shock file,
- variable “steady_state_file_path” which is a path to steady-state file,

- variable “calibration_file_path” which is a path to calibration file or files.
- boolean flag “Plot” if raised will plots graphs,
- boolean flag “Output” if raised will output simulation results to excel file,
- string output_dir is the path to output folder where rub results will be saved.
  If this parameter is not set, then Platform will query 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 are set, then output directory will be set to the parent of “src” folder.

Call to Kalman filter function additionally returns filtered and smoothed shocks:

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

Parameters of this function are:

- variable “meas” which is a path to a file with measurement data,
- variable “Prior” which is used to set initial values of error covariance matrix for Kalman Filter,
- variable “Filter” which is a name of Kalman filter algorithm,
- variable “Smoother” which is a name of Kalman filter smoother algorithm.

And finally, estimate(model=model) function estimates model parameters and runs MCMC sampling.

Test Program

Tests folder contains modules with run examples of number of models. Each module requires user to provide a path to a model file. User may also specify which output variables she/he wants to output or plot. If “output_variables” are not set, then all variables will be outputted or plotted. By default, results will be stored in excel files or in python sqlite database in data folder. Plots will be saved in graphs folder. Below we show 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...
rng_date,yy = 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 produces graphs shown below.

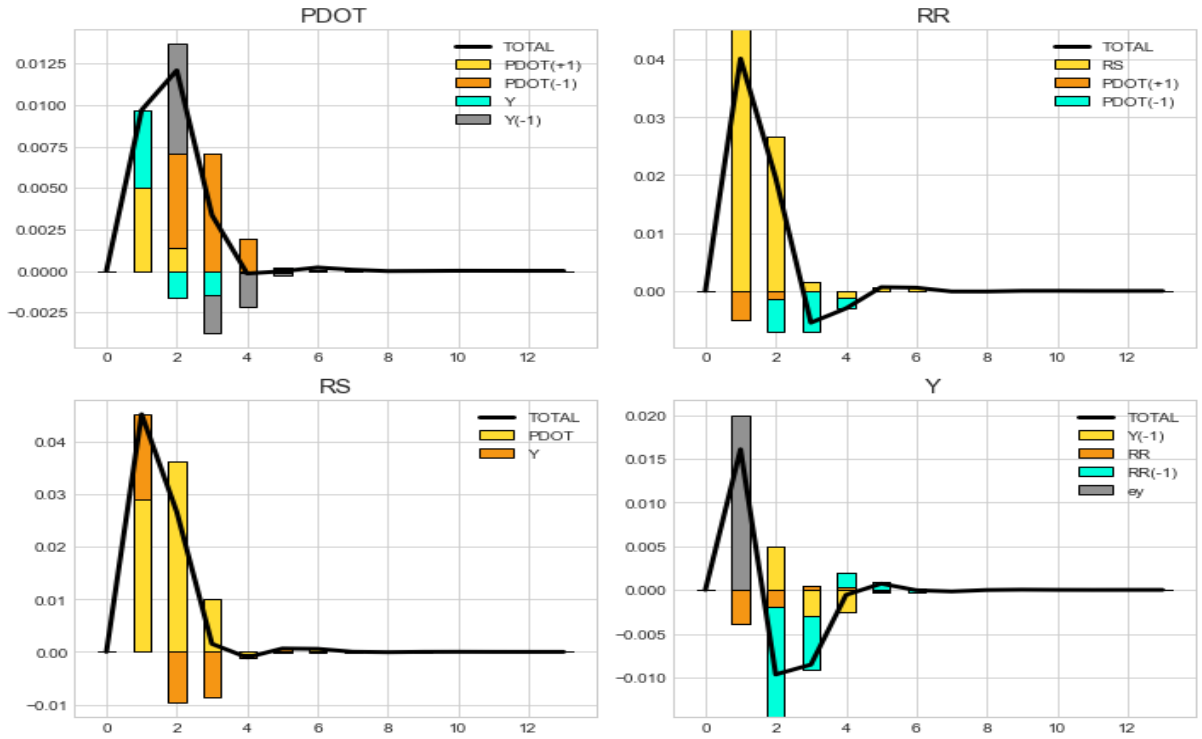


Fig.3. IRFs of inflation (PDOT), real (RR) and nominal (RS) interest rates, and output (Y). Output variable (Y) is shocked at period 1 by 2%.

Below we present model file and results of forecast of simple Real Business Cycle (RBC) model. Shocks to output (Y) and total factor productivity (A) are stochastic and are described by multivariate normal distribution. The mean and covariance matrix are set in "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.

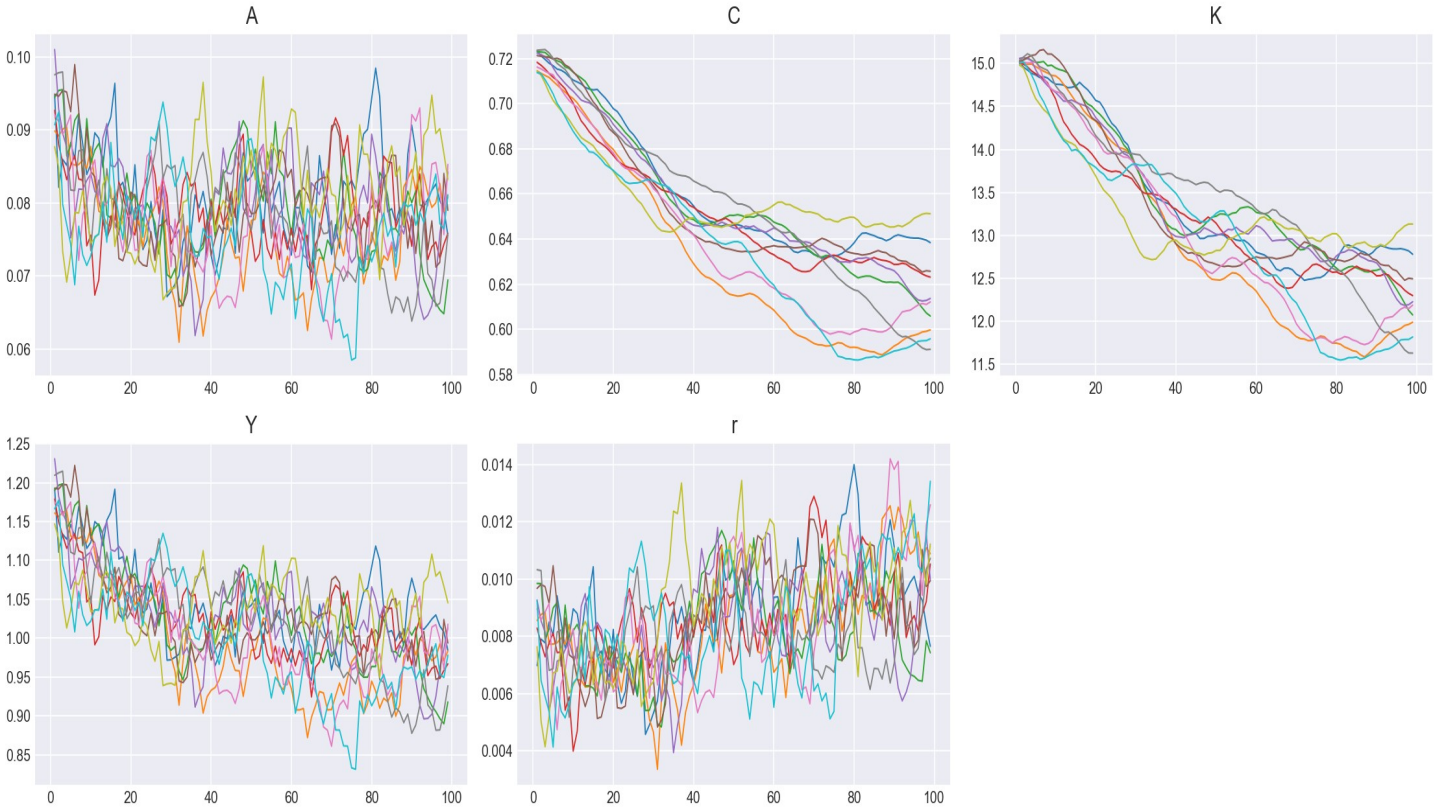


Fig.4. Ten realization paths of macroeconomic variables. Here A is the total factor productivity, C is the consumption, K is the capital, Y is the output, and r is the interest rate.

Kalman Filter and Smoother

Next example illustrates test of US potential output model. Here user specifies a path to model file, path to measurement data, and path to results output file. The importModel function parses this model file and constructs model object which is then passed as a parameter to 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']
```

```
# 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
rng_date,yy,epsilonhat,etahat = kalman_filter(model=model,meas=meas,fout=fout,Output=False,Plot=True)
```

Call to Kalman filter function returns list of dates (*rng_date)*, results of Kalman filter and smoother wrapped in list (*yy*) and filtered (*epsilonhat*) and smoothed shocks (*etaha*). Observation variables are read from measurement data file. They can also be sourced from HAVER, ECOS, EDI and World Bank databases. Example below shows how to source GDP, CPI index, growth rate and unemployment observation variables from HAVER database. Here 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 run of this test produces plots below. The solid blue lines show filtered endogenous variables, the green and yellow lines show results of applying LRX and HP filters, and the dots show the data points. By default, RX and HP filters are applied to measurement variables with names that end in "_BAR".

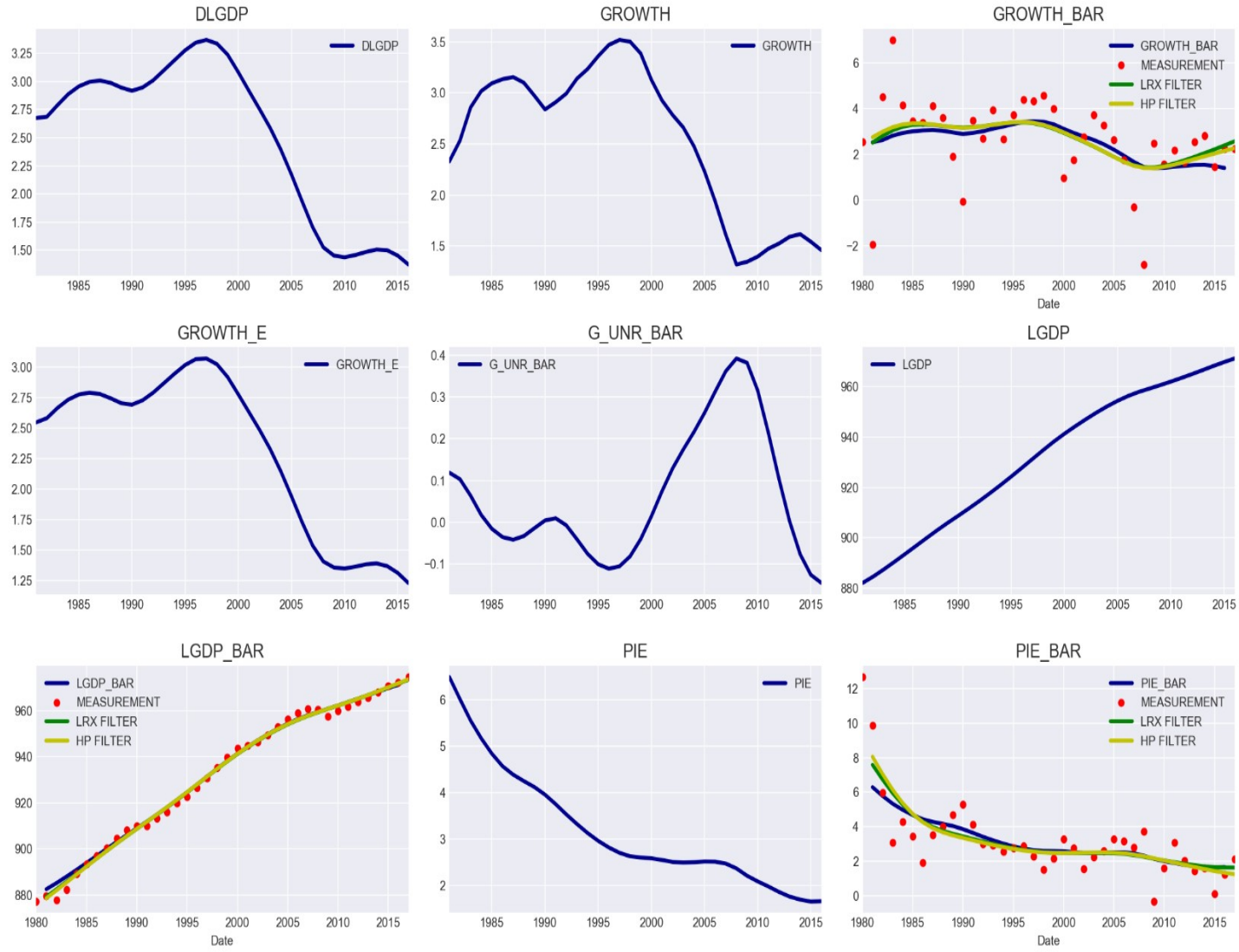


Fig.5. HP, LRX, and Kalman filter produce similar results for potential output model.

Estimating Model Parameters

User can estimate model parameters given measurement data. This is achieved by finding parameters that maximize likelihood of model fit. One can choose all or a subset of parameters by selecting initial values and lower and upper bounds of 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 “estimate” is raised in “estimate” function. Sampling of parameters can be accomplished by passing parameter “sample” equal to true. Markov Chain Monte Carlo methods are applied to sample model parameters from probability distributions. The names of four parameters above are followed by the staring values, lower and upper bounds, and type of probability density function distribution, and parameters means and standard deviations. Example of parameters sampling for 300 draws is shown below.

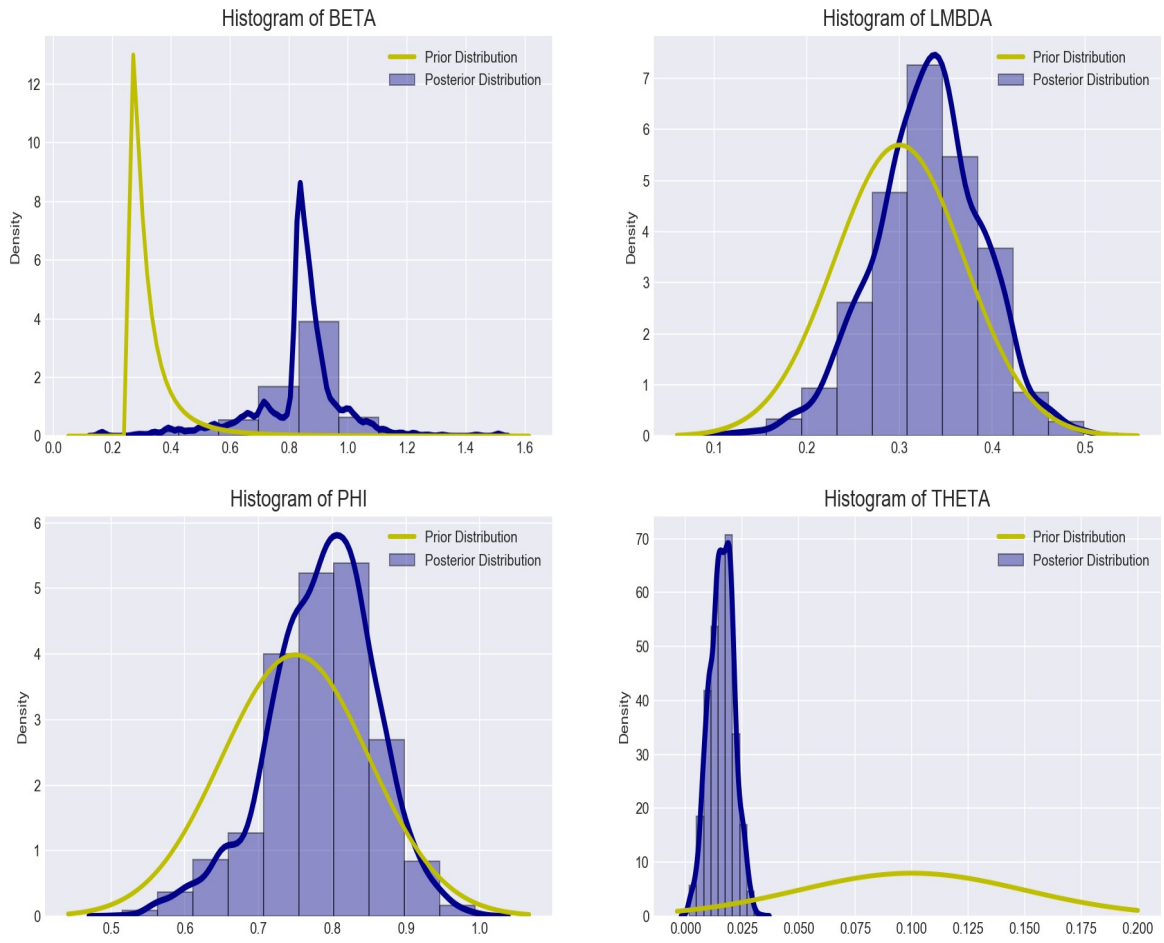


Fig.6. Distribution of US MVF potential output model. Yellow color lines show prior distribution, and blue lines – posterior distribution.

These sampling algorithms use sum of prior and posterior logarithms of probabilities. The number of draws is controlled by parameter “Ndraws” of estimate function.

Two dimensional projections of multidimensional samples covariances are presented below. The blue vertical and horizontal lines display mean of samples, while the red lines show the optimal values of parameters.

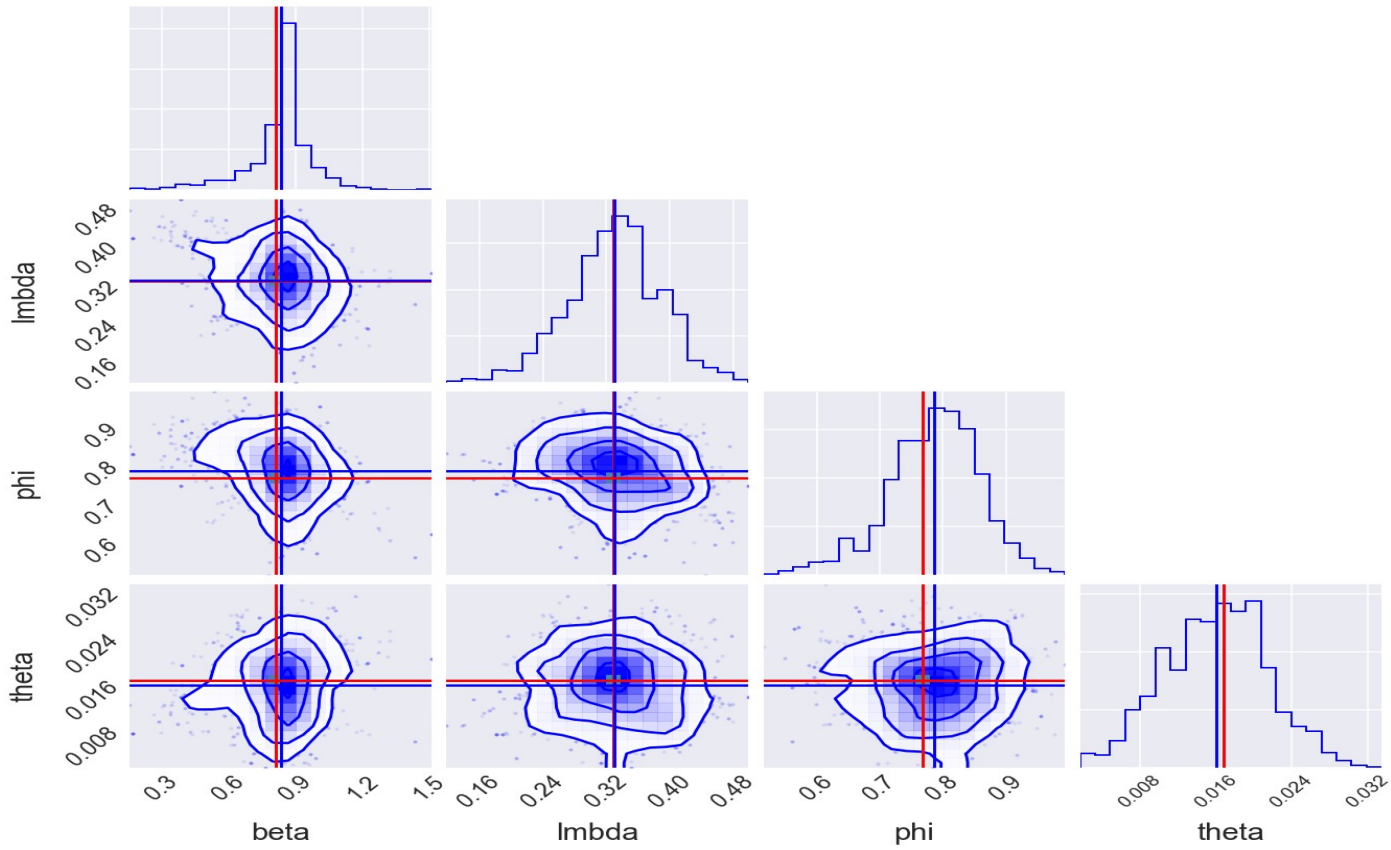


Fig.7. Two dimensional projections of model parameters.

Judgmental Adjustments

User may have a specific view on future endogenous variables path. This can be programed by "exogenizing" endogenous variables and "endogenizing" exogenous shock variables. Example below shows judgmental adjustments to nominal interest rate "RS". Framework finds shock values "SHK_RS" that bring path of "RS" to the desired level. This shock is endogenized by calling model 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. It shows imposition of "soft" and "hard" tunes on macroeconomic variables. Seven scenarios include no tunes, soft tunes, hard tunes, 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)
```

```
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
rng_date,yy1 = 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)
rng_date,yy2 = 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)
rng_date,yy3 = 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)
rng_date,yy4 = run(model=model)

### 5. Anticipated soft tune
d = {"SHK_L_S": [(3,10*i)]}
model.setShocks(d)
rng_date,yy5 = 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)
rng_date,yy6 = 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)
rng_date,yy7 = run(model=model)

This setup produces seven scenarios' results shown below.

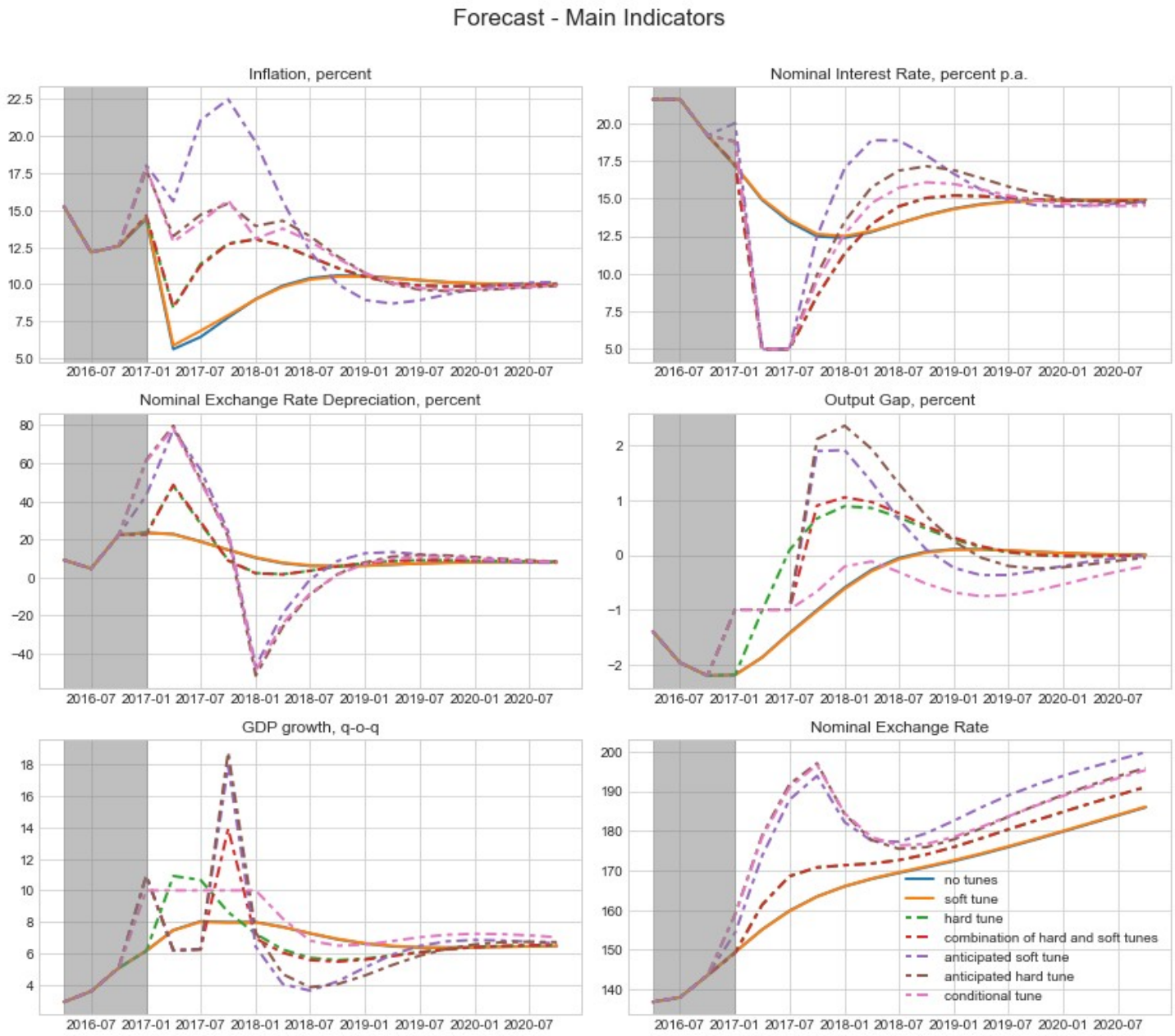


Fig.7. Forecast of macroeconomic variables for different sets of user judgments on future path of these variables.

XI. Forecasting Economic Impact of COVID-19 Pandemic

Lastly, we illustrate application of Python Platform to forecast impact of COVID-19 virus. We use model of Eichenbaum, Rebelo and Trabandt (ERT) that integrates New Keynes framework with sticky prices and wages with epidemiological Susceptible-Infected-Recovered (SIR) model of virus transmission. Original Omicron virus strain that originated in December 2020 was followed half a year later by Delta strain, which was more contagious and more aggressive. While

Omicron caused massive lockdown measures adopted by the US government, and large recession of economic activity, the Delta strain impact was much milder. Because of that we only accounted economic impact of the first strain. 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*

 # 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.

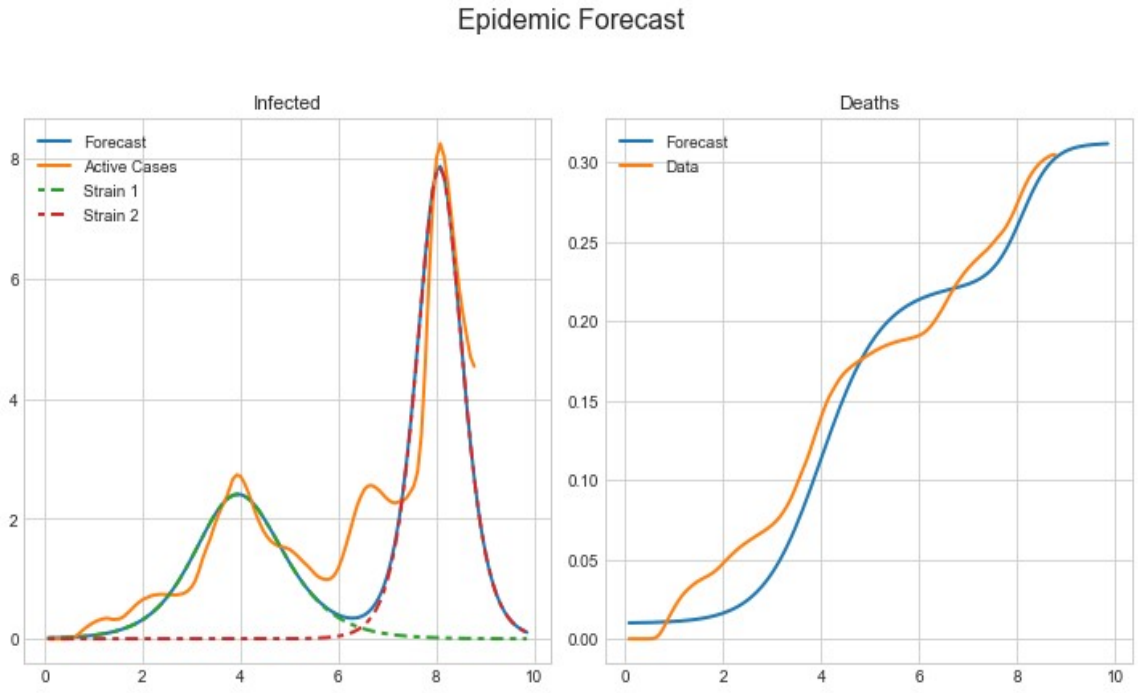


Fig.10. Impact of Omicron and Delta virus strain on number of infected people and number of deaths. Vertical axes show population percentage. Green color and red color lines show transmission of the first and the second virus strains. The orange color lines show the actual data.

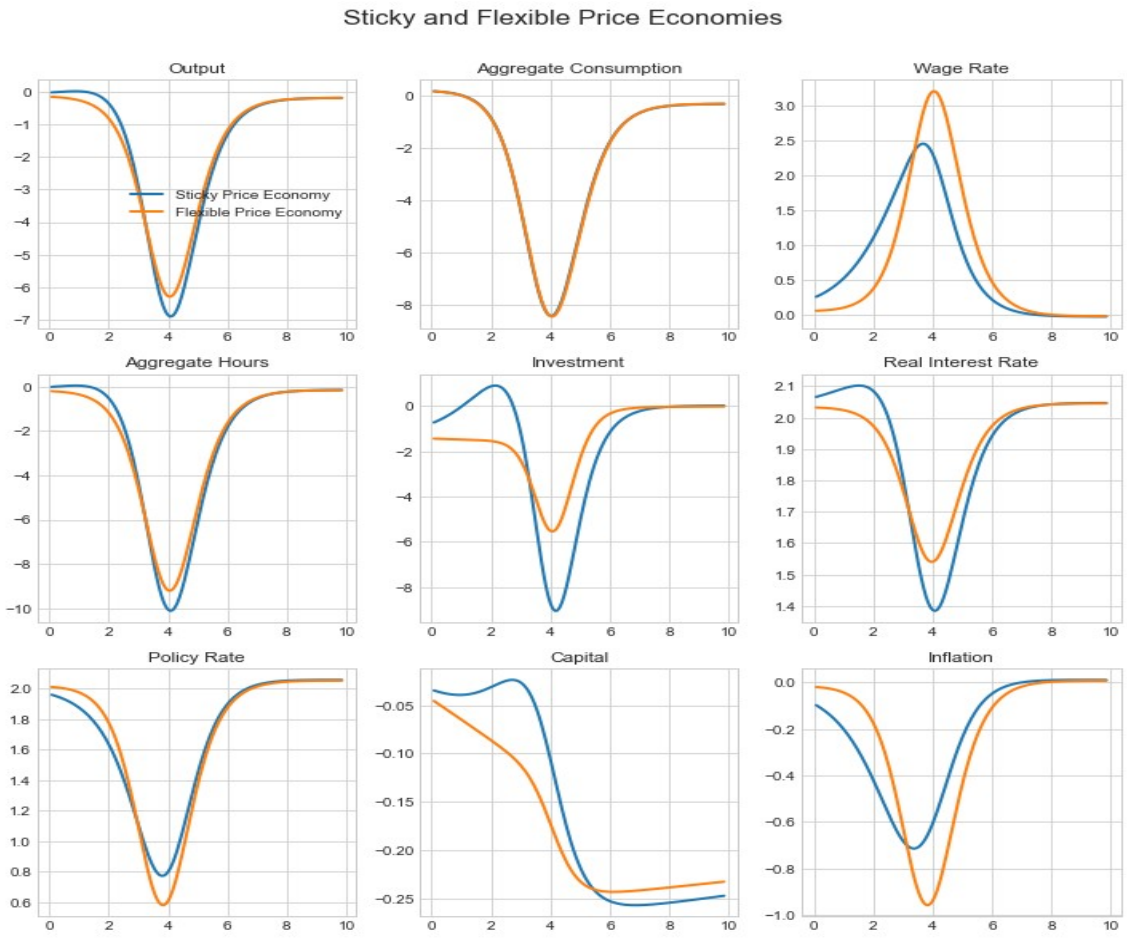


Fig.11. Detrimental effects of the first COVD-19 virus strain on economic activity.

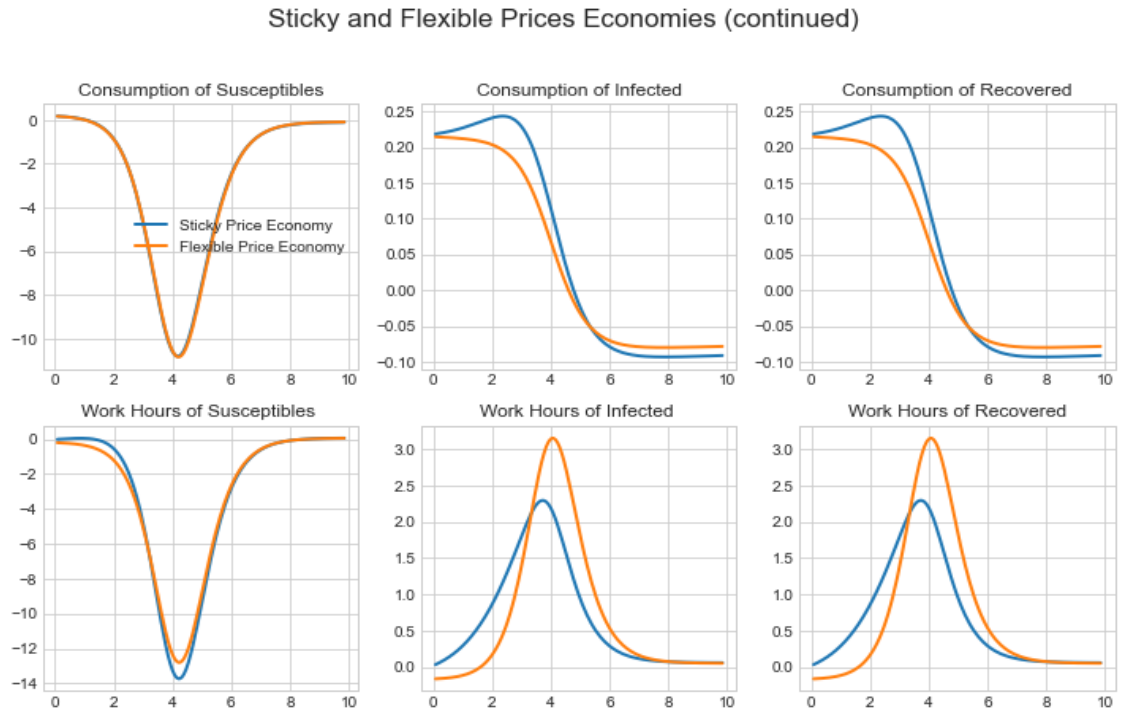


Fig.12. COVD-19 pathogen causes a deep reduction in consumption of susceptible, infected, and recovered individuals. It causes large reduction of working hours of susceptible population and minor increase of infected and recovered.

XII. EXAMPLES

Below we present four examples of model files of simple monetary model, Kalman filter, optimization, and estimation of Peter Ireland 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...
   rng_date,yy = \
   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
    rng_date,yy,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)

if __name__ == '__main__':
```

```
    """ The main test program. """
    test()
```

Optimization Example

```
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

```
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
   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"]
```

XIII. APPENDICES

Graphical User Interface

To facilitate model development, we created a graphical user interface which allows to enter model equations, endogenous and exogenous variables, parameters, and shocks.

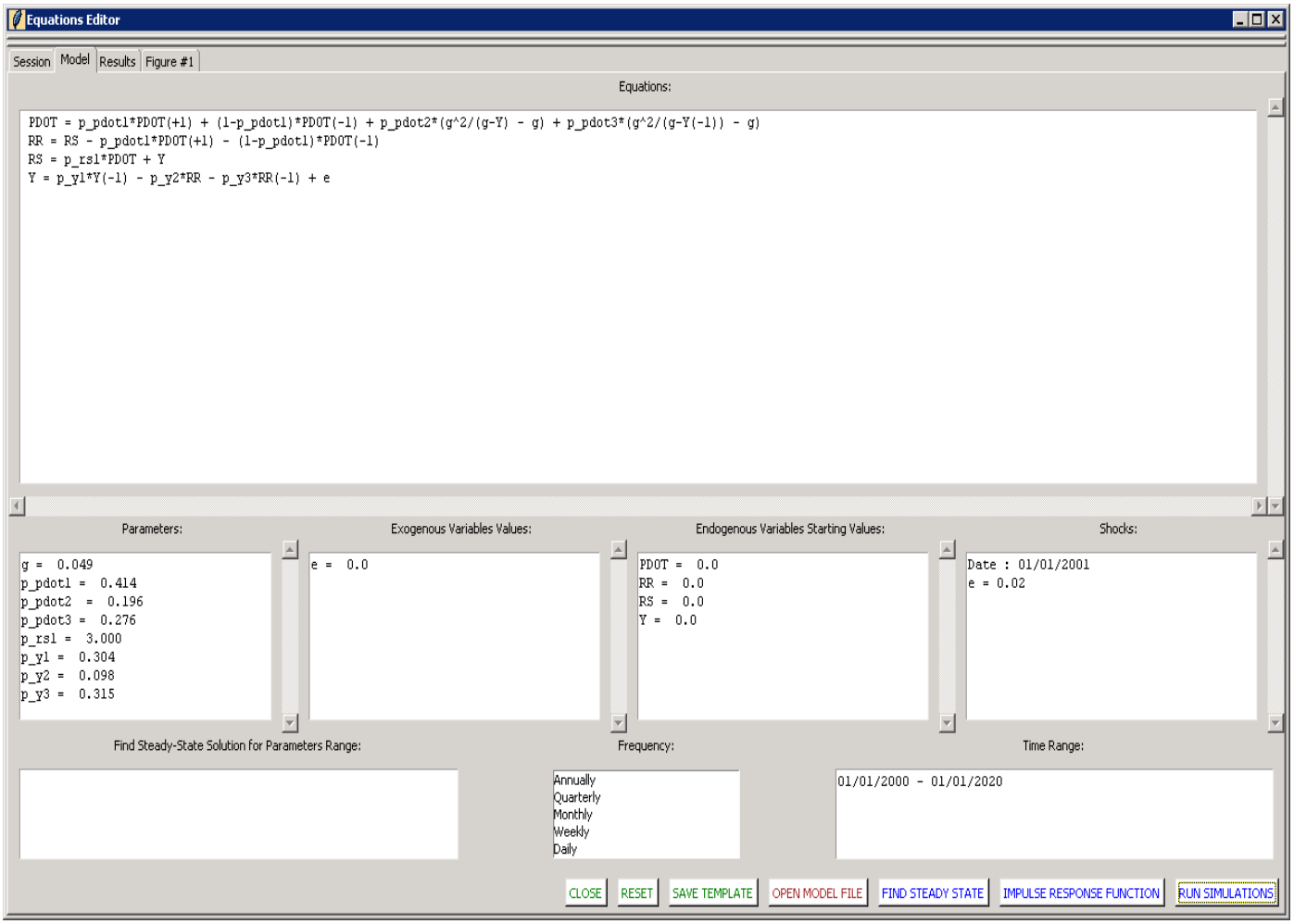
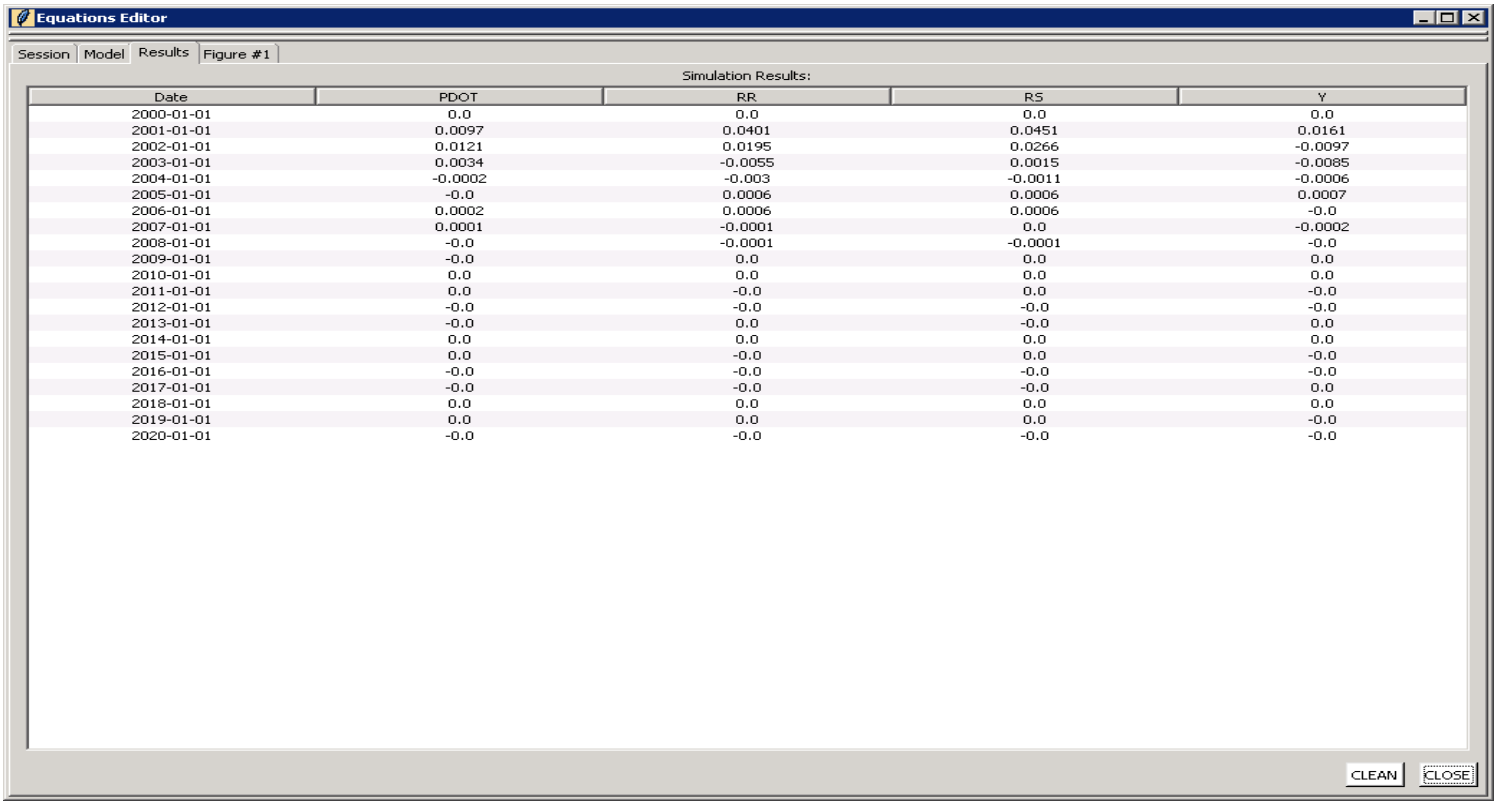


Fig.13. Editor facilitates users to enter model equations, names of variables, parameters and shocks and run forecasts.

When user runs simulations, the model specifications are saved into temporary YAML model file for further analysis. The results of this run are presented in tabular form in the Results tab.



Date	PDOT	RR	RS	Y
2000-01-01	0.0	0.0	0.0	0.0
2001-01-01	0.0097	0.0401	0.0451	0.0161
2002-01-01	0.0121	0.0195	0.0266	-0.0097
2003-01-01	0.0034	-0.0055	0.0015	-0.0085
2004-01-01	-0.0002	-0.003	-0.0011	-0.0006
2005-01-01	-0.0	0.0006	0.0006	0.0007
2006-01-01	0.0002	0.0006	0.0006	-0.0
2007-01-01	0.0001	-0.0001	0.0	-0.0002
2008-01-01	-0.0	-0.0001	-0.0001	-0.0
2009-01-01	-0.0	0.0	0.0	0.0
2010-01-01	0.0	0.0	0.0	0.0
2011-01-01	0.0	-0.0	0.0	-0.0
2012-01-01	-0.0	-0.0	-0.0	-0.0
2013-01-01	-0.0	0.0	-0.0	0.0
2014-01-01	0.0	0.0	0.0	0.0
2015-01-01	0.0	-0.0	0.0	-0.0
2016-01-01	-0.0	-0.0	-0.0	-0.0
2017-01-01	-0.0	-0.0	-0.0	0.0
2018-01-01	0.0	0.0	0.0	0.0
2019-01-01	0.0	0.0	0.0	-0.0
2020-01-01	-0.0	-0.0	-0.0	-0.0

Fig.14. Tabular presentation of macroeconomic variables.

And plots are displayed a Figure tab.

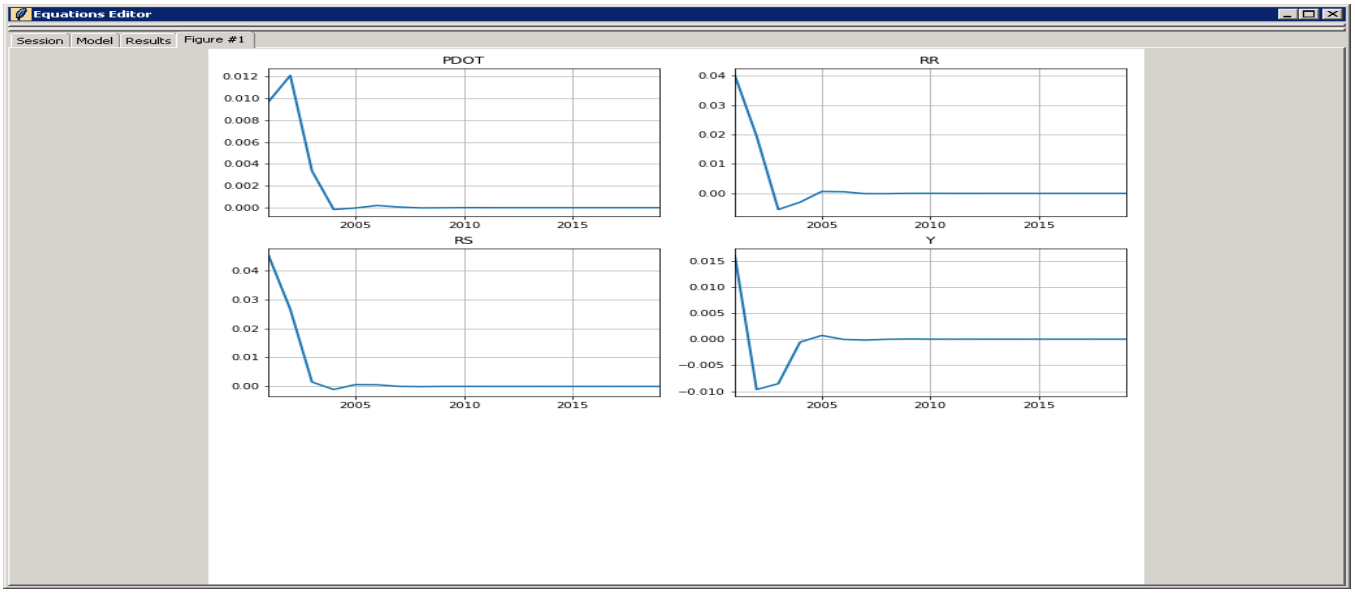


Fig.15. Graphs of models' variables.