The World Bank's MFMod Framework in Python with Modelflow

Andrew Burns and Ib Hansen

Apr 27, 2023

CONTENTS

Foreword

Lorem Ipsum “Neque porro quisquam est qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit…” “There is no one who loves pain itself, who seeks after it and wants to have it, simply because it is pain…”

freestar

freestar Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vestibulum aliquam varius mi. Suspendisse pharetra egestas viverra. Aenean viverra hendrerit sagittis. Curabitur vel lectus at arcu mattis blandit. Quisque aliquet erat nunc, vitae consequat eros venenatis eu. Vivamus ut arcu eget ipsum mollis iaculis. Aliquam rhoncus bibendum orci. Donec lacinia, mauris placerat auctor vehicula, odio eros efficitur leo, et porttitor est urna vitae erat. Cras tempor nec purus at tincidunt. Maecenas viverra massa diam, sit amet tristique mi scelerisque non. Etiam scelerisque, risus ac mollis hendrerit, ex velit vehicula tortor, quis accumsan leo enim sed leo. Suspendisse potenti. Nulla libero diam, eleifend nec sollicitudin ut, varius non eros.

Ut sit amet mollis ipsum. Donec tempor magna ac blandit gravida. Phasellus viverra, arcu at euismod auctor, lectus justo vehicula eros, sit amet posuere felis mi ac purus. Nullam gravida lacinia bibendum. Vivamus ultrices justo sed aliquam feugiat. Mauris vulputate sapien in tempus posuere. Morbi nec purus eget ipsum fermentum congue. Vivamus auctor, mi sit amet lacinia suscipit, ipsum lectus pulvinar risus, non condimentum eros felis sed quam. Pellentesque consectetur leo sit amet condimentum commodo. Orci varius natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Duis risus mi, elementum ac leo ut, ultrices scelerisque dui.

Sed quis arcu et dui viverra interdum vitae id enim. In euismod diam quis eleifend viverra. Nullam sodales dictum turpis, vestibulum sodales erat. Morbi quis orci dictum mauris volutpat porttitor at at sapien. Maecenas nec metus ut felis malesuada dapibus. Duis semper lacus eget hendrerit congue. Aenean condimentum, ligula ac sagittis rutrum, turpis elit pulvinar libero, eget tristique sapien sem eget lacus. Curabitur egestas velit quis eros volutpat rhoncus. Nunc quam nibh, commodo ac egestas non, tristique sit amet nisl. Ut vitae lacinia justo.

Indermit Gil World Bank Chief Economist

Part I

The World Bank's MFMod Framework and Modelflow

CHAPTER ONE

INTRODUCTION

Warning: This Jupyter Book is work in progress.

This paper describes the implementation of the World Bank's MacroFiscalModel (MFMod, see Burns *et al.* [2019]) using the open source solution program ModelFlow (Hansen, 2023).

The impetus for this paper and the work that it summarizes was to make available to a wider constituency the work that the Bank has done over the past several decades to disseminate Macro-structural models[1] – notably those that form part of its MFMod (MacroFiscalModel) framework.

1.1 The MFMod Framework at the World Bank

MFMod is the World Bank's work-horse macro-structural economic modelling framework. It exists both a linked system of 184 country specific models that can be solved either independently or as a larger system (`MFMod`), and as a series of standalone customized models, known collectively as MFMod Standalones (MFMod SAs) that have been developed from the central model to the fit the specific needs of individual countries. Both modelling systems can be solved using the EViews modelling languiage, or through the intermediation of an easy-to-use excel front end developed by the Bank.

The main `MFMod` global model evolved from earlier macro-structural models developed during the 2000s to strengthen the basis for the forecasts produced by the World Bank. Some examples of these models were released on the World Bank's isimulate platform early in 2010 along with several CGE models dating from this period. These earlier models were substantially extended into what has become the main MFMod (MacroFiscalModel) model during 2014. Since 2015, MFMod replaced the Bank's RMSIM-X model ([Addison, 1989]), as the Bank's main tool for forecasting and economic analysis, and is used for the World Bank's twice annual forecasting exercise The Macro Poverty Outlook.

The main documentation for `MFMod` are Burns *et al.* [2019].

[1] Economic modelling has a long tradition at the World Bank. The Bank has had a long-standing involvement in macroeconomic modelling, initally with linear programming polanning models [Chenery, 1971], and then CGE models []. Indeed, the popular modelling package GAMS, which is widely used to solve CGE and Linear Programming models, started out as a project begun at the World Bank in the 1976 [Addison, 1989].

1.1.1 Climate aware version of MFMod

Most recently, the Bank has extended the standard MFMod framework to incorporate the main features of climate change [Burns *et al.*, 2021]– both in terms of the impact of the economy on climate (principally through green-house gas emissions, like $CO_2, N_2O, CH_4, ...$) and the impact of the changing climate on the economy (higher temperatures, changes in rainfall quantity and variability, increased incidence of extreme weather) and their impacts on the economy (agricultural output, labor productivity, physical damages due to extreme weather events, sea-level rises etc.).

Variants on the model initially described in Burns *et al.* [2021], have been developed for [xx] countries and underpin the economic analysis contained in many of the World Bank's Country Climate Development Reports.

1.2 Early steps to bring the MFMod system to the broader economics community

Bank staff were quick to recognize that the models built for its own needs could be of use to the broader economics community. An initial project `isimulate` made several versions of this earlier model available for simulation on the isimulate platform in 2007, and these models continue to be available there. The `isimulate` platform housed (and continues to house) public access to earlier versions of the MFMod system, and allows simulation of these and other models – but does not give researchers access to the code or the ability to construct complex simulations.

In another effort to make models widely available a large number (more than 60 as of June 2023) customized stand-alone models (collectively known as called MFModSA - MacroFiscalModel StandAlones) have been developed from the main model. Typically developed for a country-client (Ministry of Finance, Economy or Planning or Central Bank), these Stand Alones extend the standard model by incorporating additional details not in the standard model that are of specific import to different economies and the country-clients for whom they were built, including: a more detailed breakdown of the sectoral make up of an economy, more detailed fiscal and monetary accounts, and other economically important features of the economy that may exist only inside the aggregates of the standard model.

Training and dissemination around these customized versions of MFMod have been ongoing since 2013. In addition to making customized models available to client governments, Bank teams have run technical assistance program designed to train government officials in the use of these models, their maintenance, modification and revision.

1.3 Moving the framework to an open-source footing

Models in the MFMod family are normally built using the proprietary EViews econometric and modelling package. While offering many advantages for model development and maintenance, its cost may be a barrier to clients in developing countries. As a result, the World Bank joined with Ib Hansen, a Danish economist formerly with the European Central Bank and the Danish Central Bank, who over the years has developed `modelflow` a generalized solution engine written in Python for economic models. Together with World Bank, Hansen has worked to extend `modelflow` so that MFMod models can be ported and run in the framework.

This paper reports on the results of these efforts. In particular, it provides step by step instructions on how to install the `modelflow` framework, import a World Bank macrostructural model, perform simulations with that model and report results using the many analytical and reporting tools that have been built into `modelflow`. It is not a manual for `modelflow`, such a manual can be found here nor is it documentation for the MFMod system, such documentation can be found here [Burns *et al.*, 2019] and here [Burns *et al.*, 2021], [Burns *et al.*, 2021]). Nor is it documentation for the specific models described and worked with below.

1.4 Macrostructural models

The economics profession uses a wide range of models for different purposes. Macro-structural models (also known as semi-structural or Macro-econometric models) are a class of models that seek to summarize the most important interconnections and determinants of economic activity in an economy. Computable General Equilibrium (CGE), and Dynamic Stochastic General Equilibrium (DSGE) models are other classes of models that also seek, using somewhat different methodologies, to capture the main economic channels by which the actions of agents (firms, households, governments) interact and help determine the structure, level and rate of growth of economic activity in an economy.

Olivier Blanchard, former Chief Economist at the International Monetary Fund, in a series of articles published between 2016 and 2018 that were summarized in Blanchard [2018], lays out his views on the relative strengths and weaknesses of each of these systems, concluding that each has a role to play in helping economists analyze the macro-economy. Typically, organizations, including the World Bank, use all of these tools, privileging one or the other for specific purposes. Macrostructural models like the MFMod framework are widely used by Central Banks, Ministries of Finance; and professional forecasters both for the purposes of generating forecasts and policy analysis.

1.4.1 A system of equations

Mathematically, macro-structural models are a system of equations comprised of two kinds of equations and three kinds of variables.

Types of variables in macro-structural models

- `Identities` are variables that are determined by a well defined accounting rule that always holds. The famous GDP Identity Y=C+I+G+(X-M) is one such identity, that indicates that GDP at market prices is definitionally equal to Consumption plus Investment plus Government spending plus Exports less Imports.
- `Behavioural` variables are determined by equations that typically attempt to summarize an economic (vs accounting) relationship. Thus, the equation that says Real Consumption = f(Disposable Income,the price level, and animal spirits) is a behavioural equation – where the relationship is drawn from economic theory. Because these equations do not fully explain the variation in the dependent variable and the sensitivities of variables to the changes in other variables are uncertain, these equations and their parameters are typically estimated econometrically and are subject to error.
- `Exogenous` variables are not determined by the model. Typically there are set either by assumption or from data external to the model. For an individual country model, the exogenous variables would often include the global price of crude oil because the level of activity of the economy itself is unlikely to affect the world price of oil.

In a fully general form it can be written as:

$$
\begin{aligned}
y_t^1 &= f^1(y_{t+u}^1 ..., y_{t+u}^n ..., y_t^2 ..., y_t^n ... y_{t-r}^1 ..., y_{t-r}^n, x_t^1 ... x_t^k, ... x_{t-s}^1 ..., x_{t-s}^k) \\
y_t^2 &= f^2(y_{t+u}^1 ..., y_{t+u}^n ..., y_t^1 ..., y_t^n ... y_{t-r}^1 ..., y_{t-r}^n, x_t^1 ... x_t^k, ... x_{t-s}^1 ..., x_{t-s}^k) \\
&\vdots \\
y_t^n &= f^n(y_{t+u}^1 ..., y_{t+u}^n ..., y_t^1 ..., y_t^{n-1} ... y_{t-r}^1 ..., y_{t-r}^n, x_t^1 ... x_t^r, x ..._{t-s}^1 ..., x_{t-s}^k)
\end{aligned}
$$

where y_t^1 is one of n endogenous variables and x_t^1 is an exogenous variable and there are as many equations as there are unknown (endogenous variables).

Substituting the variable mnemonics Y,C,I,G,X,M for the simple model the above can be rewritten as as a system of 6 equations in 6 unknowns:

$$
\begin{aligned}
Y_t &= C_t + I_t + G + t + (X_t - M_t) \\
C_t &= c_t(C_{t-1}, C_{t-2}, I_t, G_t, X_t, M_t, P_t) \\
I_t &= c_t(I_{t-1}, I_{t-2}, C_t, G_t, X_t, M_t, P_t) \\
G_t &= c_t(G_{t-1}, G_{t-2}, C_t, I_t, X_t, M_t, P_t) \\
X_t &= c_t(X_{t-1}, X_{t-2}, C_t, I_t, G_t, M_t, P_t, P^f_t) \\
M_t &= c_t(M_{t-1}, M_{t-2}, C_t, I_t, G_t, X_t, P_t, P^f_t)
\end{aligned}
$$

and where P_t, P^f_t (domestic and foreign prices, respectively) are exogenous in this simple model.

CHAPTER

TWO

MODELFLOW AND THE MFMOD MODELS OF THE WORLD BANK

At the World Bank models built using the MFMod framework are developed in EViews. When disseminated to clients, the models are operated in a World Bank customized EViews environment. But as a systems of equations and associated data the models can be solved, and operated under any system capable of solving a system of simultaneous equations – as long as the equations and data can be transferred from EViews to the secondary system. `Modelflow` is such a system and offers a wide range of features that permit not only solving the model, but also provide a rich and powerful suite of tools for analyzing the model and reporting results.

2.1 A brief history of ModelFlow

Modelflow is a python library that was developed by Ib Hansen over several years while working at the Danish Central Bank and the European Central Bank. The framework has been used both to port the U.S. Federal Reserve's macro-structural model to python, but also been used to bring several stress-testing models developed by European Central Banks and the European Central Bank into a python environment.

Beginning in 2019, Hansen has worked with the World Bank to develop additional features that facilitate working with models built using the Bank's MFMod Framework, with the objective of creating an open source platform through which the Bank's models can be made available to the public.

This paper, and the models that accompany it, are the product of this collaboration.

2.2 Installation of Modelflow

Modelflow is a python package that defines the `model` class, its methods and a number of other functions that extend and combine pre-existing python functions to allow the easy solution of complex systems of equations including macro-structural models like MFMod. To work with `modelflow`, a user needs to first install python (preferably the Anaconda variant), several supporting packages, and of course the `modelflow` package itself. While `modelflow` can be run directly from the python command-line or IDEs (Interactive Development Environments) like `Spyder` or Microsoft's `Visual Code`, it is suggested that users also install the Jupyter notebook system. Jupyter Notebook facilitates an interactive approach to building python programs, annotating them and ultimately doing simulations using MFMod under `modelflow`. This entire manual and the examples in it were all written and executed in the Jupyter Notebook environment.

2.2.1 Installation of Python

`Python` is an extremely powerful, versatile and extensible open-source language. It is widely used for artificial intelligence application, interactive web sites, and scientific processing. As of 14 November 2022, the `Python Package Index` (PyPI), the official repository for third-party Python software, contained over 415,000 packages that extend its functionality[1]. Modelflow is one of these packages.

Python comes in many flavors and `modelflow` will work with any of them. However, **users are strongly advised to use the Anaconda version of Python**.

The remainder of this section points to instructions on how to install the Anaconda version of python (under Windows, MacOS and under Linux). `Modelflow` works equally well under all three. This is followed by section that describes the steps necessary to create an anaconda environment with all the necessary packages to run `modelflow`.

Installation of Anaconda under Windows

The definitive source for installing Anaconda under windows can be found here.

> **Warning: It is strongly advised that Anaconda be installed for a single user (Just Me)** This is much easier to maintain over time. Installing "For all users on this computer" the other option offered by the `anaconda` installer will substantially increase the complexity of maintaining python on your computer.

Installation of Python under macOS

The definitive source for installing Anaconda under macOS can be found here.

Installation of Python under Linux

The definitive source for installing Anaconda under Linux can be found here.

2.3 Installation of Modelflow

`Modelflow` is a python package that defines the modelflow class `model` among others. `Modelflow` has many dependencies. Installing the class the first time can take some time depending on your internet connection and computer speed. It is essential that you follow all of the steps outlined below to ensure that your version of `modelflow` operates as expected.

> **Warning:** The following instructions concern the installation of `modelflow` within an Anaconda installation of python. Different flavors of Python may require slight changes to this recipe, but are not covered here.
>
> `Modelflow` is built and tested using the anaconda python environment. It is strongly recommended to use Anaconda with `modelflow`.
>
> If you have not already installed Anaconda following the instructions in the preceding section, please do so **Now**.

[1] Wikipedia article on python

2.3.1 Installation of `modelflow` under Anaconda

1. Open the anaconda command prompt
2. Execute the following commands by copying and pasting them – either line by line or as a single mult-line step
3. Press enter

```
conda create -n ModelFlow -c ibh -c  conda-forge modelflow_pinned_developement_test -y
conda activate ModelFlow
pip install dash_interactive_graphviz
conda install pyeviews -c conda-forge -y
jupyter contrib nbextension install --user
jupyter nbextension enable hide_input_all/main
jupyter nbextension enable splitcell/splitcellcd
jupyter nbextension enable toc2/main
```

Depending on the speed of your computer and of your internet connection installation could take as little as 10 minutes or more than 1/2 an hour.

At the end of the process you will have a new conda environment called `modelflow`, and this will have been activated. The computer set up is complete and the user is ready to work with `modelflow`.

The following sections give a brief introduction to Jupyter notebook, which is a flexible tool that allows us to execute python code, interact with the modelflow class and World Bank Models and annotate what we have done for future replication.

---

note

Once installed, a `modelflow` environment can be updated by activating the `modelflow` environment created above – `conda activate modelflow` – and then executing the command: `conda install modelflow -c ibh --no-deps`.

Part II

Some python essentials for using WorldBank models with modelflow

CHAPTER

THREE

INTRODUCTION TO JUPYTER NOTEBOOK

Jupyter Notebook is a web application for creating, annotating, simulating and working with computational documents. Originally developed for python, the latest versions of EViews also support Jupyter Notebooks. Jupyter Notebook offers a simple, streamlined, document-centric experience and can be a great environment for documenting the work you are doing, and trying alternative methods of achieving desirable results. Many of the methods in `modelflow` have been developed to work well with Jupyter Notebook. Indeed this documentation was written as a series of Jupyter Notebooks bound together with Jupyter Book.

Jupyter Notebook is not the only way to work with modelflow or Python. As users become more advanced they are likely to migrate to a more program-centric IDE (Interactive Development Environment) like Spyder or Microsoft Visual Code.

However, to start Jupyter Notebooks are a great way to learn, follow work done by others and tweak them to fit your own needs.

There are many fine tutorials on Jupyter Notebook on the web, and The official Jupyter site is a good starting point. The following aims to provide enough information to get a user started. Another good reference is here.

3.1 Starting Jupyter Notebook

Each time, a user wants to work with `modelflow`, they will need to activate the `modelflow` environment by

1. Opening the Anaconda command prompt window
2. Activate the ModelFlow environment we just created by executing the following command

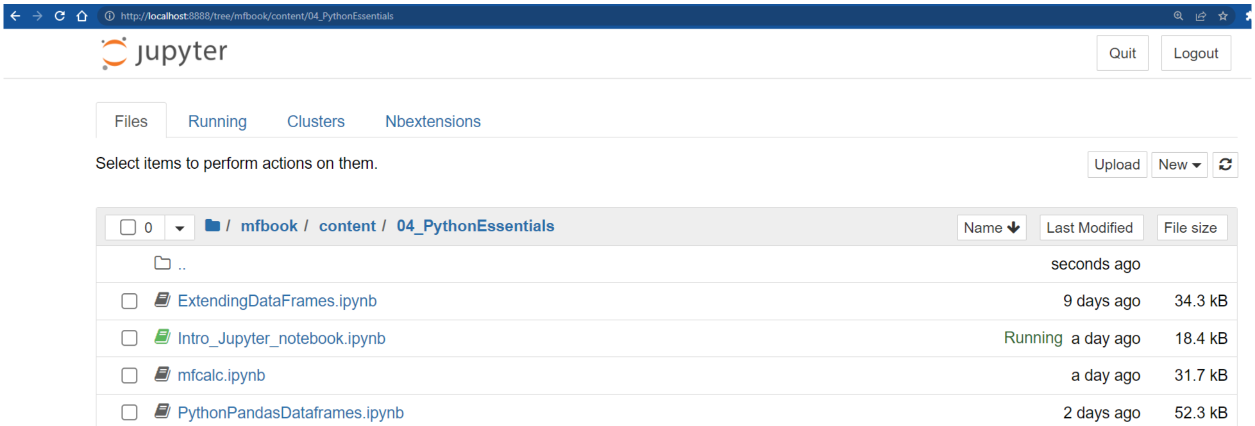
```
conda activate modelflow
```

From here, any number of mechanisms can be used to interact with `modelflow` and World Bank models.

To use Jupyter Notebook the Jupyter notebook, must be first started. Following steps 1-2 above, a user would need to execute from the conda command line:

```
jupyter notebook
```

This will launch the Jupyter environment in your default web browser, which should look something like this:



where the directory structure presented is that of the directory from the `jupyter notebook` command was executed.

> **Warning:** Note the directory from which you execute the `jupyter notebook` **mfbook** in the example above will be the **root** directory for the jupyter session, and only directories and files below this root directory will be accessible by jupyter.

3.2 Creating a notebook

The idea behind jupyter notebook was to create an interactive version of the notebooks that scientists use(d) to:

- record what they have done
- perhaps explain why
- document how data was generated, and
- record the results of their experiments

The motivation for these notebooks and Jupyter notebook is to record the precise steps taken to produce a set of results, which if followed by others would allow the to generate the same results.

To create a notebook you must select from the Jupyter Notebook menu

File-> New Notebook

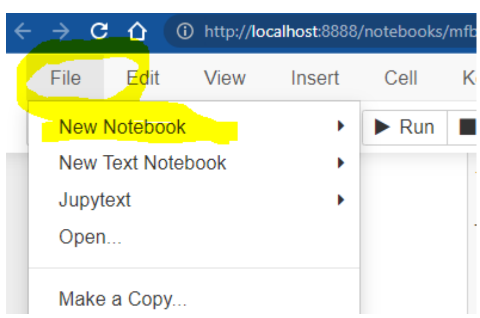


Fig. 3.1: A newly created Jupyter Notebook session

This will generate a blank unnamed notebook with one empty cell, that looks something like this:

`![NewCell](./Newcell.png)`

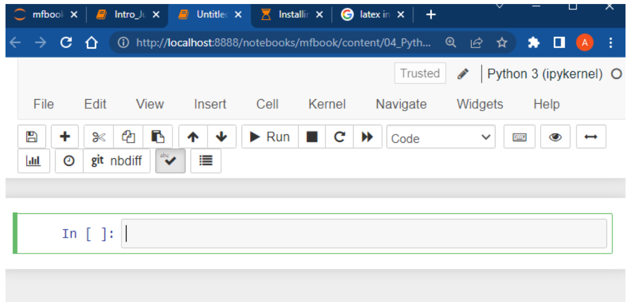


Fig. 3.2: A newly created Jupyter Notebook

Warning: Each notebook has associated with it a "Kernel", which is an instance of the computing environment in which code will be executed. For Jupyter Notebooks that work with `modelflow` this will be a Python Kernel. If your computer has more than one "kernel's" installed on it, you may be prompted when creating a new notebook for the kernel with which to associate it. Typically this should be the Python Kernel under which your modelflow was built – currently python 3.9 in April 2023.

3.3 Jupyter Notebook cells

A Jupyter Notebook is comprised of a series of cells.

Jupyter Notebook cells can contain:

- **computer code** (typically python code, but as noted other kernels – like Eviews – can be used with jupyter).
- **markdown text**: plain text that can include special characters that make some text appear as bold, or indicate the text is a header, or instruct Jupyter Notebook to render the text as a mathematical formula. All of the text in this document was entered using Jupyter Notebook's markdown language
- Results (in the form of tables or graphs) from the execution of computer code specified in a code cell

Every cell has two modes:

1. Edit mode – indicated by a green vertical bar. In edit mode the user can change the code, or the markdown.
2. Select/Copy mode – indicated by a blue vertical bar. This will be teh state of the cell when its content has been executed. For markdown cells this means that the text and special characters have been rendered into formatted text. For code cells, this means the code has been executed and its output (if any) displayed in an output cell.

Users can switch between Edit and Select/Copy Mode by hitting Enter

This entire book was generated using markdown cells, code cells and output cells from Jupyter Notebooks.

Note: Jupyter Notebooks were designed to facilitate *replicability*: the idea that a scientific analysis should contain - in addition to the final output (text, graphs, tables) - all the computational steps needed to get from raw input data to the results.

3.3.1 How to add, delete and move cells

The newly created Jupyter Notebook will have a code cell by default. Cells can be added, deleted and moved either via mouse using the toolbar or by keyboard shortcut.

Using the Toolbar

- **+ button**: add a cell below the current cell
- **scissors**: cut current cell (can be undone from "Edit" tab)
- **clipboard**: paste a previously cut cell to the current location
- **up- and down arrows**: move cells (cell must be in Select/Copy mode – vertical side bar must be blue)
- **hold shift + click cells in left margin**: select multiple cells (vertical bar must be blue)

Using keyboard short cuts

- **esc + a**: add a cell above the current cell
- **esc + b**: add a cell below the current cell
- **esc + d+d**: delete the current cell

3.3.2 Change the type of a cell

You can also change the type of a cell. New cells are by default "code" cells.

Using the Toolbar

- Select the desired type from the drop down. options include
  - Markdown
  - Code
  - Raw NBConvert
  - Heading

Using keyboard short cuts

- **esc + m**: make the current cell a markdown cell
- **esc + y**: make the current cell a code cell

Auto-complete and context-sensitive help

When editing a code cell, you can use these short-cuts to autocomplete and or call up documentation for a command.

- **tab**: autocomplete and method selection
- **double tab**: documention (double tab for full doc)

3.4 Execution of cells

Every cell in a Jupyter Notebook can be executed, either by using the Run button on the Jupyter Notebook menu, or by using one of **two keyboard shortcuts**:

- **ctrl + Enter**: Executes the code in the cell or formats the markdown of a cell. The current cell retains the focus – cursor stays on cell executed.
- **shift + enter**: Executes the code in the cell or formats the markdown of a cell. Focus (cursor) jumps to the next cell

For other useful shortcuts see “Help” => “Keyboard Shortcuts” or simply press keyboard icon in the toolbar.

3.4.1 Executing python code

Below is a code with some standard python that declares a variable “x”, assigns it the value 10, declares a second variable “y” and assigns it the value 45. The final line of y alone, instructs python to display the value of the variable y. The results of the operation appear in Jupyter Notebook as an output cell Out[#]. By pressing **Ctrl-Enter** the code will be executed and the output displayed below.

```
x = 10
y = 45
y
```

```
45
```

The semi-colon “;” suppresses output in Jupyter Notebook

In the example below, a semi-colon “;” has been appended to the final line. This suppresses the display of the value contained by y; As a result there is no output cell.

```
x = 10
y = 45
y;
```

Another way to display results is to use the print function.

```
x = 10
print(x)
```

```
10
```

Variables in a Jupyter Notebook session are persistent, as a result in the subsequent cell, we can declare a variable ‘z’ equal to 2*y and it will have the value 90.

```
z=y*2
z
```

```
90
```

3.5 Markdown cells and the markdown scripting language in Jupyter Notebook

Text cells in a notebook can be made more interesting by using markdown.

Cells designated as markdown cells when executed are rendered in a rich text format (html).

Markdown is a lightweight markup language for creating formatted text using a plain-text editor. Used in a markdown cell of Jupyter Notebook it can be used to produce nicely formatted text that mixes text, mathematical formulae, code and outputs from executed python code.

Rather than the relatively complex commands of html <h1></h1>, markdown uses a simplified set of commands to control how text elements should be rendered.

3.5.1 Common markdown commands

Some of the most common of these include:

symbol	Effect
#	Header
##	second level
###	third level etc.
Bold text	**Bold text**
Italics text	*Italics text*
* text	Bulleted text or dot notes
1. text	1. Numbered bullets

3.5.2 Tables in markdown

Tables like the one above can be constructed using | as separators.

The |:–|:——————| on the second line tells the Table generator how to justifythe cintents of columns. :– means left justify :–: means center justify and –: means right justify.

Below is the markdown code that generated the above table:

```
| symbol           | Effect          |
|:--|:------|                                # Specifies the justification for the columns of the table.
| \#               | Header         |
| \#\#             | second level  |
| \*\*Bold text\*\* | **Bold text**   |
| \*Italics text\* | *Italics text*   |
|
| 1\. text   | 1. Numbered bullets   |
```

3.5.3 Displaying code

To display a (unexecutable) block of code within a markdown cell, encapsulate it (surround it) with backticks `.

For a multiline section of code use three backticks at the beginning and end.

``` Multi line text to be rendered as code ```.

will be rendered as: `text to be rendered as code`.

```
Multi line
text to be rendered as code
```

For inline code references ' a sigle back tick at the beginning and end suffices.

This sentence:

An example sentence with some back-ticked `text as code` in the middle.

will render as:

An example sentence with some back-ticked `text as code` in the middle.

3.5.4 Rendering mathematics in markdown

Jupyter Notebook's implementation of Markdown supports `latex` mathematical notation.

Inline enclose the `latex` code in `$`:

An Equation: `$y_t = \beta_0 + \beta_1 x_t + u_t\$` will renders as: $y_t = \beta_0 + \beta_1 x_t + u_t$

if enclosed in `$$ $$` it will be centered on its own line.

$$y_t = \beta_0 + \beta_1 x_t + u_t$$

Complex and multi-line math

```
\begin{align}
Y_t  &=  C_t+I_t+G+t+ (X_t-M_t) \\
C_t &= c_t(C_{t-1},C_{t-2},I_t,G_t,X_t,M_t,P_t)\\
I_t &= c_t(I_{t-1},I_{t-2},C_t,G_t,X_t,M_t,P_t)\\
G_t &= c_t(G_{t-1},G_{t-2},C_t,I_t,X_t,M_t,P_t)\\
X_t &= c_t(X_{t-1},X_{t-2},C_t,I_t,G_t,M_t,P_t,P^f_t)\\
M_t &= c_t(M_{t-1},M_{t-2},C_t,I_t,G_t,X_t,P_t,P^f_t)
\end{align}
```

The above `latex` mathematics code uses the `&` symbol to tell `latex` to align the different lines (separated by `\\`) on the character immediately after the `&`. In this instance the equals "=" sign.

$$Y_t = C_t + I_t + G + t + (X_t - M_t) \tag{3.1}$$
$$C_t = c_t(C_{t-1}, C_{t-2}, I_t, G_t, X_t, M_t, P_t) \tag{3.2}$$
$$I_t = c_t(I_{t-1}, I_{t-2}, C_t, G_t, X_t, M_t, P_t) \tag{3.3}$$
$$G_t = c_t(G_{t-1}, G_{t-2}, C_t, I_t, X_t, M_t, P_t) \tag{3.4}$$
$$X_t = c_t(X_{t-1}, X_{t-2}, C_t, I_t, G_t, M_t, P_t, P_t^f) \tag{3.5}$$
$$M_t = c_t(M_{t-1}, M_{t-2}, C_t, I_t, G_t, X_t, P_t, P_t^f) \tag{3.6}$$
```

3.5.5 links to more info on markdown

There are several very good markdown cheatsheets on the internet, one of these is here

CHAPTER

FOUR

SOME PYTHON BASICS

Before using `modelflow` with the World Bank's MFMod models, users will have to understand at least some basic elements of `python` syntax and usage. Notably they will need to understand about packages, libraries and classes, how to access them.

4.1 Starting python in windows

To begin using `modelflow`, python itself needs to be started. This can be done either using the `Anaconda` navigator or from the command line shell. In either case, the user will need to start python and select the `modelflow` environment.

4.2 Anaconda navigator

1. Start Anaconda Navigator by typing Anaconda in the Start window and opening the Navigator (see Figure).
2. From Anaconda Navigator select the `Modelflow` environment (see figure)

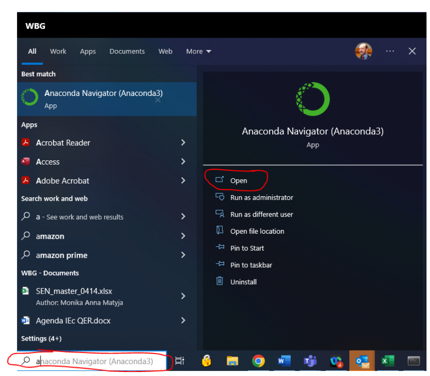


Fig. 4.1: A newly created Jupyter Notebook session

1. Once the environment is selected the user can either select a command line environment or start jupyter notebook by clicking on either the
   1. Jupyter Notebook environment
   2. The command line environment
   3. A programming IDE environment

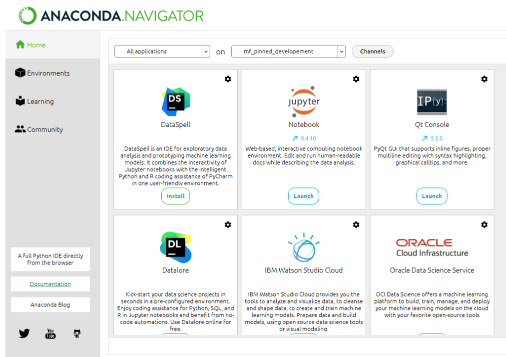


Fig. 4.2: A newly created Jupyter Notebook session

4.3 Python packages, libraries and classes

Some features of `python` are built-in out-of-the-box. Others build up on these basic features.

A **python class** is a code template that defines a python object. Classes can have properties [variables or data] associated with them and methods (behaviours or functions) associated with them. In python a class is created by the keyword class. An object of type class is created (instantiated) using the class's "constructor" – a special method that creates an object that is an instance of a class.

A **module** is a Python object consisting of Python code. A module can define functions, classes and variables. A module can also include runnable code.

A **python package** is a collection of modules that are related to each other. When a module from an external package is required by a program, that package (or module in the package) must be **imported** into the current session for its modules can be put to use.

A **python library** is a collection of related modules or packages.

`Modelflow` is a python package that *inherits* (build on or adds to) the methods and properties of other `python` classes like `pandas`, `numpy` and `mathplotlib`.

Note: In modelflow the model is a class and we can create an instance of a model (an object filled with the characteristics of the class) by executing the code `mymodel = model(myformulas)` see below for a working example.

4.4 Importing packages, libraries, modules and classes

Some libraries, packages, and modules are part of the core python package and will be available (importable) from the get-go. Others are not, and need to be installed before importing them into a session.

If you followed the modelflow installation instructions you have already downloaded and installed on your computer all the packages necessary for running World Bank models under modelflow. But to work with them in a given Jupyter Notebook session or in a program context, you will also need to `import` them into your session before you call them.

Note: **Installation** of a package is not the same as **import**ing a package. To be imported a package must be installed once on the computer that wishes to use it. Once it has been installed, the package must be imported into each python session where it is to be used.

Typically a python program will start with the importation of the libraries, classes and modules that will be used. Because a Jupyter Notebook is essentially a heavily annotated program, it also requires that packages used be imported.

As described above packages, libraries and modules are containers that can include other elements. Take for example the package Math.

To import the Math Package we execute the command `import math`. Having done that we can can call the functions and data that are defined in it.

```
# the "#"" in a code cell indicates a comment, test after the # will not be executed
import math

# Now that we have imported math we can access some of the elements identified in the␣
↪package,
# For example math contains a definition for pi, we can access that by executing the␣
↪pi method
# of the library math
math.pi
```

```
3.141592653589793
```

4.4.1 Import specific elements or classes from a module or library

The python package `math` contains several functions and classes.

If I want I can import them directly. Then when I call them I will not have to precede them with the name of their libary. to do this I use the **from** syntax. `from math import pi,cos,sin` will import the pi constant and the two functions cos and sin and allow me to call them directly.

Compared these calls with the one in the preceding section – there the call to the method pi has to be preceded by its namespace designator math. i.e. `math.pi`. Below we import pi directly and can just call it with pi.

```
from math import pi,cos,sin

print(pi)
print(cos(3))
```

```
3.141592653589793
-0.9899924966004454
```

4.4.2 import a class but give it an alias

A class and instead of using its full name as above or it can be given an alias, that is hopefully shorter but still obvious enough that the user knows what class is being referred to.

For example `import math as m` allows a call to pi using the more succint syntax `m.py`.

```
import math as m
print(m.pi)
print(m.cos(3))
```

```
3.141592653589793
-0.9899924966004454
```

4.4.3 Standard aliases

Some packages are so frequently used that by convention they have been “assigned” specific aliases.

For example:

Common aliases

Alias	aliased package	example	functionalty
pd	pandas	import pandas as pd	Pandas are used for storing and retriveing data
np	numpy	import numpy as np	Numpy gives access to some advanced mathematical features

You don’t have to use those conventions but it will make your code easier to read by others who are familiar with it.

CHAPTER

FIVE

INTRODUCTION TO PANDAS DATAFRAMES

Modelflow is built on top of the Pandas library. Pandas is the Swiss knife of data science and can perform an impressing array of date oriented tasks.

This tutorial is a very short introduction to how pandas dataframes are used with Modelflow. For a more complete discussion see any of the many tutorials on the internet, notably:

- Pandas homepage
- Pandas community tutorials

5.1 Import the pandas library

As with any python program, in order to use a package or library it must first be imported into the session. As noted above, by convention pandas is imported as pd

```
import pandas as pd
```

Pandas, like any library, contains many classes and methods. The discussion below focuses on **Series** and **DataFrames** two classes that are part of the pandas library. Both `series` and `dataframes` are containers that can be used to store time-series data and that have associated with them a number of very useful methods for displaying and manipulating time-series data.Unlike other statistical packages neither `series` nor `dataframes` are inherently or exclusively time-series in nature. `Modelflow` and macro-economists use them in this way, but the classes themselves are not dated in anyway out-of-the-box.

5.2 The `Pandas` class `series`

A pandas series is class that can be used to instantiate an object that holds a two dimensional array comprised of values and an index.

The constructor for a `Series` object is `pandas.Series()`. The content inside the parentheses will determine the nature of the series-object generated. As an object-oriented language Python supports `overrides` (which is to say a method can have more than one way in which it can be called). Specifically there can be different constructors defined for a class, depending on how the data that is to be used to initialize it is organized.

5.2.1 Series declared from a list

The simplest way to create a Series is to pass an array of values as a Python list to the Series constructor.

Note: A list in python is a comma delimited collection of items. It could be text, numbers or even more complex objects. When declared (and returned) list are enclosed in square brackets.

For example both of the following two lines are perfectly good examples of lists.

mylist=[2,7,8,9] mylist2=["Some text","Some more Text",2,3]

The list is entirely agnostic about the type of data it contains.

In the examples below Simplest, Simple and simple3 are all series – although series3 which is derived from a list mixing text and numeric values would be hard to interpret as an economic series.

```
values=[7,8,9,10,11]
weird=["Some text","Some more Text",2,3]

# Here the constructor is passed a numeric list
Simplest=pd.Series([2,3,4,5,6])
Simplest
```

```
0    2
1    3
2    4
3    5
4    6
dtype: int64
```

```
# In this case the constructor is passed a variable that contains a list
simple2=pd.Series(values)
simple2
```

```
0     7
1     8
2     9
3    10
4    11
dtype: int64
```

```
# Here the constructor is passed a variable containing a list that is a mix of
# alphanumerics and numerical values
simple3=pd.Series(weird)
simple3
```

```
0         Some text
1    Some more Text
2                 2
3                 3
dtype: object
```

Note that all three series have different length.

Moreover, constructed in this way (by passing a list to the constructor) each of these `Series` are automatically assigned a zero-based index (a numerial index that starts with 0).

5.2.2 Series declared using a specific index

In this example the series Simple and Simple2 are recreated (overwritten), but this time an index is specified. Here the index is declared as a(nother) list.

```
# In this example the constructor is given both the values
# and specific values for the index
Simplest=pd.Series([2,3,4,5,6],index=[1966,1967,1996,1999,2000])
Simplest
```

```
1966    2
1967    3
1996    4
1999    5
2000    6
dtype: int64
```

```
simple2=pd.Series(values,index=[1966,1967,1996,1999,2000])
simple2
```

```
1966     7
1967     8
1996     9
1999    10
2000    11
dtype: int64
```

Now the Series look more like time series data!

5.2.3 Create Series from a dictionary

In python a dictionary is a data structure that is more generally known in computer science as an associative array. A dictionary consists of a collection of key-value pairs, where each key-value pair *maps* or *links* the key to its associated value.

Note: A dictionary is enclosed in curly brackets {}, versus a list which is enclosed in square brackets[].

Thus mydict={“1966”:2,”1967”:3,”1968”:4,”1969”:5,”2000”:-15} creates an object called mydict. `mydict`maps (or links) the key “1966” links to the value 2.

Note: In this example the Key was a string but we could just as easily made it a numerical value:

mydict2={1966:2,1967:3,1968:4,1969:5,2000:-15} creates an object called mydict2 that links (maps) the key “1966” to the value 2.

The series constructor also accepts a dictionary, and maps the key to the index of the Series.

```
mydict2={1966:2,1967:3,1968:4,1969:5,2000:6}
simple2=pd.Series(mydict2)
simple2
```

```
1966    2
1967    3
1968    4
1969    5
2000    6
dtype: int64
```

5.3 Properties and methods of `DataFrames` in `modelflow`

Any class can have both properties (data) and methods (functions that operate on the data of the particular instance of the class). With object-oriented programming languages like python, classes can be built as supersets of existing classes. The `modelflow` class `model` inherits or encapsulates all of the features of the pandas dataframe and extends it in many important ways. Some of the methods below are standard pandas methods, others have been added to it by `modelflow` features

Much more detail on standard pandas dataframes can be found on the official pandas website.

5.3.1 `DataFrames`

The `DataFrame` is the primary structure of pandas and is a two-dimensional data structure with named rows and columns. Each columns can have different data types (numeric, string, etc).

By convention, a dataframe if often called df or some other modifier followed by df, to assist in reading the code.

5.3.2 Creating or instantiating a dataframe

Like any object, a `DataFrame` can be created by calling the constructor of the pandas class `DataFrame`.

Each class has many constructors, so there are very many ways to create a `dataframe`. The `pandas.DataFrame()` method is constructor for the `DataFrame` class. It takes several forms (as with `Series`), but always returns an instance (instantiates) of a `DataFrame` object – i.e. a variable whose contents are a `DataFrame`.

The code example below creates a `DataFrame` of three columns A,B,C; indexed between 2019 and 2021. Macroeconomists may interpret the index as dates, but for pandas they are just numbers.

Below a `DataFrame` named `df` is instantiated from a dictionary and assigned a specific index by passing a list of years as the index.

```
df = pd.DataFrame({'B': [1,1,1,1],'C':[1,2,3,6],'E':[4,4,4,4]},index=[2018,2019,2020,
↪2021])
df
```

```
      B  C  E
2018  1  1  4
2019  1  2  4
2020  1  3  4
2021  1  6  4
```

Note: In the DataFrames that are used in macrostructural models like MFMod, each column is often interpreted as a time-series of an economic variable. So in this dataframe, normally A, B and C each be interpreted as economic time series.

That said, there is nothing in the DataFrame class that suggests that the data it stores must be time-series or even numeric in nature.

5.3.3 Adding a column to a dataframe

If a value is assigned to a column that does not exist, pandas will add a column with that name and fill it with values resulting from the calculation.

Note: The size of the object assigned to the new column must match the size (number of rows) of the pre-existing DataFarame.

```
df["NEW"]=[10,12,10,13]
df
```

```
      B  C  E  NEW
2018  1  1  4   10
2019  1  2  4   12
2020  1  3  4   10
2021  1  6  4   13
```

5.3.4 Revising values

If the column exists than the = method will revise the values of the rows with the values assigned in the statement.

Warning: The dimensions of the list assigned via the = method must be the same as the DataFrame (i.e. there must be exactly as many values as there are rows). Alternatively if only one value is provided, then that value will replace all of the values in the specified column (be broadcast to the other rows in the column).

```
df["NEW"]=[11,12,10,14]

df
```

```
      B  C  E  NEW
2018  1  1  4   11
2019  1  2  4   12
2020  1  3  4   10
2021  1  6  4   14
```

```
# replace all of the rows of column B with the same value
df['B']=17
df
```

```
       B   C   E   NEW
2018  17   1   4    11
2019  17   2   4    12
2020  17   3   4    10
2021  17   6   4    14
```

5.4 Column names in Modelflow

Modelflow variable names

Modelflow places more restrictions on column names than do pandas *per se*.

While pandas dataframes are very liberal in what names can be given to columns, `modelflow` is more restrictive.

Specifically, in modelflow a variable name must:

- start with a letter
- be upper case

Thus while all these are legal column names in pandas, some are illegal in modelflow.

Variable Name	Legal in modelfow?	Reason
IB	yes	Starts with a letter and is uppercase
ib	no	lowercase letters are not allowed
42ANSWER	No	does not start with a letter
_HORSE1	No	does not start with a letter
A_VERY_LONG_NAME_THAT_IS_LEGAL_3	Yes	Starts with a letter and is uppercase

5.5 .index and time dimensions in Modelflow

As we saw above, series have indices. Dataframes also have indices, which are the row names of the dataframe.

In `modelflow` the index series is typically understood to represent a date.

For yearly models a list of integers like in the above example works fine.

For higher frequency models the index can be one of pandas datatypes.

Warning: Not all datetypes work well with the graphics routines of modelflow. Users are advised to use the `pd.period_range()` method to generate date indexes.

For example:

```
dates = pd.period_range(start='1975q1',end='2125q4',freq='Q')
df.index=dates
```

5.5.1 Leads and lags

In modelflow leads and lags can be indicated by following the variable with a parenthesis and either -1 or -2 two for one or two period lags (where the number following the negative sign indicates the number of time periods that are lagged). Positive numbers are used for forward leads (no +sign required).

When a method defined by the `modelflow` class encounters something like `A(-1)`, it will take the value from the row above the current row. No matter if the index is an integer, a year, quarter or a millisecond. The same goes for leads, `A(+1)` will return the value of `A` in the next row.

As a result in a quarterly model `B=A(-4)` would assign B the value of A from the same quarter in the previous year.

5.5.2 .columns lists the column names of a dataframe

The method `.columns` returns the names of the columns in the dataframe.

```
df.columns
```

```
Index(['B', 'C', 'E', 'NEW'], dtype='object')
```

5.5.3 .size indicates the dimension of a list

so `df.columns.size` returns the number of columns in a dataframe.

```
df.columns.size
```

```
4
```

The dataframe df has 4 columns.

5.5.4 .eval() evaluates calculates an expression on the data of a dataframe

`.eval` is a native dataframe method, which does calculations on a `dataframe` and returns a revised `datafame`. With this method expressions can be evaluated and new columns created.

```
df.eval('''X = B*C
           THE_ANSWER = 42''')
```

```
       B  C  E  NEW    X  THE_ANSWER
2018  17  1  4   11   17          42
2019  17  2  4   12   34          42
2020  17  3  4   10   51          42
2021  17  6  4   14  102          42
```

```
df
```

```
       B  C  E  NEW
2018  17  1  4   11
2019  17  2  4   12
2020  17  3  4   10
2021  17  6  4   14
```

In the above example the resulting dataframe is displayed but is not stored.

To store it, the results of the calculation must be assigned to a variable. The pre-existing dataframe can be overwritten by assigning it the result of the eval statement.

```
df=df.eval('''X = B*C
           THE_ANSWER = 42''')
df
```

```
       B  C  E  NEW    X  THE_ANSWER
2018  17  1  4   11   17          42
2019  17  2  4   12   34          42
2020  17  3  4   10   51          42
2021  17  6  4   14  102          42
```

With this operation the new columns, x and THE_ANSWER have been appended to the dataframe df.

Note: The `.eval()` method is a native pandas method. As such it cannot handle lagged variables (because pandas do not support the idea of a lagged variable.

The `.mfcalc()` and the `.upd()` methods discussed below are `modelflow` features that extend the functionalities native to `dataframe` that allows such calculations to be performed.

5.5.5 .loc[] selects a portion (slice) of a dataframe

The `.loc[]` method allows you to display and/or revise specific sub-sections of a column or row in a dataframe.

.loc[row,column] A single element

`.loc[row,column]` operates on a single cell in the dataframe. Thus the below displays the value of the cell with index=2019 observation from the column C.

```
df.loc[2019,'C']
```

```
2
```

.loc[:,column] A single column

The lone colon in a loc statement indicates all the rows or columns. Here all of the rows.

```
df.loc[:,'C']
```

```
2018    1
2019    2
2020    3
2021    6
Name: C, dtype: int64
```

.loc[row,:] A single row

Here all of the columns, for the selected row.

```
df.loc[2019,:]
```

```
B             17
C              2
E              4
NEW           12
X             34
THE_ANSWER    42
Name: 2019, dtype: int64
```

.loc[:,[names...]] Several columns

Passing a list in either the rows or columns portion of the loc statement will allow multiple rows or columns to be displayed.

```
df.loc[[2018,2021],['B','C']]
```

```
       B  C
2018  17  1
2021  17  6
```

.loc using the colon to select a range

with the colon operator we can also select a range of results.

Here from 2018 to 2019.

```
df.loc[2018:2020,['B','C']]
```

```
       B  C
2018  17  1
2019  17  2
2020  17  3
```

.loc[] can also be used on the left hand side to assign values to specific cells

This can be very handy when updating scenarios.

```
df.loc[2019:2020,'C'] = 17
df
```

```
       B   C  E  NEW    X  THE_ANSWER
2018  17   1  4   11   17          42
2019  17  17  4   12   34          42
2020  17  17  4   10   51          42
2021  17   6  4   14  102          42
```

Warning: The dimensions on the right hand side of = and the left hand side should match. That is: either the dimensions should be the same, or the right hand side should be `broadcasted` into the left hand slice.

For more on broadcasting see here

For more info on the .loc[] method

- Description
- Search

For more info on pandas:

- Pandas homepage
- Pandas community tutorials

CHAPTER

SIX

MODELFLOW EXTENSIONS TO PANDAS

Modeflow inherits all the capabilities of pandas and extends some as well.

Data in a dataframe can be modified directly with built-in pandas functionalities like `.loc[]` and `eval()`, but `modelflow` extends these capabilities with in important ways with the `.upd()` and `.mfcalc()` methods.

6.1 .upd() method of modelflow

The `.upd()` method extends pandas by giving the user a concise and expressive way to modify data in a dataframe using a syntax that a database-manager or macroeconomic modeler might find more natural.

Notably it allows the user to employ formula's to do updates, and supports both lags and leads on variables.

`.upd()` can be used to:

- Perform different types of updates
- Perform multiple updates each on a new line
- Perform changes over specific periods
- Use one input which is used for all time frames, or a separate input for each time
- Preserve pre-shock growth rates for out of sample time-periods
- Display results

6.1.1 `.upd()` method operators

Below are some of the operators that can be used in the `.upd()` method

Types of update:

Update to perform	Use this operator
Set a variable equal to the input	=
Add the input to the input	+
Set the variable to itself multiplied by the input	*
Increase/Decrease the variable by a percent of itself (1+input/100)	%
Set the growth rate of the variable to the input	=growth
Change the growth rate of the variable to its current growth rate plus the input value in percentage points	+growth
Specify the amount by which the variable should increase from its previous period level ($\Delta = var_t - var_{t-1}$)	=diff

> **Danger:** Note: the syntax of an update command requires that there be a space between variable names and the operators.
>
> Thus `df.upd("A = 7")` is fine, but `df.upd("A =7")` will generate an error.
>
> Similarly `df.upd("A * 1.1")` is fine, but `df.upd("A* 1.1")` will generate an error.

6.1.2 `.upd()` some examples

6.1.3 Setting up the python environment

In order to use `.upd()` all of the necessary libraries must be **imported** into the python session.

```
%load_ext autoreload
%autoreload 2

# First import pandas and the model into the  workspace
# There is no problem importing multiple times, though it is not very efficient.
import pandas as pd

from modelclass import model
# functions that improve rendering of modelflow outputs under Jupyter Notebook
model.widescreen()
model.scroll_off()
```

```
<IPython.core.display.HTML object>
```

Now create a dataframe using standard pandas syntax. In this instance with years as the index and a dictionary defining the variables and their data.

```
# Create a dataframe using standard pandas

df = pd.DataFrame({'B': [1,1,1,1],'C':[1,2,3,6],'E':[4,4,4,4]},index=[2018,2019,2020,
↪2021])
df
```

```
      B  C  E
2018  1  1  4
2019  1  2  4
2020  1  3  4
2021  1  6  4
```

A somewhat more creative way to initialize the dataframe for dates would use a loop to specify the dates that get passed to the constructor as an argument.

Below a dataframe df with two Series (A and B), is initialized with the values 100 for all data points.

The index is defined dynamically by a loop `index=[2020+v for v in range(number_of_rows)]` that runs for number_of_rows times (6 times in this example) setting v equal to 2020+0, 2020+1,…,202+5. The resulting list whose values are assigned to index is [2020,2021,2022,2023,2024,2025].

The big advantage of this method is that if the user wanted to have data created for the period 1990 to 2030, they would only have to change number_of_rows from 6 to 41 and 2020 in the loop to 1990.

The second example simplifies further by just specifying the begin and end point of the range.

```
#define the number of years for which the data is to be created.
number_of_rows = 6

# call the dataframe constructor
df = pd.DataFrame(100,
       index=[2020+v for v in range(number_of_rows)], # create row index
       # equivalent to index=[2020,2021,2022,2023,2024,2025]
       columns=['A','B'])                                  # create column name
df

df1 = pd.DataFrame(200,
       index=[v for v in range(2020,2030)], # create row index
       # equivalent to index=[2020,2021,...,2030]
       columns=['A1','B1'])                                  # create column name
df1
```

```
       A1   B1
2020  200  200
2021  200  200
2022  200  200
2023  200  200
2024  200  200
2025  200  200
2026  200  200
2027  200  200
2028  200  200
2029  200  200
```

6.1.4 Use .upd to create a new variable (= operator)

With standard pandas a user can add a column (series) to a dataframe simply by assigning a adding to a dataframe. For example:

`df['NEW2']=[17,12,14,15]`

`.upd()` provides this functionality as well.

```
df2=df.upd('c = 142')
df2
```

```
        A    B      C
2020  100  100  142.0
2021  100  100  142.0
2022  100  100  142.0
2023  100  100  142.0
2024  100  100  142.0
2025  100  100  142.0
```

Note: Note that the new variable name was entered as a lower case 'c' here. Lowercase letters are not legal `modelflow` variable names. The `.upd()` method knows is part of modelflow and knows this rule, so it automatically translates lowercase entries into upper case so that the statement works.

6.1.5 Multiple updates and specific time periods

The modelflow method `.upd()` takes a string as an argument. That string can contain a single update command or can contain multiple commands.

Moreover by including a <Begin End> date clause in a given update command, the update will be restricted to the associated time period.

The below illustrates this, modifying two existing variables A, B over different time periods and creating a new variable.

> **Danger:** Note that the third line inherits the time period of the previous line.
>
> Note also the submitted string can include comments as well (denoted with the standard python # indicator).

```
df.upd("""
# Same number of values as years
<2021 2024> A = 42 44 45 46    # 4 years
<2020      > B = 200            # 1 year
c = 500                         # Same period as previous line
<-0 -1> D = 33                    # All years
""")
```

```
         A    B      C     D
2020   100  200  500.0  33.0
2021    42  100    0.0  33.0
2022    44  100    0.0  33.0
2023    45  100    0.0  33.0
2024    46  100    0.0  33.0
2025   100  100    0.0  33.0
```

Time scope of .upd()

Made this a margin just to see

The update command takes a variety of mathematical operators `=, +, *, % =GROWTH, +GROWTH, =DIFF` and applies them to data for the period set in the leading <>.

If the user wants to modify a series or group of series for only a specific point in time or a period of time, she can indicate the period in the command line.

- If **one date** is specified the operation is applied to a single point in time
- If **two dates** are specifies the operation is applied over a period of time.

The selected time period will persist until re-set with a new time specification. Useful to avoid visual noise if several variables are going to be updated for the same time period.

The time period can be rest to the full time-period by using the special <-0 -1> time period. More generally:

- Indicates the start of the dataframe use -0
- Indicates the end of the dataframe use -1

If no time is provided the dataframe start and end period will be used.

6.1.6 Setting specific datapoints to specific values

This example, demonstrates the equals operator. The = operator indicates that the variable a should be set equal to the indicated values following the = operator (42 44 45 46 in the first line, 200 in the second and 500 inthe third). The dates enclosed in <> indicate the period over which the change should be applied.

Either:

- The number of data points provided must match the number of dates in the period, Or
- Only one data point is provided, it is applied to all dates in the period.

If only one period is to be modified then it can be followed by just one date.

Note that the final line inherited the time period set in the second line.

```
df.upd("""
# Same number of values as years
<2021 2024> A = 42 44 45 46     # 4 years
<2023      > B = 200            # 1 year
c = 500
""")
```

```
         A    B      C
2020   100  100    0.0
2021    42  100    0.0
2022    44  100    0.0
2023    45  200  500.0
2024    46  100    0.0
2025   100  100    0.0
```

6.1.7 Adding the specified values to all values in a range (the + operator)

NB: Here upd with the + operator indicates that we are adding 42.

```
df.upd('''
# Or one number to all years in between start and end
<2022 2024> B  +  42    # one value broadcast to 3 years
''')
```

```
         A    B
2020   100  100
2021   100  100
2022   100  142
2023   100  142
2024   100  142
2025   100  100
```

6.1.8 Multiplying all values in a range by the specified values (the * operator)

```
df.upd('''
# Same number of values as years
<2021 2023> A *  42 44 55
''')
```

```
         A    B
2020   100  100
2021  4200  100
2022  4400  100
2023  5500  100
2024   100  100
2025   100  100
```

6.1.9 Increasing all values in a range by a specified percent amount (the % operator)

In this example:

- A is increased by 42 and 44% over the range 2021 through 2022.
- B is increased by 10 percent in all years
- C is a new variable, is created and set to 100 for the whole range
- C is decreased by 12 percent over the range 2023 through 2025.

```
df.upd('''
<2021 2022 > A %  42 44
<-0 -1> B % 10            # all rows
C = 100                   # all rows persist
<2023 2025> C % -12       # now only for 3 years
''')
```

```
        A      B      C
2020  100  110.0  100.0
2021  142  110.0  100.0
2022  144  110.0  100.0
2023  100  110.0   88.0
2024  100  110.0   88.0
2025  100  110.0   88.0
```

6.1.10 Set the percent growth rate to specified values (=GROWTH)

```
res = df.upd('''
# Same number of values as years
<2021 2022> A =GROWTH  1 5
<2020> c = 100
<2021 2025> c =GROWTH 2
''')
print(f'Dataframe:\n{res}\n\nGrowth:\n{res.pct_change()*100}\n') # Explained b
```

```
Dataframe:
           A    B           C
2020  100.00  100  100.000000
2021  101.00  100  102.000000
2022  106.05  100  104.040000
2023  100.00  100  106.120800
2024  100.00  100  108.243216
2025  100.00  100  110.408080

Growth:
             A    B    C
2020       NaN  NaN  NaN
2021  1.000000  0.0  2.0
2022  5.000000  0.0  2.0
2023 -5.704856  0.0  2.0
2024  0.000000  0.0  2.0
2025  0.000000  0.0  2.0
```

6.1.11 Add or subtract from the existing percent growth rate (+GROWTH operator)

The below example is a bit more complicated.

The first line sets the growth rate of A to 1% in all periods beginning in 2021

The second command adds 2 3 4 5 6 to the growth rates in each period after 2021, resulting in growth rates of 3,4,5,6,7.

```
res =df.upd('''
<2021 > A =GROWTH  1  # All selected years set to the same growth rate
a +growth   2  # Add to the existing growth rate these numbers
''')
print(f'Dataframe:\n{res}\n\nGrowth:\n{res.pct_change()*100}\n')
```

```
Dataframe:
        A    B
2020  100  100
2021  103  100
2022  100  100
2023  100  100
2024  100  100
2025  100  100

Growth:
             A    B
2020       NaN  NaN
2021  3.000000  0.0
2022 -2.912621  0.0
2023  0.000000  0.0
2024  0.000000  0.0
2025  0.000000  0.0
```

6.1.12 Set the change in a variable to specific values (=diff operator)

$\Delta = var_t - var_{t-1} = somenumber$

Here sets the value of A in 2021 to 2 more than the value of 2020, and the 2022 value as 4 more than the **revised** value of 2021.

The second line creates a new variable "UPBY2" to the data frame and sets it equal to 100 for all periods,

The third line adds 2 to the previous periods value UPBY2.

```
df.upd('''
< 2021 2022> A =diff  2 4   # Same number of values as years
<2020 > UpBy2 = 100 # sets rows equal to the same  number for all years in between␣
↪start and end
<2021 2025> UpBy2 =diff  2

''')
```

```
        A    B  UPBY2
2020  100  100  100.0
2021  102  100  102.0
2022  106  100  104.0
2023  100  100  106.0
2024  100  100  108.0
2025  100  100  110.0
```

6.1.13 Recall that we have not overwritten df, so the df dataframe is unchanged.

```
df
```

```
        A    B
2020  100  100
2021  100  100
2022  100  100
2023  100  100
2024  100  100
2025  100  100
```

Note: The method `.upd()` only operates on on variable. A command like `.upd('A = B')` would not work. For these kind of functions, use `.mfcalc()` (see next section).

6.1.14 Keep growth rates after the update time – the –kg option

In a long projection it can sometime be useful to be able to update variables for which new information is available, but for the subsequent periods keep the growth rate the same as before the update. In database management this is frequently done when two time-series with different levels are spliced together.

The -kg or –keep_growth option instructs modelview to calculate the growth rate of the existing pre-change series, and then use it to preserve the pre-change growth rates of the series for the periods that were **not** changed.

This allows to update variables for which new information is available, but keep the growth rate the same as before the update in the period after the update time.

The default keep_growth behaviour

The `upd()` method has a parameter `keep_growth`, which by default is equal to `False`.

`keep_growth` determines how data in the time periods after those where an update is executed are treated.

If `keep_growth` is `False` then data in the sub-period after a change is left unchanged.

if `keep_growth` is set to "`True`" then the system will preserve the pre-change growth rate of the affected variable in the time period *after the change*.

Note: At the line level:

- `keep_growth=True` can be expressed as –kg
- `keep_growth=False` can be expressed as –nkg

Let's see this in a concrete example. Consider the following `dataframe` df with two variables A and B, that each grow by 2% per period, with A initialized at a level of 100 and B at a level of 110 so that we can see each separately on a graph.

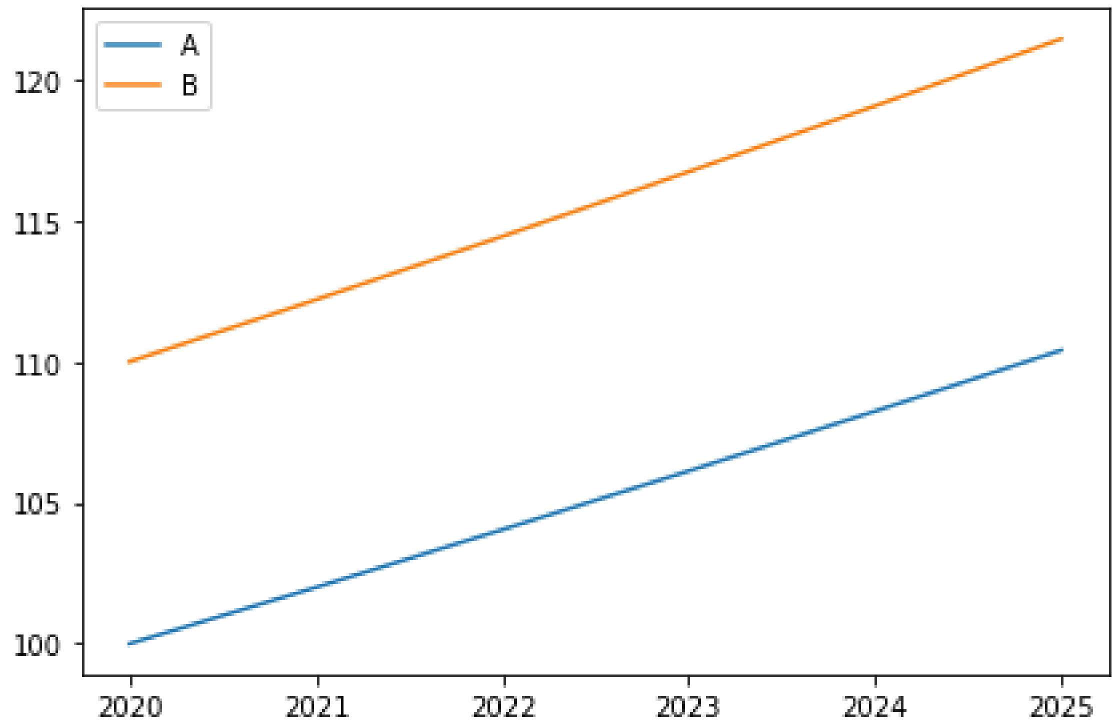
```
df = pd.DataFrame(100,
       index=[2020+v for v in range(number_of_rows)], # create row index
       # equivalent to index=[2020,2021,2022,2023,2024,2025]
       columns=['A','B'])

df=df.upd("""<2021 -1> A =growth 2
            <2020 -1>   B = 110
           <2021 -1>    B =growth 2
           """)
# Store these variables for later use in comparisons
df['A_ORIG']=df['A']
df['B_ORIG']=df['B']
df
```

	A	B	A_ORIG	B_ORIG
2020	100.000000	110.000000	100.000000	110.000000
2021	102.000000	112.200000	102.000000	112.200000
2022	104.040000	114.444000	104.040000	114.444000
2023	106.120800	116.732880	106.120800	116.732880
2024	108.243216	119.067538	108.243216	119.067538
2025	110.408080	121.448888	110.408080	121.448888

```
df[['A','B']].plot()
```

```
<AxesSubplot:>
```

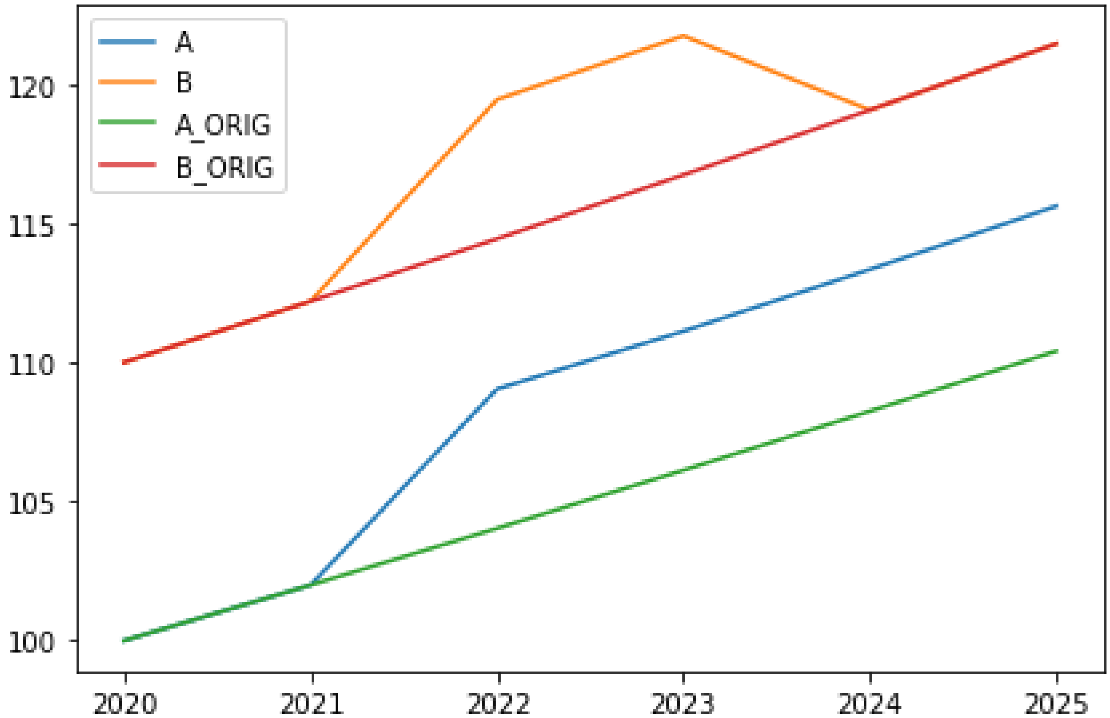


Now lets modify each by adding 5 to the level in 2022 and 2023. For B we will do setting the keep_growth option as False and for 'B' keep_growth positive. While the keep_growth is a global variable it can be set at the line level also using the –kg option (`keep_growth=True`) and –nkg option (`keep_growth=False`).

```
df=df.upd("""
            <2022 2023> A + 5 --kg
            <2022 2023> B + 5 --nkg
            """)

df[['A','B','A_ORIG','B_ORIG']].plot()
```

```
<AxesSubplot:>
```



In the first example 'A' (the green and blue lines) the level of A is increased by 5 for two periods (2021-2022). The subsequent values are also increased and they were calculated to maintain the growth rate of the original series.

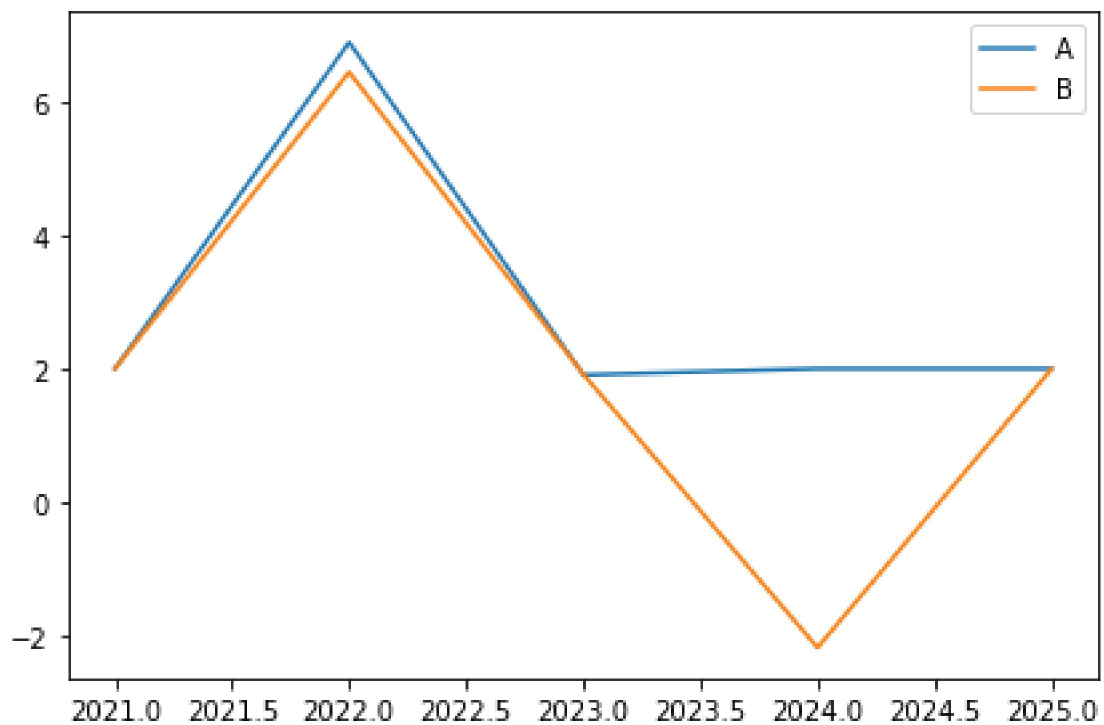
For the 'B' variable the same level change was input but because of the `--nkg` (equivalent to `keep_growth=False`) the periods after the change were unaffected and retained their old values.

Below are plots the growth rates of the two transformed series.

Here the growth in both series accelerates in 2022, by slightly less than 5 percentage points because a) the base of each is more than 100, with the base of B being higher (it was initialized at 110). In 2023 the growth rate of A returns to 2 percent, while the growth rate of B is actually negative because the level (see earlier graph) has fallen back to its original level.

```
dfg=df[['A','B']].pct_change()*100
dfg.plot()
```

```
<AxesSubplot:>
```



6.1.15 .upd(,,,keep_growth) some more examples

6.1.16 Initialize a new dataframe First make a dataframe with some growth rate

```
# instantiate a new dataframe with one column 'A' with avlue 100 everywhere and index␣
↪2020-2025
dftest = pd.DataFrame(100,
       index=[2020+v for v in range(number_of_rows)], # create row index
       # equivalent to index=[2020,2021,2022,2023,2024,2025]
       columns=['A'])                                 # create column name

# Update a to have growth rate accelerationg linearly by 1 from 1 oercent to 5 percent
original = dftest.upd('<2021 2025> a =growth 1 2 3 4 5')
print(f'Levels:\n{original}\n\nGrowth:\n{original.pct_change()*100}\n')
```

```
Levels:
               A
2020  100.000000
2021  101.000000
2022  103.020000
2023  106.110600
2024  110.355024
2025  115.872775

Growth:
        A
2020  NaN
2021  1.0
```

(continues on next page)

(continued from previous page)

```
2022  2.0
2023  3.0
2024  4.0
2025  5.0
```

6.1.17 now update A in 2021 to 2023 to a new value

Below performs the same operation, the first time the updated value is assigned to the `dataframe` `nkg` and the default behaviour of `keep_growth` is `False`

In the second example the `-kg` line option is specified, telling modelflow to maintain the growth rates of the dependent variable in the periods after the update is executed.

```
nokg = original.upd('''
<2021 2025>  a =growth 1 2 3 4 5
<2021 2023>  a = 120
''',lprint=0)

kg = original.upd('''
<2021 2025>  a =growth 1 2 3 4 5
<2021 2023>  a = 120  --kg
''',lprint=0)


kg=kg.rename(columns={"A":"KG"})        #rename cols to facilitate display
nokg=nokg.rename(columns={"A":"NOKG"}) #rename cols to facilitate display

combo=pd.concat([kg,nokg], axis=1)
combo


print(f'Levels\n{combo}\n\nGrowth\n{combo.pct_change()*100}')
```

```
Levels
          KG        NOKG
2020  100.00  100.000000
2021  120.00  120.000000
2022  120.00  120.000000
2023  120.00  120.000000
2024  124.80  110.355024
2025  131.04  115.872775

Growth
        KG      NOKG
2020   NaN       NaN
2021  20.0  20.00000
2022   0.0   0.00000
2023   0.0   0.00000
2024   4.0  -8.03748
2025   5.0   5.00000
```

Note: In the first example where KG (keep_growth) **was not set**, because the level was set constant for three periods at 120 the rate of growth was 0 for the final two years of the set period. But following this update, the level of A in 2023 is 120. With keep_Growth=False (its default value)m the level of A in 2024 remains at its unchanged unchanged (lower) level of 100.35. As a result, the growth rate in 2024 is negative.

In the **–kg** example, the pre-exsting growth rate (of 4%) is applied to the new value of 120 and so the level in 2024 is (120*1.04)=124.8 and 2025 is 131.04.

.upd() with the option keep_growth set globally

Above the line level option --keep_growth or --kg was used to keep the growth rate(or not) for a given operation.

This works because by default the option Keep_growth is set to false, implementing --kg at the line level temporarily set the keep_growth flag to true for the specific line (and those following).

The keep_growth flag can also be set globally for all the lines by setting the option in the command line.

keep_growth=True.

Now as default, all lines will keep the growth rate (unless overridden at the line level with --nkg or --no_keep_growth.

- c,d are updated in 2022 and 2023 and keep the growth rates afterwards
- e the --no_keep_growth in this line prevents the updating 2024-2025

```
# Create a data frame
dftest = pd.DataFrame(100,
       index=[2020+v for v in range(number_of_rows)], # create row index
       # equivalent to index=[2020,2021,2022,2023,2024,2025]
       columns=['A','B','C','D','E'])                        # create column␣
↪name
df
```

```
                 A           B      A_ORIG      B_ORIG
2020  100.000000  110.000000  100.000000  110.000000
2021  102.000000  112.200000  102.000000  112.200000
2022  109.040000  119.444000  104.040000  114.444000
2023  111.120800  121.732880  106.120800  116.732880
2024  113.343216  119.067538  108.243216  119.067538
2025  115.610080  121.448888  110.408080  121.448888
```

```
dfres = dftest.upd('''
<2022 2023> c = 200
<2022 2023> d = 300
<2022 2023> e = 400  --no_keep_growth
''',keep_growth=True)  # <=  Set keep_growth to True for the entirety of the command,
                       # except for e where it is overridden by the --no_keep_growth␣
↪flag
print(f'Dataframe:\n{dfres}\n\nGrowth:\n{dfres.pct_change()*100}\n')
```

```
Dataframe:
        A    B      C      D    E
2020  100  100  100.0  100.0  100
2021  100  100  100.0  100.0  100
2022  100  100  200.0  300.0  400
2023  100  100  200.0  300.0  400
```

(continues on next page)

(continued from previous page)

```
2024  100  100  200.0  300.0  100
2025  100  100  200.0  300.0  100

Growth:
        A    B      C      D      E
2020  NaN  NaN    NaN    NaN    NaN
2021  0.0  0.0    0.0    0.0    0.0
2022  0.0  0.0  100.0  200.0  300.0
2023  0.0  0.0    0.0    0.0    0.0
2024  0.0  0.0    0.0    0.0  -75.0
2025  0.0  0.0    0.0    0.0    0.0
```

6.1.18 Update several variable in one line

Sometime there is a need to update several variable with the same value over the same time frame. To ease this case .update can accept several variables in one line

```
df.upd('''
<2022 2024> h i j k =      40      # earlier values are set to zero by default
<2020>      p q r s =       1000   # All values beginning in 2020 set to 1000
<2021 -1>   p q r s =growth 2      # -1 indicates the last year of dataframe
''')
```

```
               A           B      A_ORIG      B_ORIG     H     I     J     K  \
2020  100.000000  110.000000  100.000000  110.000000   0.0   0.0   0.0   0.0
2021  102.000000  112.200000  102.000000  112.200000   0.0   0.0   0.0   0.0
2022  109.040000  119.444000  104.040000  114.444000  40.0  40.0  40.0  40.0
2023  111.120800  121.732880  106.120800  116.732880  40.0  40.0  40.0  40.0
2024  113.343216  119.067538  108.243216  119.067538  40.0  40.0  40.0  40.0
2025  115.610080  121.448888  110.408080  121.448888   0.0   0.0   0.0   0.0

                P            Q            R            S
2020  1000.000000  1000.000000  1000.000000  1000.000000
2021  1020.000000  1020.000000  1020.000000  1020.000000
2022  1040.400000  1040.400000  1040.400000  1040.400000
2023  1061.208000  1061.208000  1061.208000  1061.208000
2024  1082.432160  1082.432160  1082.432160  1082.432160
2025  1104.080803  1104.080803  1104.080803  1104.080803
```

6.1.19 .upd(„scale=<number, default=1>) Scale the updates

When running a scenario it can be useful to be able to create a number of scenarios based on one update but with different scale.

This can be particularly useful when we want to do sensitivity analyses of model results, depending on how heavily a shocked variable is hit

When using the scale option, scale=0 the baseline while scale=0.5 is a scenario half the severity.

In the example below the values of the dataframes are printed. We use the scale option (setting to to 0, 0.5 and 1) to run three scenarios using the same code but where the update in each case is multiplied by either 0, 0.5 or 1.

Note: Here we are just printing the outputs, a more interesting example would involve the solving a model using different levels of a given shock.

```
print(f'input dataframe: \n{df}\n\n')
for severity in [0,0.5,1]:
    # First make a dataframe with some growth rate
    res = df.upd('''
    <2021 2025>
    a =growth 1 2 3 4 5
    b + 10
    ''',scale=severity)
    print(f'{severity=}\nDataframe:\n{res}\n\nGrowth:\n{res.pct_change()*100}\n\n')
    #
    # Here the updated dataframe is only printed.
    # A more realistic use case is to simulate a model like this:
    # dummy_ = mpak(res,keep='Severity {serverity}')    # more realistic
```

```
input dataframe:
               A           B       A_ORIG       B_ORIG
2020  100.000000  110.000000  100.000000  110.000000
2021  102.000000  112.200000  102.000000  112.200000
2022  109.040000  119.444000  104.040000  114.444000
2023  111.120800  121.732880  106.120800  116.732880
2024  113.343216  119.067538  108.243216  119.067538
2025  115.610080  121.448888  110.408080  121.448888


severity=0
Dataframe:
          A           B      A_ORIG       B_ORIG
2020  100.0  110.000000  100.000000  110.000000
2021  100.0  112.200000  102.000000  112.200000
2022  100.0  119.444000  104.040000  114.444000
2023  100.0  121.732880  106.120800  116.732880
2024  100.0  119.067538  108.243216  119.067538
2025  100.0  121.448888  110.408080  121.448888

Growth:
        A         B  A_ORIG  B_ORIG
2020  NaN       NaN     NaN     NaN
2021  0.0  2.000000     2.0     2.0
2022  0.0  6.456328     2.0     2.0
2023  0.0  1.916279     2.0     2.0
2024  0.0 -2.189501     2.0     2.0
2025  0.0  2.000000     2.0     2.0


severity=0.5
Dataframe:
               A           B       A_ORIG       B_ORIG
2020  100.000000  110.000000  100.000000  110.000000
2021  100.500000  117.200000  102.000000  112.200000
2022  101.505000  124.444000  104.040000  114.444000
2023  103.027575  126.732880  106.120800  116.732880
2024  105.088126  124.067538  108.243216  119.067538
```

(continues on next page)

(continued from previous page)

```
2025  107.715330  126.448888  110.408080  121.448888

Growth:
       A         B  A_ORIG  B_ORIG
2020  NaN       NaN     NaN     NaN
2021  0.5  6.545455     2.0     2.0
2022  1.0  6.180887     2.0     2.0
2023  1.5  1.839285     2.0     2.0
2024  2.0 -2.103118     2.0     2.0
2025  2.5  1.919399     2.0     2.0


severity=1
Dataframe:
               A           B      A_ORIG      B_ORIG
2020  100.000000  110.000000  100.000000  110.000000
2021  101.000000  122.200000  102.000000  112.200000
2022  103.020000  129.444000  104.040000  114.444000
2023  106.110600  131.732880  106.120800  116.732880
2024  110.355024  129.067538  108.243216  119.067538
2025  115.872775  131.448888  110.408080  121.448888

Growth:
       A          B  A_ORIG  B_ORIG
2020  NaN        NaN     NaN     NaN
2021  1.0  11.090909     2.0     2.0
2022  2.0   5.927987     2.0     2.0
2023  3.0   1.768240     2.0     2.0
2024  4.0  -2.023293     2.0     2.0
2025  5.0   1.845042     2.0     2.0
```

6.1.20 .upd(„lprint=True) prints values the before and after update

The `lPrint` option of the method `upd()` is by defualt = `False`. By setting it true an update command will output the results of the calculation comapriong the values of the dataframe (over the impacted period) before, after and the difference between the two.

```
df.upd('''
# Same number of values as years
<2021 2022> A *  42 44
''',lprint=1)
```

```
Update * [42.0, 44.0] 2021 2022
A                    Before                After                Diff
2021               102.0000            4284.0000           4182.0000
2022               109.0400            4797.7600           4688.7200
```

```
                A           B      A_ORIG      B_ORIG
2020   100.000000  110.000000  100.000000  110.000000
2021  4284.000000  112.200000  102.000000  112.200000
2022  4797.760000  119.444000  104.040000  114.444000
2023   111.120800  121.732880  106.120800  116.732880
```

(continues on next page)

(continued from previous page)

```
2024   113.343216  119.067538  108.243216  119.067538
2025   115.610080  121.448888  110.408080  121.448888
```

6.1.21 .upd(,,create=True) Requires the variable to exist

Until now .upd has created variables if they did not exist in the input dataframe.

To catch misspellings the parameter `create` can be set to False. New variables will not be created, and an exception will be raised.

Here Pythons exception handling is used, so the notebook will continue to run the cells below.

```
try:
    xx = df.upd('''
    # Same number of values as years
    <2021 2022> Aa *  42 44
    ''',create=False)
    print(xx)
except Exception as inst:
    xx = None
    print(inst)
```

```
Variable to update not found:AA, timespan = [2021 2022]
Set create=True if you want the variable created:
```

6.2 `.mfcalc()` an extension of standard Pandas

Like `.upd()`, the `.mfcalc()` method can be used to extend the functionality of standard pandas. It is actually a much more powerful method that can be used to solve models or mini-models or see how modelflow normalizes equations. It can be particularly useful when creating scenarios – uses that are presented elsewhere.

Here , the focus is but is on using `mfcalc()` to perform quick and dirty calculations and modify datafames.

6.2.1 workspace initialization

Setting up our python session to use pandas and modelflow by importing their packages. `modelmf` is an extension of dataframes that is part of the modelflow installation package (and also used by modelflow itself).

```
import pandas as pd  # Python data science library
import modelmf       # Add useful features to pandas dataframes
                     # using utlities initially developed for modelflow
```

6.2.2 Create a simple dataframe

Create a Pandas dataframe with one column with the name A and 6 rows.

Set set the index to 2020 through 2026 and set the values of all the cells to 100.

- `pd.DataFrame` creates a dataframe Description
- The expression `[v for v in range(2020,2026)]` dynamically creates a python list, and fills it with integers beginning with 2020 and ending 2025

```
df = pd.DataFrame(                                    # call the dataframe constructure
    100.000,                                              # the values
    index=[v for v in range(2020,2026)],             #index
    columns=['A']                                     # the column name
                 )
df   # the result of the last statement is displayed in the output cell
```

```
          A
2020  100.0
2021  100.0
2022  100.0
2023  100.0
2024  100.0
2025  100.0
```

6.2.3 `.mfcalc()` example to calculate a new series

Use mfcalc to calculate a new column (series) as a function of the existing A column series

The below call creates a new column x.

```
df.mfcalc('x = x(-1) + a')
```

```
* Take care. Lags or leads in the equations, mfcalc run for 2021 to 2022
```

```
          A      X
2020  100.0    0.0
2021  100.0  100.0
2022  100.0  200.0
2023  100.0  300.0
2024  100.0  400.0
2025  100.0  500.0
```

> **Warning:** By default `.mfcalc` will initialize a new variable with zeroes.
>
> Moreover, if a formula passed to `.mfcalc` contains a lag a value will be calculated for the a row only if there is data in the series for the preceding row.
>
> These two behaviors affects how calculations generated with `.mfcalc` are executed and can generate results that may sometimes by unexpected.

The initialization of new variables with zero and the treatment of lags combined means that when the command `df.mfcalc('x = x(-1) + a')` is executed, the value for X in 2020 will be zero (not n/a). This results because there was no X variable defined for 2019 (no such row exists). `modelflow` first initializes all values of X with zero. It then goes to calculate X in 2020. There is no X value for 2019 so it skips ahead to 2021 and calculates X as equal to 0 (the value of x in 2020) + the value for a in 2021 – etc.

```
df
```

```
          A
2020  100.0
2021  100.0
2022  100.0
2023  100.0
2024  100.0
2025  100.0
```

6.2.4 Storing the result of an `.mfcalc()` call

Above the results of the `.mfcalc()` operation was not assigned to an object – the `DataFrame` object df itself was not changed.

Below the results of the same operation are assigned to the variable df2 and therefore stored.

```
df2=df.mfcalc('x = x(-1) + a') # Assign the result to df2
df2
```

```
* Take care. Lags or leads in the equations, mfcalc run for 2021 to 2022
```

```
          A      X
2020  100.0    0.0
2021  100.0  100.0
2022  100.0  200.0
2023  100.0  300.0
2024  100.0  400.0
2025  100.0  500.0
```

6.2.5 Recalculate A so it grows by 2 percent

`mfcalc()` knows that it can not start to calculate in 2020 A (the lagged variable) has no value in 2019.

`.mfcalc()` therefore begins its calculation in 2021. Note, the existing value for 2020 is preserved. This behaviour differs from other programs that might return a n/a value for the 2020.

```
res = df.mfcalc('a =  1.02 *  a(-1)')
res
```

```
* Take care. Lags or leads in the equations, mfcalc run for 2021 to 2022
```

```
               A
2020  100.000000
2021  102.000000
2022  104.040000
```

(continues on next page)

(continued from previous page)

```
2023  106.120800
2024  108.243216
2025  110.408080
```

```
res.pct_change()*100 # to display the percent changes
```

```
        A
2020  NaN
2021  2.0
2022  2.0
2023  2.0
2024  2.0
2025  2.0
```

6.2.6 `.mfcalc()` - the showeq option

The `showeq` option is by default = `False`.

By setting equal to `True`, mfcalc can be used to express the normalization of an entered equation.

```
df.mfcalc('dlog( a) =  0.02',showeq=1);
```

```
* Take care. Lags or leads in the equations, mfcalc run for 2021 to 2022
FRML <> A=EXP(LOG(A(-1))+0.02)$
```

In `modelflow` the expression `dlog(a)` refers to the difference in the natural logarithm $dlog(x_t) \equiv ln(x_t) - ln(x_{t-1})$ and is equal to the growth rate for the variable.

`.mfcalc()` normalizes the equation such that the systems solves for a as follows:

$$\begin{aligned}
dlog(a) &= 0.02 \\
log(a) - log(a_{t-1}) &= .02 \\
log(a) &= log(a_{t-1}) + .02 \\
a &= e^{log(a_{t-1})+0.02} \\
a &= a_{t-1} * e^{0.02}
\end{aligned}$$

which expressed in the business logic language of `modelflow` is:

A=EXP(LOG(A(-1))+0.02)

6.2.7 Using the diff() operator with mfcalc

The diff() operator, effectively normalizes to an equation that will add the value to the right of the equals sign to the lagged variable inserted in the diff operator. Thus, diff(a)=x normalizes to a=a(-1)+x

```
df.mfcalc('diff(a) =  2',showeq=1)
```

```
* Take care. Lags or leads in the equations, mfcalc run for 2021 to 2022
FRML <> A=A(-1)+(2)$
```

```
          A
2020  100.0
2021  102.0
2022  104.0
2023  106.0
2024  108.0
2025  110.0
```

6.2.8 mfcalc with several equations and arguments

In addition to a single equation multiple commands can be executed with one command.

However, **be careful** because the equation commands are executed simultaneously, which, combined with the treatments of lags, means that results may differ from what they would be if the commands were run sequentially.

For example:

```
res = df.mfcalc('''
diff(a) =  2
x = a + 42
''')

res

# use res.diff() to see the difference
```

```
* Take care. Lags or leads in the equations, mfcalc run for 2021 to 2022
```

```
          A      X
2020  100.0    0.0
2021  102.0  144.0
2022  104.0  146.0
2023  106.0  148.0
2024  108.0  150.0
2025  110.0  152.0
```

In this example the `DataFarame` df was initialized to 100 for the period 2020 through 2025.

The first line of the `.mfcalc()` routine produces results only for the period 2021 - 2025 because there is no value for `a` in 2019. The value of a in 2020 is unchanged, and the following values rise by 2 in each period.

When calculating X however, `.mfcalc` does not use the final result of the calculation of `A`, but the intermediate result (the values for 2021 through 2025.

As a result, it is this series that is passed to the second question which adds 42 to that result.

X in 2020 is not 142 as one might have expected but zero, the value to which the newly created variable defaults.

Compare the results above with the results (below) when the two steps are now undertaken in two separate calls to `.mfcalc()`.

```
res1 = df.mfcalc('''
diff(a) =  2
''')

res2 = res1.mfcalc('''
x = a + 42
''')
res2
```

```
* Take care. Lags or leads in the equations, mfcalc run for 2021 to 2022
```

```
          A      X
2020  100.0  142.0
2021  102.0  144.0
2022  104.0  146.0
2023  106.0  148.0
2024  108.0  150.0
2025  110.0  152.0
```

Danger: In `.mfcalc()`, when there are multiple equation commands in a single call, they are executed simultaneously. This, combined with `mfcalc`'s treatments of lags, means only the results of the lagged calculation will be passed to other commands equations defined in the `.mfcalc` command. As a consequence, results may differ from what would be expected and what would be seen if the two commands were run sequentially.

6.2.9 Setting a time frame with mfcalc.

It can useful in some circumstances to limit the time frame for which the calculations are performed. Specifying a start date and end date enclosed in <> in a line restricts the time period over which subsequent calculations are performed.

In the example below zeroes are generated for x prior to 2023 when the expressions are executed.

```
res = df.mfcalc('''
<2023 2025>
diff(a) =  2
x = a + 42
''')

res.diff()

res
```

```
          A      X
2020  100.0    0.0
2021  100.0    0.0
2022  100.0    0.0
```

(continues on next page)

(continued from previous page)

```
2023  102.0  144.0
2024  104.0  146.0
2025  106.0  148.0
```

Part III

Using modelflow with World Bank models

CHAPTER SEVEN

USING MODELFLOW WITH WORLD BANK MODELS

The `Modelflow` python package has been developed to solve a wide range of models, see the modelflow github web site for working examples of the Solow Model, the FR/USB model and others.

The package has been substantially expanded to include special features that enable it to work with World Bank models originally developed in EViews and designed to use EViews Model Object for simuation.

This chapter illustrates how to access these models, how to load them into a `modelflow` anaconda environment on your computer and how to perform a variety of simulations

7.1 Accessing a world bank model

At this time several World bank macrostructural models are available to download and use with `modelflow`. These include a macrostructural model for:

- Indonesia
- Nepal
- Croatia
- Iraq
- Kenya
- Bolivia

Each of these models has been developed as part of the outreach work of the World Bank. The basic modelling framework of each of these models is outlined in Burns *et al.* [2019] with specific extensions reflecting features of the individual country modelled.

This book uses as an example a climate aware model for Pakistan developed in 2020 and described in Burns *et al.* [2021] .

The World Bank models are distributed in the `pcim` file format of the `modelflow` and can be downloaded by right clicking on the links above. The Pakistan model can be downloaded here by right clicking on the above link and selecting Save Link as and placing the file on a directory accessible by your `modelflow` installation.

```
from worldbankMFModModels import pak
```

7.2 Preparing your python environment

As always, the `modelflow` and other python packages that will be used need to be imported into your python session. The examples here and this book were written and solved in a *Jupyter Notebook*. There are some Jupyter specific commands included in these examples and these are annotated. However, the bulk of the content of the programs can be run in other environments, including Interactive Development Environments (IDE) like `Spyder`or `MS Visual Code`. All the programs have been tested under `spyder` as well as Jupyter Notebook.

It is assumed that:

1. you have already installed `modelflow` and its various support packages following the instructions in Chapter xx
2. you are using Anaconda, and that
3. you have activated your `modelflow` environment by executing the following command from your python command line:

```
conda activate modelflow
```

where `modelflow` is the name you have given to the `conda` environment into which you installed `modelflow`.

```
# import the model class from modelflow package
from modelclass import model
import modelmf        # Add useful features to pandas dataframes
                      # using utlities initially developed for modelflow



model.widescreen()    # These modelflow commands ensure that outputs from modelflow␣
↪play well with Jupyter Notebook
model.scroll_off()

%load_ext autoreload
%autoreload 2
```

```
<IPython.core.display.HTML object>
```

7.3 Working with PakMod under modelflow

The basic method for working with any model is the same. Indeed the initial steps followed here are the same as were followed during the simple model discussion.

Process:

1. Prepare the workspace
2. Load the model Modelflow
3. Design some scenarios
4. Simulate the model
5. Visualize the results

7.3.1 Load a pre-existing model, data and descriptions

To load a model use the `model.modelload()` method of `modelflow`.

The command below

```
mpak,bline = model.modelload('C:\mflow\modelflow-manual\papers\mfbook\content\models\
↪pak.pcim', alfa=0.7,run=1,keep= 'Baseline')
```

instantiates (creates an instance of) a `modelflow model` object and assigns it to the variable name mpak. The `run=1` option executes the model and assigns the result of the model execution to the dataframe `bline`. The model is solved with the parameter alfa set to 0.7. The $alfa \in (0,1)$ parameter determines the step size of the solution engine. The larger alfa the larger the step size. Larger step sizes solve faster, but may have trouble finding a unique solution. Smaller step sizes take longer to solve but are more likely to find a unique solution. Values of alfa=.7 work well for World Bank models.

The `keep` option instructs `modelflow` to maintain in the model object (`mpak`) the results of the intitial scenario, assigning it the text name `Baseline`.

```
#Replace the path below with the location of the pak.pcim file on your computer
mpak,bline = model.modelload('C:\mflow\modelflow-manual\papers\mfbook\content\models\
↪pak.pcim', \
                                    alfa=0.7,run=1,keep= 'Baseline')
```

```
file read:  C:\mflow\modelflow-manual\papers\mfbook\content\models\pak.pcim
```

Note: the variable `bline` contains the dataframe with the results of the simulation. This is distinct from the data that is stored by the `kept=` command. That said, the data associated with each, while stored separately, have the same numerical values. The `keep` option is described in more detail toward the end of this section.

Box [^BoxWBMnemonics]: World Bank Mnemonics

A typical World Bank model will have in excess of 300 variables. Each has a mnemonic that is structured in a specific way, The root for almost all are 14 characters long (some special variables have additional characters appended to this root) (see discussion in section).

12345678901234

CCCAAMMMNNNNUC

where:

Letters	Meaning
CCC	The three-leter ISO code for a country – i.e. IDN for Indonesia, RUS for Russia
AA	The two-letter major accounting system to which the variable attaches,
MMM	The three-letter major sub-category of the data - i.e. GDP, EXP - expenditure
NNNN	The four-letter minor sub-category MKTP for market prices
U	The measure (K: real variable;C: Current Values; X: Prices)
C	denotes the Currency (N: National currency; D: USD; P: PPP)

Common major accounting systems mnemonics: the, AAs from above:

Code	Meaning
NY	National income
NE	National expenditure Accounts
NV	Value added accounts
GG	General Government Accounts
BX	Balance of Payments: Exports
BM	Balance of Payments: Imports
BN	Balance of Payments: Net
BF	Balance of Payments: Financial Account

Thus

Mnemonic	Meaning
IDNNYGDPMKTPKN	Indonesia GDP at market prices, real in Indonesian Rupiah
KENNECPNPRVTXN	Kenya Private (household) consumption expenditure schillings deflator
BOLGGEXPGNFSCN	Bolivia Government Expenditure on Goods and services (GNFS) in current Bolivars
HRVGGREVDCITCN	Croatia Government Revenues Direct Corporate Income Taxes in current Euros
NPLBXGSRNFSVCD	Nepal BOP Exports of non-factor services (goods and services) in current USD

7.3.2 Extracting information about the model

The newly loaded python object `mpak` is an instance of the model class and as such inherits the `methods` (functions) and `properties` (data) of that class. To learn about the model there are a variety of information methods that can be used to extract information about the model and its data.

Information about a specific

Method	Example	Information returned
`.des`	`modelname['PAKNECONPRVTXN'].des`	Dictionary of mnemonics and their variable descriptions
`.desc`	`modelname['PAKNECONPRVTXN'].desc`	List of variable description alone

Note: **Wildcards** The `*` character in the command `mpak['PAKNECON*XN'].names` is a `wildcard` character and the extopression will return all variables that begin PAKNECON and end XN. the `?` is another wildcard expression. It will match only single characters. Thus `mpak['PAKNECONPRVT?N'].names` would return three variables: `PAKNECONPRVTKN`, `PAKNECONPRVTXN`, and `PAKNECONPRVTXN`. The real, current value, and deflators for household consumption expenditure.

Information about a number of variables that meet certain search criteria

The above functions can be used in conjunction with a wildcard specification to extract the same information about a number of variables that meet the criteria. To extract a list of all variables matching a pattern, we can use same methods.

Wildcards

The ***** operator matches multiple characters, the **?** operator matches just one character

Method	Example	Information returned
.des	modelname.['*partialname*'].des	Returns Dictionary of all mnemonic and variable descriptions whose mnemonic matches
.desc	modelname.['*partialname'].desc	Returns list of variable descriptions whose mnemonic matches
.names	modelname.['*partialname'].names	Returns list of variable mnemonics that match

The ! operator If a wildcard is preceded by an exclamation mark **!** the search will be done over the description of variables instead of the mnemonic

Method	Example	Information returned
.des	modelname.['!*GDP*'].des	Returns Dictionary of all mnemonic and variable descriptions whose description contains the string GDP
.desc	modelname.['!*Consumption*'].desc	Returns list of variable descriptions whose description contains the string Consumption
.names	modelname.['!*Agriculture*'].names	Returns list of variable mnemonics whose description contains the string Agriculture

#Operator The # operator passes a predefined list to the search function and returns variable info about the variables in the list

Method	Example	Information returned
.des	modelname.['#MyList'].des	Returns Dictionary of all mnemonic and variable descriptions of the variables contained in the list MyList
.desc	modelname.['#MyList'].desc	Returns list of variable descriptions of variables contained in the list MyList
.names	modelname.['#MyList'].names	Returns list of variable mnemonics of variables contained in the list MyList

Some examples

Return all variables that begin `PAKNECON` and end `KN`.

```
mpak['PAKNECON*XN'].names
```

```
['PAKNECONENGYXN', 'PAKNECONGOVTXN', 'PAKNECONOTHRXN', 'PAKNECONPRVTXN']
```

Return a dictionary comprised of the mnemonics and the descriptions of all the variables that begin `PAKNECONPRVT` and end `N`, but have one character between the T and the N.

```
mpak['PAKNECONPRVT?N'].des
```

```
PAKNECONPRVTCN : Pvt. Cons., LCU mn
PAKNECONPRVTKN : HH. Cons Real
PAKNECONPRVTXN : Implicit LCU defl., Pvt. Cons., 2000 = 1
```

Return a list of the full description all the variables that have the word GDP in their description.

```
mpak['!*GDP*'].des
```

```
PAKBNCABFUNDCD_       : Current Account Balance (% of GDP)
PAKGDPPCKD            : GDP per capita, 2000 US$ mn
PAKGDPPCKN            : GDP per capita, 2005 LCU mn
PAKNYGDPDISCCN        : GDP Disc., LCU mn
PAKNYGDPDISCKN        : GDP Disc., 2000 LCU mn
PAKNYGDPFCSTKN        : GDP Factor Cost Local Currency units Volumes National base␣
↪year
PAKNYGDPFCSTXN        : GDP Factor Cost Local Currency units Implicit Price␣
↪deflator
PAKNYGDPFCSTXN_A      : Add factor:GDP Factor Cost Local Currency units Implicit␣
↪Price deflator
PAKNYGDPFCSTXN_D      : Fix dummy:GDP Factor Cost Local Currency units Implicit␣
↪Price deflator
PAKNYGDPFCSTXN_FITTED : Fitted  value:GDP Factor Cost Local Currency units␣
↪Implicit Price deflator
PAKNYGDPFCSTXN_X      : Fix value:GDP Factor Cost Local Currency units Implicit␣
↪Price deflator
PAKNYGDPGAP_          : Output Gap (% of Potential GDP)
PAKNYGDPMKTPCD        : GDP, Market Prices, US$ mn
PAKNYGDPMKTPCN        : GDP, Market Prices, LCU mn
PAKNYGDPMKTPKD        : GDP, Market Prices, 2000 US$ mn
PAKNYGDPMKTPKN        : Real GDP
PAKNYGDPMKTPXN        : GDP, Marker Prices, LCU Price defl., 2000 = 1
```

Return a dictionary comprised of the variable name and description if all variables in a list.

```
#Why does this not work?
mylist=['PAKNECONPRVTKN','PAKNECONGOVTKN','PAKNEGDIFTOTKN','PAKNEEXPGNGSKN',
↪'PAKNEIMPGNFSKN']
mpak['#mylist'].des
```

```
No grouping like this. Select from:
Headline
National income accounts
National expenditure accounts
Value added accounts
Balance of payments exports
Balance of payments exports and value added
Balance of Payments Financial Account
General government fiscal accounts
World all
PAK all
```

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
Input In [6], in <cell line: 3>()
      1 #Why does this not work?
      2 mylist=['PAKNECONPRVTKN','PAKNECONGOVTKN','PAKNEGDIFTOTKN','PAKNEEXPGNGSKN
↪','PAKNEIMPGNFSKN']
----> 3 mpak['#mylist'].des

File ~\.conda\envs\modelflow\lib\site-packages\ModelFlow-1.0.8-py3.9.egg\
↪modelclass.py:1260, in Org_model_Mixin.__getitem__(self, name)
```

(continues on next page)

(continued from previous page)

```
   1252 def __getitem__(self, name):
   1253     '''
   1254     To execute the index operator []
   1255 
   1256     Uses the :any:`modelvis.vis` operator
   1257 
   1258     '''
-> 1260     a = self.vis(name)
   1262     return a

File ~\.conda\envs\modelflow\lib\site-packages\ModelFlow-1.0.8-py3.9.egg\
↪modelclass.py:3735, in Display_Mixin.vis(self, *args, **kwargs)
   3733 if not hasattr(self, '_vis'):
   3734     self._vis = mv.vis
-> 3735 return self._vis(self, *args, **kwargs)

File ~\.conda\envs\modelflow\lib\site-packages\ModelFlow-1.0.8-py3.9.egg\modelvis.
↪py:50, in vis.__init__(self, model, pat, names, df)
     48 self.__pat__ = pat
     49 if type(names) == type(None):
---> 50     self.names = self.model.vlist(self.__pat__)
     51 else:
     52     self.names = names

File ~\.conda\envs\modelflow\lib\site-packages\ModelFlow-1.0.8-py3.9.egg\
↪modelclass.py:1107, in Org_model_Mixin.vlist(self, pat)
   1105             lf='\n'
   1106             print(f'No grouping like this. Select from: {lf}{lf.join(self.
↪var_groups.keys())}')
-> 1107             raise Exception('Try with a correct group name')
   1109 ipat = upat
   1110 # breakpoint()

Exception: Try with a correct group name
```

the method `.var_description` returns the descriptor of the associated variable. This method does not accept wild-cards.

```
mpak.var_description['PAKNYGDPMKTPKN']
```

```
'*NYGDPMKTP?N'
```

Equation information methods

There are two functions to extract the equations from a model.

Command	Effect
`mpak['PAKNECONPRVTKN'].frml`	Returns a **normalized** version of the equation (the one actually used in modelflow)
`mpak['PAKNECONPRVTKN'].eviews`	In models imported from Eviews, reports the original eviews specification

The equation for consumption in `mpak` we see that it follows something very close to this formulation.

```
mpak.PAKNECONPRVTKN.frml
```

```
Endogeneous: PAKNECONPRVTKN: HH. Cons Real
Formular: FRML <DAMP,STOC> PAKNECONPRVTKN = (PAKNECONPRVTKN(-1)*EXP(PAKNECONPRVTKN_
↪A+ (-0.2*(LOG(PAKNECONPRVTKN(-1))-LOG(1.21203101101442)-LOG((((PAKBXFSTREMTCD(-
↪1)-PAKBMFSTREMTCD(-1))*PAKPANUSATLS(-1))+PAKGGEXPTRNSCN(-1)+PAKNYYWBTOTLCN(-
↪1)*(1-PAKGGREVDRCTXN(-1)/100))/PAKNECONPRVTXN(-1)))+0.
↪763938860758873*((LOG((((PAKBXFSTREMTCD-
↪PAKBMFSTREMTCD)*PAKPANUSATLS)+PAKGGEXPTRNSCN+PAKNYYWBTOTLCN*(1-PAKGGREVDRCTXN/
↪100))/PAKNECONPRVTXN))-(LOG((((PAKBXFSTREMTCD(-1)-PAKBMFSTREMTCD(-
↪1))*PAKPANUSATLS(-1))+PAKGGEXPTRNSCN(-1)+PAKNYYWBTOTLCN(-1)*(1-PAKGGREVDRCTXN(-
↪1)/100))/PAKNECONPRVTXN(-1))))-0.0634474791568939*DURING_2009-0.
↪3*(PAKFMLBLPOLYXN/100-((LOG(PAKNECONPRVTXN))-(LOG(PAKNECONPRVTXN(-1)))))) )) *␣
↪(1-PAKNECONPRVTKN_D)+ PAKNECONPRVTKN_X*PAKNECONPRVTKN_D  $

PAKNECONPRVTKN  : HH. Cons Real
DURING_2009     :
PAKBMFSTREMTCD  : Imp., Remittances (BOP), US$ mn
PAKBXFSTREMTCD  : Exp., Remittances (BOP), US$ mn
PAKFMLBLPOLYXN  : Key Policy Interest Rate
PAKGGEXPTRNSCN  : Current Transfers
PAKGGREVDRCTXN  : Direct Revenue Tax Rate
PAKNECONPRVTKN_A: Add factor:HH. Cons Real
PAKNECONPRVTKN_D: Fix dummy:HH. Cons Real
PAKNECONPRVTKN_X: Fix value:HH. Cons Real
PAKNECONPRVTXN  : Implicit LCU defl., Pvt. Cons., 2000 = 1
PAKNYYWBTOTLCN  : Total Wage Bill
PAKPANUSATLS    : Exchange rate LCU / US$ - Pakistan
```

The `mpak['PAKNECONPRVTKN'].eviews` command returns the equations before they were normalized. In most cases this is a slightly more legible form. Here following the EViews syntax, $\Delta ln()$ is written as dlog().

```
mpak.PAKNECONPRVTKN.eviews
```

```
DLOG(PAKNECONPRVTKN) =- 0.2*(LOG(PAKNECONPRVTKN( - 1)) - LOG(1.21203101101442) -␣
↪LOG((((PAKBXFSTREMTCD( - 1) - PAKBMFSTREMTCD( - 1))*PAKPANUSATLS( - 1)) +␣
↪PAKGGEXPTRNSCN( - 1) + PAKNYYWBTOTLCN( - 1)*(1 - PAKGGREVDRCTXN( - 1)/100))/
↪PAKNECONPRVTXN( - 1))) + 0.763938860758873*DLOG((((PAKBXFSTREMTCD -␣
↪PAKBMFSTREMTCD)*PAKPANUSATLS) + PAKGGEXPTRNSCN + PAKNYYWBTOTLCN*(1 -␣
↪PAKGGREVDRCTXN/100))/PAKNECONPRVTXN) - 0.0634474791568939*@DURING("2009") - 0.
↪3*(PAKFMLBLPOLYXN/100 - DLOG(PAKNECONPRVTXN))
```

7.4 Behavioural equations in the MFMod framework

Recall a behavioural equation determines the value of an endogenous variable. For many of the variables in Wold Bank models, behavioural functions are estimated using an Error Correction Framework that splits the equation into a theoretically determined long run component and a more idiosyncratic short-run component.

Looking at the eviews representation of the consumption function:

`DLOG(PAKNECONPRVTKN) =- 0.2*(LOG(PAKNECONPRVTKN( - 1)) - LOG(1.21203101101442) - LOG((((PAKBXFSTREMTCD( - 1) - PAKBMFSTREMTCD( - 1))*PAKPANUSATLS( - 1)) + PAKGGEXPTRNSCN( - 1) + PAKNYYWBTOTLCN( - 1)*(1 - PAKGGREVDRCTXN( -`

```
1)/100))/PAKNECONPRVTXN( - 1))) + 0.763938860758873*DLOG((((PAKBXFSTREMTCD
- PAKBMFSTREMTCD)*PAKPANUSATLS) + PAKGGEXPTRNSCN + PAKNYYWBTOTLCN*(1 -
PAKGGREVDRCTXN/100))/PAKNECONPRVTXN) - 0.0634474791568939*@DURING("2009") - 0.
3*(PAKFMLBLPOLYXN/100 - DLOG(PAKNECONPRVTXN))
```

Below the mnemonics are simplified to ease reading of the equation using:

Model Mnemonic	Simplified	Meaning
PAKNECONPRVTKN	CON_t^{KN}	Household Consumption
(PAKBXFSTREMTCD - PAKBMFSTREMTCD)*PAKPANUSATLS	$Remit_t^{net}$	Net remittances inflows in LCU
PAKGGEXPTRNSCN	$TRANSF_t^{hhld}$	Government transfers to households
DURING_2010	D_t^{2010}	A dummy
PAKFMLBLPOLYXN	r_t^{policy}	Policy Rate
PAKGGREVDRCTXN	$DirectTxR_t$	Direct Taxes: Effective rate
PAKNECONPRVTKN_A	$CON_t^{KN_AF}$	Add factor:Household Consumption
PAKNECONPRVTXN	CON_t^{XN}	Household Consumption Deflator
PAKNYYWBTOTLCN	$WAGEBILL_t^{CN}$	Economy-wide wage bill

$$\begin{aligned}\Delta log(CON_t^{KN}) = & -0.2 * \Bigg[LOG(CON_{t-1}^{KN}) - LOG\bigg(\frac{(Remit_{t-1}^{net} + WAGEBILL_{t-1}^{CN} + TRANSF_{t-1}^{hhld}) * (1 - DirectTxR_{t-1}/1}{CON_{t-1}^{XN}} \\ & + 0.76 * \Delta log\bigg(\frac{(Remit_t^{net} + WAGEBILL_t^{CN} + TRANSF_t^{hhld}) * (1 - DirectTxR_t/100)}{CON_t^{XN}}\bigg) \\ & + 0.030 + 0.016 * D_t^{2010} - 0.3 * \bigg(r_t^{policy}/100 - \Delta log(CON_t^{XN})\bigg) - CON_t^{KN_AF}\end{aligned}$$

Where in this instance the short-run elasticity of consumption to disposable income is .76 , and the short run elasticity of consumption to the real interest rate is 0.3.

7.4.1 The ECM specification

Pretty sure this repeats and earlier section. Delete one

The ECM approach used in World Bank models is described in [Wickens and Breusch, 1988], and addresses the above challenge by modelling both the long run relationship and the short run short run behaviour and brings them together into one equation.

The ECM specification is therefore a single equation comprised of two parts (the long run relationship, and the short-run relationship).

Consider as an example two variables say consumption and disposable income. Both have an underlying trend or in the parlance are co-integrated to degree 1. For simplicity we call them y an x.

The short run relationship

In its simplest form we might have a short run relationship between the growth rates of our two variables such that:

$$\Delta log(Y_t) = \alpha + \beta \Delta log(X_t) + \epsilon_t$$

or substituting lower case letters for the logged values.

$$\Delta y_t = \alpha + \beta \Delta x_t + \epsilon_t$$

The long run equation

The long run relates the level of the two (or more) variables. We can write a simple version of that equation as:

$$Y_t = \alpha X_t^{\beta} + \eta_t$$

Rewriting this (in logarithms) it can be expressed as:

$$y_t = ln\ (\alpha) + \beta y_t + \eta_t$$

7.4.2 The long run equation in the steady state

First we note that in the steady state the expected value of the error term in the long run equation is zero ($\eta_t = 0$) so in those conditions we can simplify the long run relationship to:

$$y_t = ln\ (\alpha) + \beta x_t$$

or equivalently (substituting A for the log of α).

$$y_t - A - \beta x_t = 0$$

Moreover if we multiplied this by some arbitrary constant say $-\lambda$ it would still equal zero.

$$-\lambda(y_t - A - \beta x_t)$$

and in the steady state this will also be true for the lagged variables

$$-\lambda(y_{t-1\_}A - \beta x_{t-1})$$

7.5 Putting it together

From before we have the short run equation:

$$\Delta y_t = \alpha + \beta \Delta x_t + \epsilon_t$$

Inserting the steady state expression into the short run equation makes no difference (in the long run) because in the long run it is equal to zero.

$$\Delta y_t = -\lambda(y_{t-1\_}A - \beta x_{t-1}) + \alpha + \beta \Delta x_t + \epsilon_t$$

When the model is not in the steady state the expression $y_{t-1} - A - \beta x_{t-1}$ is of course the error term from the long run equation (a measure of how far the dependent variable is from equilibrium).

7.5.1 Lambda, the speed of adjustment

The parameter λ can be interpreted as the speed of adjustment. As long as λ is greater than zero and less or equal to one if there are no further disturbances ($\epsilon_t = 0$) the expression multiplied by lambda will slowly decline toward zero. How fast depends on how large or small is λ.

To be convergent λ must be between 0 and 2, if its is negative or greater than one, then the long run portion of the equation will cause the disequilibrium to grow each period (λ >1) not diminish or if (λ >1<2) output will oscillate from positive to negative ($\lambda < 0$).

Intuitively, the long run error term measures how far we are from equilibrium one period earlier (at t-1). The ECM term ensures that we will slowly converge to equilibrium – the point at which the long run equation holds exactly. If $ $\lambda isgreaterthanzerobutlessthanone(orequaltoone)someportionofthepreviousperiodyear'sdisequilibriumwillbeabsorbedeac$

Looking at an ECM equation we can then break it up into its component parts. For the consumption function it will look something like this:

$$\Delta c_t = -\lambda(\underbrace{log(C_{t-1}) - log(Wages_{t-1} - Taxes_{t-1} + Transfers_{t-1} + \alpha)}_{\text{Long run}} + \beta \underbrace{\Delta y_t}_{\text{short run}}$$

CHAPTER

EIGHT

SCENARIO ANALYSIS

An essential feature of a model is that when given a specific set of inputs (the exogenous variables to the model) it will always return the same results.

Below a new `ModelFlow` sessionis prepared, initializing a pandas session and importing and solving a saved WBG model (NB: these are precisely the same commands they used to start the previous chapter) and would form the essential initialization commands of any python session using `ModelFlow`.

```
# import the model class from modelflow package
from modelclass import model
import modelmf       # Add useful features to pandas dataframes
                     # using utlities initially developed for modelflow

model.widescreen()   # These modelflow commands ensure that outputs from modelflow␣
↪play well with Jupyter Notebook
model.scroll_off()

%load_ext autoreload
%autoreload 2


#Load a saved version of the Pakistan model and solve it,
#saving the results in the model object mpak, and the resulting dataframe in bline

#Replace the path below with the location of the pak.pcim file on your computer
mpak,bline = model.modelload('C:\mflow\modelflow-manual\papers\mfbook\content\models\
↪pak.pcim', \
                                alfa=0.7,run=1,keep= 'Baseline')
```

```
<IPython.core.display.HTML object>
```

```
file read:  C:\mflow\modelflow-manual\papers\mfbook\content\models\pak.pcim
```

As noted, when the model is solved without changing any inputs (as was the case of the load) the model should return (reproduce) exactly the same data as before[^fn2]. To test this for `mpak` the results from the simulation can be compared by using the `basedf` and `lastdf` DataFrames.

[^fn2:] If it does not, the model has violated he principle of reproducibility and there is something wrong (usually one of the identities does not hold).

Below, the percent difference between the values of the variables for real GDP and Consumer demand in the two `dataframes .basedf` and `lastdf` is zero following a simulation where the inputs were not changed – confirming the reproduction of results.

```
# Need statement to change the default format
mpak.smpl(2020,2030)
mpak['PAKNYGDPMKTPKN PAKNECONPRVTKN'].difpctlevel.mul100.df
```

```
      PAKNYGDPMKTPKN  PAKNECONPRVTKN
2020             0.0             0.0
2021             0.0             0.0
2022             0.0             0.0
2023             0.0             0.0
2024             0.0             0.0
2025             0.0             0.0
2026             0.0             0.0
2027             0.0             0.0
2028             0.0             0.0
2029             0.0             0.0
2030             0.0             0.0
```

8.1 Different kinds of simulations

The `modelflow` package performs 4 different kinds of simulation:

1. A shock to an exogenous variable in the model
2. An exogenous shock of a behavioural variable, executed by exogenizing the variable
3. An endogenous shock of a behavioural variable, executed by shocking the add factor of the variable.
4. A mixed shock of a behavioural variable, achieved by temporarily exogenixing the variable.

Although technically modelflow would allow us to shock identities, that would violate their nature as accounting rules. **Effectively such a shock would break the economic sense of the model.**

As a result, this we possibility is not discussed.

8.1.1 A shock to an exogenous variable

A World Bank model will reproduce the same values if inputs (exogenous variables) are not changed. In the simulation below, the oil price is changed – increasing by $25 for the three years between 2025 and 2027 inclusive.

As a first step a new input dataframe is created as a copy of the original and then the oil price in that data frame is modified using the `mfcalc` method to change the value for the three years in question.

Finally pandas math is used to display the initial value, the changed value and the difference between the two, confirming that the mfcalc statement revised the oil price data.

```
#Make a copy of the baseline dataframe
oilshockdf=mpak.basedf
oilshockdf=oilshockdf.mfcalc("<2025 2027> WLDFCRUDE_PETRO = WLDFCRUDE_PETRO +25")

compdf=mpak.basedf.loc[2000:2030,['WLDFCRUDE_PETRO']]
compdf['LASTDF']=oilshockdf.loc[2000:2030,['WLDFCRUDE_PETRO']]
compdf['Dif']=compdf['LASTDF']-compdf['WLDFCRUDE_PETRO']

compdf.loc[2024:2030]
```

```
      WLDFCRUDE_PETRO       LASTDF   Dif
2024        80.367180    80.367180   0.0
2025        85.336809   110.336809  25.0
2026        90.613742   115.613742  25.0
2027        96.216983   121.216983  25.0
2028       102.166709   102.166709   0.0
2029       108.484346   108.484346   0.0
2030       115.192643   115.192643   0.0
```

Running the simulation

Having created a new dataframe comprised of all the old data plus the changed data for the oil price, a simulation can now be run.

In the command below, the simulation is run from 2020 to 2040, using the `oilshockdf` as the input `DataFrame`. The results of the simulation are assigned to a new `DataFrame` named `ExogOilSimul`. The `Keep` command ensures that the mpak model object stores (keeps) a copy of the results identified by the text name *'$25 increase in oil prices 2025-27'*.

```
#Simulate the model
ExogOilSimul = mpak(oilshockdf,2020,2040,keep='$25 increase in oil prices 2025-27')
```

Results

`ModelFlow` tools can be used to visualize the impacts of the shock; as a print out; as charts and within Jupyter notebook as an interactive widget.

The display below confirms that the shock was executed as desired. The `dif.df` method returns the difference between the `.lastdf` and `.basedf` values of the selected variable(s) as a `DataFrame`. The `with mpak.set_smpl(2020,2030):` clause temporarily restricts the sample period over which the following **indented** commands are executed.

Alternatively the `mpak.smpl(2020,2030)` could be used. This would restricts the time period of over which **all** subsequent commands are executed.

```
with mpak.set_smpl(2020,2030):
    print(mpak['WLDFCRUDE_PETRO'].dif.df);
```

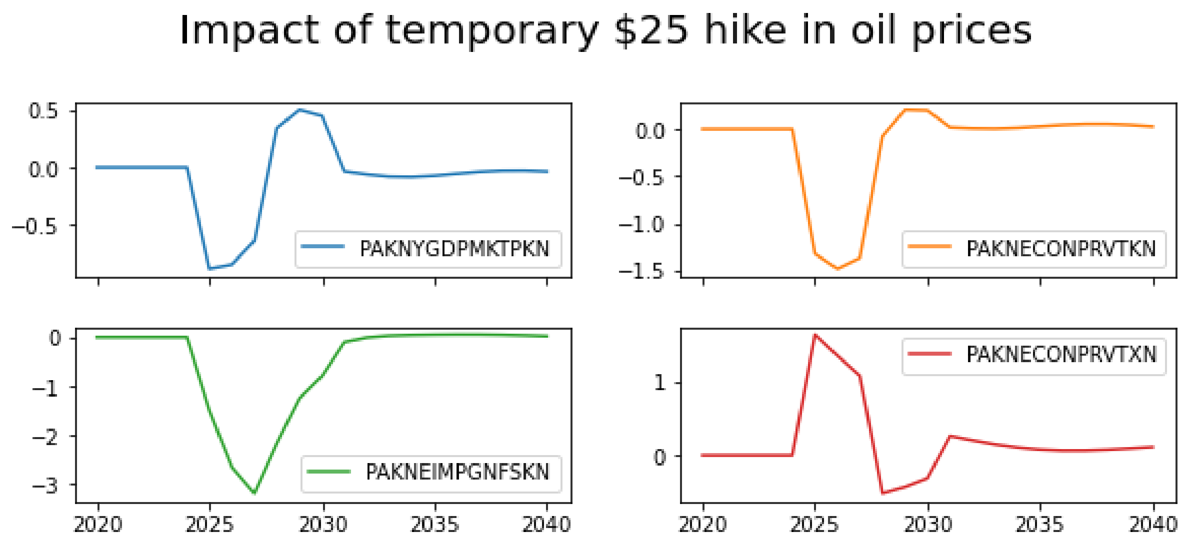
```
      WLDFCRUDE_PETRO
2020              0.0
2021              0.0
2022              0.0
2023              0.0
2024              0.0
2025             25.0
2026             25.0
2027             25.0
2028              0.0
2029              0.0
2030              0.0
```

Below the impact of this change on a few variables are expressed graphically and in a table.

The first variable `PAKNYGDPMKTPKN` is Pakistan's real GDP, the second `PAKNECONPRVTKN` is real consumption and the third is the Consumer price deflator `PAKNECONPRVTXN`.

```
mpak['PAKNYGDPMKTPKN PAKNECONPRVTKN PAKNEIMPGNFSKN PAKNECONPRVTXN'].difpctlevel.
↪mul100.plot(title="Impact of temporary $25 hike in oil prices")
```

```
C:\Users\wb268970\.conda\envs\modelflow\lib\site-packages\IPython\core\pylabtools.
↪py:151: UserWarning: This figure was using constrained_layout, but that is␣
↪incompatible with subplots_adjust and/or tight_layout; disabling constrained_
↪layout.
  fig.canvas.print_figure(bytes_io, **kw)
```




```
print(round(mpak['PAKNYGDPMKTPKN PAKNECONPRVTKN PAKNEIMPGNFSKN PAKNECONPRVTXN'].
↪difpctlevel.mul100.df,2))
```

```
      PAKNYGDPMKTPKN  PAKNECONPRVTKN  PAKNEIMPGNFSKN  PAKNECONPRVTXN
2020            0.00            0.00            0.00            0.00
2021            0.00            0.00            0.00            0.00
2022            0.00            0.00            0.00            0.00
2023            0.00            0.00            0.00            0.00
2024            0.00            0.00            0.00            0.00
2025           -0.89           -1.32           -1.49            1.64
2026           -0.85           -1.48           -2.65            1.35
2027           -0.64           -1.37           -3.19            1.08
2028            0.34           -0.08           -2.17           -0.51
2029            0.50            0.20           -1.25           -0.43
2030            0.45            0.19           -0.80           -0.31
2031           -0.04            0.02           -0.10            0.26
2032           -0.06            0.01           -0.01            0.20
2033           -0.08            0.00            0.03            0.15
2034           -0.08            0.01            0.04            0.11
2035           -0.07            0.03            0.05            0.08
2036           -0.06            0.04            0.05            0.06
2037           -0.04            0.05            0.05            0.06
2038           -0.03            0.05            0.05            0.08
2039           -0.03            0.04            0.04            0.09
2040           -0.04            0.03            0.02            0.11
```

The graphs show the change in the level as a percent of the previous level. They suggest that a temporary $25 oil price hike would reduce GDP in the first year by about 0.9 percent, that the impact would diminish by the third year to -.64 percent, and then turn positive in the fourth year when the price effect was eliminated.

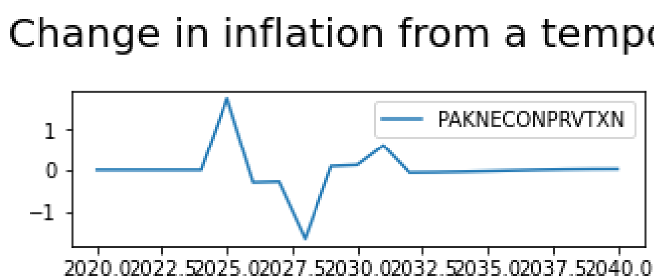
The impacts on household consumption are stronger but follow a similar pattern.

The GDP impact is smaller partly because the decline in domestic demand reduces imports. Because imports enter into the GDP identity with a negative sign. Therefore a reduction in imports actually increase aggregate GDP – or in this case partially offsets the declines coming from reduced consumption (and investment).

Finally as could be expected, initially prices rise sharply with higher oil prices. However, as the slow down in growth is felt, inflationary pressures turn negative and the overall impact on the price level turns negative. The graph and table above shows what is happening to the **price level**. To see the impact on inflation (the rate of growth of prices), a separate graph can be generated using `difpct.mul100`, which shows the change in the rate of growth of variables where the growth rate is expressed as a per cent $\left[\left(\frac{x_t^{shock}}{x_{t-1}^{shock}}-1\right)-\left(\frac{x_t^{baseline}}{x_{t-1}^{baseline}}-1\right)\right]*100$.

```
mpak['PAKNECONPRVTXN'].difpct.mul100.plot(title="Change in inflation from a temporary
↪$25 hike in oil prices")
```

```
C:\Users\wb268970\.conda\envs\modelflow\lib\site-packages\IPython\core\pylabtools.
↪py:151: UserWarning: This figure was using constrained_layout, but that is␣
↪incompatible with subplots_adjust and/or tight_layout; disabling constrained_
↪layout.
  fig.canvas.print_figure(bytes_io, **kw)
```



Ib how come this graph shows up so small. How can we affect its size?

This view, gives a more nuanced result. The inflation rate increases initially by about 1.2 percentage points, but falls compared with the baseline below in the 2026-2027 period as the as the influence of the slowdown in GDP more than offsets the continued inflationary impetus from the lagged increase in oil prices. In 2028, when oil prices drop back to their previous level, there is an additional dis-inflationary force and sharp drop in inflation as compared with the baseline. Overtime, the boost to demand from lower prices translates into an acceleration in growth and a return of inflation back to its trend rate.

8.1.2 An exogenous shock to a Behavioural variable

Behavioural equations can be de-activated by exogenizing them, either for the entire simulation period, or for a selected sub period. In this example, consumption is exogenized for the entire simulation period.

To motivate the simulation, it is assumed that a change in weather patterns has increased the number of sunny days by 10 percent. This increases households happiness and causes them to permanently increase their spending by 2.5% beginning in 2025.

Such a shock can be specified either manually or by using the `.fix()` method. Below the simpler `.fix()` method is used, but the equivalent manual steps performed by `.fix()` are also explained.

To exogenize PAKNECONPRVTKN for the entire simulation period, initially a new DataFrame Cfixed is created as a slightly modified version of mpak.basedf using the .fix() command.

Cfixed=mpak.fix(mpak.basedf,PAKNECONPRVTKN)

This does two things, that could have been done manually. First it sets the dummy variable PAKNECONPRVTKN_D=1 for the entire simulation period. Recall the consumption equation like all behavioural equations of World Banjk models implemented in ModelFlowis expressed in tow parts.

$$cons = (1 - cons_D) * \left[C'(X) \right] + cons_d * cons_x$$

When $cons_D = 1$ the first part (as it does in this scenario) the equation evaluate to zero and consumption is equal to (1)* $cons_x$. If instead (which would be the normal case $cons_d$ were set to zero. the the equation would simplify to $cons = C'(X)$

Then .fix() method then sets the variable PAKNECONPRVTKN_X in the Cfixed dataframe equal to the value of PAKNECONPRVTKN in the basedf .DataFrame. All the other variables are just copies of their values in .basedf.

With PAKNECONPRVTKN_D=1 throughout the normal behavioral equation is effectively de-activated or exogenized … $PAKNECONPRVTKN = PAKNECONPRVTKN_X$.

```
mpak.smpl() # reset the active sample period to the full model.
Cfixed=mpak.fix(bline,'PAKNECONPRVTKN')
```

For the moment, the equation is exogenized but the values have been set to the same values as the .basedf dataframe, so solving the model will not change anything.

The .upd() method can be used to implement the assumption that Real consumption (PAKNECONPRVTYKN) would be 2.5% stronger.

```
Cfixed=Cfixed.upd("<2025 2040> PAKNECONPRVTKN_X  * 1.025")
```

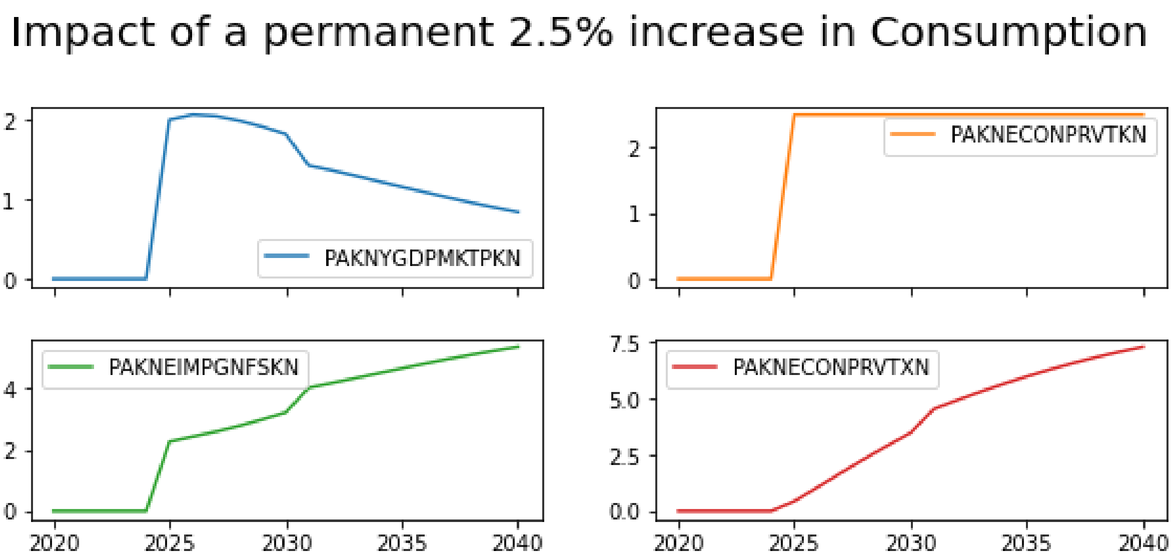
To perform the simulation, the revised CFixed DataFrame is input to the mpak model solve routine.

CFixedRes = mpak(Cfixed,2020,2040,keep='2.5% increase in C 2025-40 (fix)')

And then the results can be examined graphically as before.

```
CFixedRes = mpak(Cfixed,2020,2040,keep='2.5% increase in C 2025-40') # simulates the␣
↪model
mpak['PAKNYGDPMKTPKN PAKNECONPRVTKN PAKNEIMPGNFSKN PAKNECONPRVTXN'].difpctlevel.
↪mul100.plot(title="Impact of a permanent 2.5% increase in Consumption")
```

```
C:\Users\wb268970\.conda\envs\modelflow\lib\site-packages\IPython\core\pylabtools.
↪py:151: UserWarning: This figure was using constrained_layout, but that is␣
↪incompatible with subplots_adjust and/or tight_layout; disabling constrained_
↪layout.
  fig.canvas.print_figure(bytes_io, **kw)
```




```python
import pandas as pd
with pd.option_context('display.float_format', '{:,.2f}'.format):
    with mpak.set_smpl(2020,2040):
        print(mpak['PAKNYGDPMKTPKN PAKNECONPRVTKN PAKNEIMPGNFSKN PAKNECONPRVTXN'].
↪difpctlevel.mul100.df)
```

```
      PAKNYGDPMKTPKN  PAKNECONPRVTKN  PAKNEIMPGNFSKN  PAKNECONPRVTXN
2020            0.00            0.00            0.00            0.00
2021            0.00            0.00            0.00            0.00
2022            0.00            0.00            0.00            0.00
2023            0.00            0.00            0.00            0.00
2024            0.00            0.00            0.00            0.00
2025            0.00            0.00            0.00            0.00
2026            0.00            0.00            0.00            0.00
2027            0.00            0.00            0.00            0.00
2028            0.00            0.00            0.00            0.00
2029            0.00            0.00            0.00            0.00
2030            0.00            0.00            0.00            0.00
2031           -0.36           -0.09            0.47            0.48
2032           -0.27           -0.01            0.45            0.36
2033           -0.19            0.04            0.42            0.27
2034           -0.12            0.08            0.38            0.20
2035           -0.08            0.10            0.33            0.16
2036           -0.05            0.10            0.28            0.14
2037           -0.03            0.10            0.23            0.14
2038           -0.03            0.08            0.19            0.14
2039           -0.03            0.06            0.14            0.14
2040           -0.04            0.04            0.10            0.15
```

The permanent rise in consumption by 2.5 percent causes a temporary increase in GDP of close to 2% (1.86). Higher imports tend to diminish the effect on GDP. Over time higher prices due to the inflationary pressures caused by the additional demand cause the GDP impact to diminish to close to less than 1 percent by 2040.

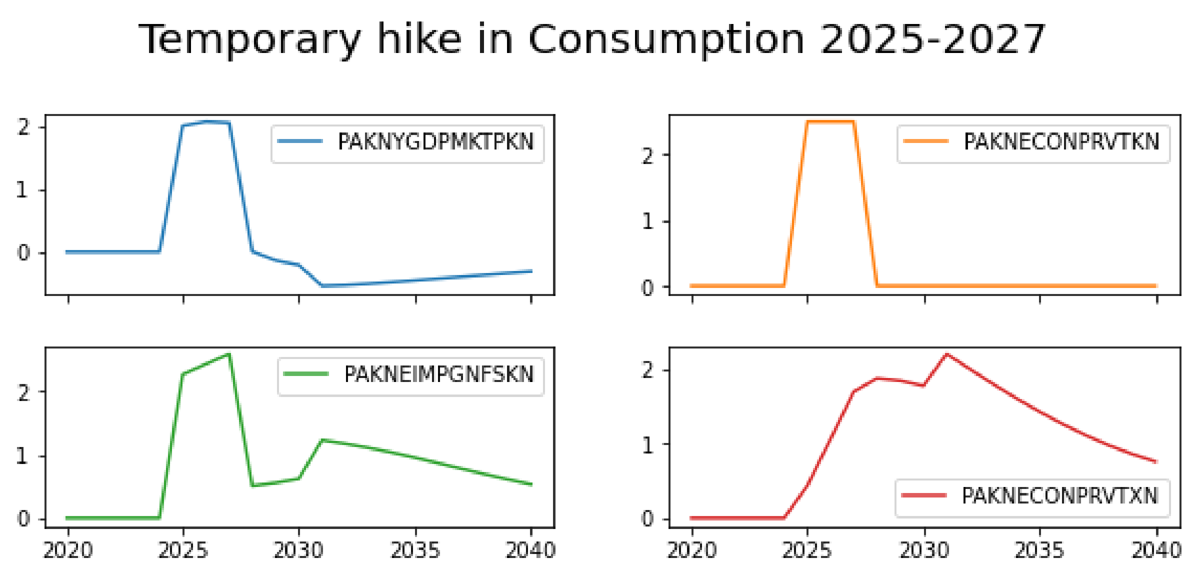
8.1.3 Temporarily exogenize a behavioural variable

The third method of formulating a scenario involves temporarily exogenizing a variable. The methodology is the same except the period for which the variable is exogenized is different.

Here the set up is basically the same as before.

```
#reset the active sample period to the full period
mpak.smpl()
# create a copy of the bline DataFrame, but setting the PAKNECONPRVTKN_D variable to␣
↪1 for the period 2025 through 2027
CTempExogAll=mpak.fix(bline,'PAKNECONPRVTKN')
# multiply the exogenized value of consumption by 2.5% for 2025 through 2027
CTempExogAll=CTempExogAll.upd("<2025 2027> PAKNECONPRVTKN_X * 1.025")

#Solve the model
CTempXAllRes = mpak(CTempExogAll,2020,2040,keep='2.5% increase in C 2025-27 -- exog␣
↪whole period') # simulates the model
mpak['PAKNYGDPMKTPKN PAKNECONPRVTKN PAKNEIMPGNFSKN PAKNECONPRVTXN'].difpctlevel.
↪mul100.plot(title="Temporary hike in Consumption 2025-2027")
```

```
C:\Users\wb268970\.conda\envs\modelflow\lib\site-packages\IPython\core\pylabtools.
↪py:151: UserWarning: This figure was using constrained_layout, but that is␣
↪incompatible with subplots_adjust and/or tight_layout; disabling constrained_
↪layout.
  fig.canvas.print_figure(bytes_io, **kw)
```



The results are quite different. GDP is boosted initially as before but when consumption drops back to its pre-shock level, GDP and imports decline sharply.

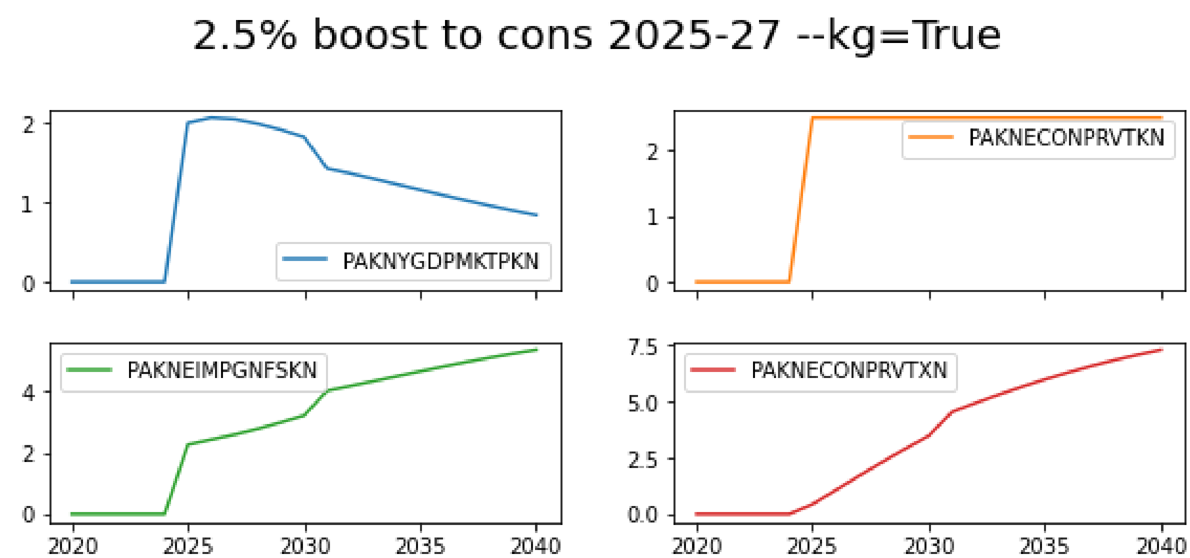
Prices (and inflation) are higher initially but when the economy starts to slow after 2025 prices actually fall (deflation). While prices are falling, the level of prices remains higher at the end of the simulation.

Temporary shock exogenized for the whole period

This scenario is the same as the previous, but this time the `--KG` (keep_growth) option is used to maintain the pre-shock growth rates of consumption in the post-shock period. Effectively this is the same as a permanent increase in the level of consumption by 2.5% because the final shocked value of consumption (which was 2.5% higher then its pre-shock level) is grown at the same pre-shock rate – ensuring that all post-shock variables are also up by 2.5%.

```
mpak.smpl() # reset the active sample period to the full model.
CTempExogAllKG=mpak.fix(bline,'PAKNECONPRVTKN')
CTempExogAllKG = CTempExogAllKG.upd('''
<2025 2027> PAKNECONPRVTKN_X * 1.025 --kg
''',lprint=0)

#Now we solve the model
CTempXAllResKG = mpak(CTempExogAllKG,2020,2040,keep='2.5% increase in C 2025-27 --
↪exog whole period --KG=True') # simulates the model
mpak['PAKNYGDPMKTPKN PAKNECONPRVTKN PAKNEIMPGNFSKN PAKNECONPRVTXN'].difpctlevel.
↪mul100.plot(title="2.5% boost to cons 2025-27 --kg=True")
```

```
C:\Users\wb268970\.conda\envs\modelflow\lib\site-packages\IPython\core\pylabtools.
↪py:151: UserWarning: This figure was using constrained_layout, but that is
↪incompatible with subplots_adjust and/or tight_layout; disabling constrained_
↪layout.
  fig.canvas.print_figure(bytes_io, **kw)
```



8.1.4 Exogenize the variable only for the period during which it is shocked

This scenario introduces a subtle but import difference. Here we the variable is again exogenized using the fix syntax. But this time it is exogonized only for the period where the variable is shocked.

This means that the consumption function will be de-activated for only three years (instead of the whole period as in the previous examples). As a result, the values that consumption takes in 2028, 2029, … 2040 depend on the model, not the level it was set to when exogenized (which was the case in the 3 previous versions).

Looking at the maths of the model the consumption equation is effectively split into two.

for the period before 2025 $cons_d = 0$ and the consumption equation simplifies to:

$cons = C(X)$

for the period 2025-2028 it is exogenized ($cons_d = 1$) so it simplifies to:

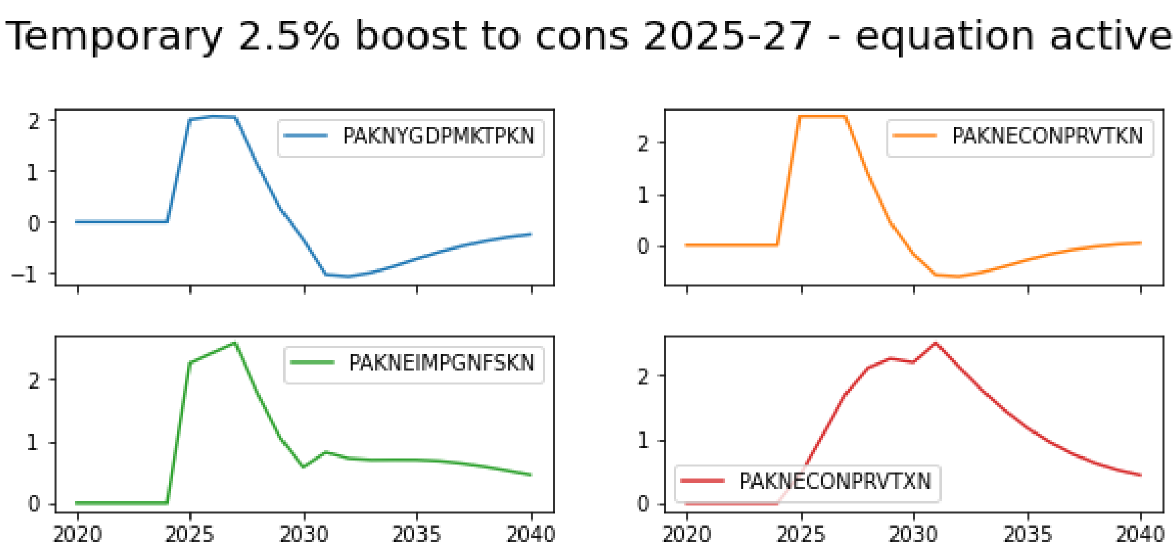
$cons = cons_x$

but in the final period 2028-2040 ($cons_d = 0$) and the equation reverts to:

$cons = C(X)$

```
mpak.smpl() # reset the active sample period to the full model.
CExogTemp=mpak.fix(bline,'PAKNECONPRVTKN',2025,2027)#Consumption is exogenized only for three years 2025 2026 and 2027 PAKNECONPRVTKN_D=1 for 2025,2026, 2027 0 elsewhere.
CExogTemp = CExogTemp.upd('<2025 2027> PAKNECONPRVTKN_X * 1.025',lprint=0)       #In subsequent years it's level will be determined by the equation

#Solve the model
CExogTempRes = mpak(CExogTemp,2020,2040,keep='2.5% increase in C 2025-27 -- temporarily exogenized') # simulates the model
mpak['PAKNYGDPMKTPKN PAKNECONPRVTKN PAKNEIMPGNFSKN PAKNECONPRVTXN'].difpctlevel.mul100.plot(title="Temporary 2.5% boost to cons 2025-27 - equation active")
```

```
C:\Users\wb268970\.conda\envs\modelflow\lib\site-packages\IPython\core\pylabtools.py:151: UserWarning: This figure was using constrained_layout, but that is incompatible with subplots_adjust and/or tight_layout; disabling constrained_layout.
  fig.canvas.print_figure(bytes_io, **kw)
```



These results have subtle differences compared with the previous. The most obvious is visible in looking at the graph for Consumption. Rather than reverting immediately to its earlier pre-shock level, it falls more gradually and actually overshoots (falls below its earlier level), before returning slowly to its pre-shock level. That is because unlike in the previous shocks, its path is being determined endogenously and reacting to changes elsewhere in the model, notably changes to prices, wages and government spending as well as the lagged level of consumption.

```
print('Consumption base and shock levels\r\n');

print('Real values in 2030');
print(f'Base value:  {bline.loc[2028,"PAKNECONPRVTKN"]:,.0f}.\tShocked value:
↪{CExogTempRes.loc[2028,"PAKNECONPRVTKN"]:,.0f}.\r\n'
    f'Percent difference: {round(100*((CExogTempRes.loc[2030,"PAKNECONPRVTKN"]-bline.
↪loc[2028,"PAKNECONPRVTKN"])/bline.loc[2028,"PAKNECONPRVTKN"]),2)}')
print('\r\nReal values in 2040');
print(f'Base value:  {bline.loc[2040,"PAKNECONPRVTKN"]:,.0f}.\tShocked value:
↪{CExogTempRes.loc[2040,"PAKNECONPRVTKN"]:,.0f}.\r\n'
    f'Percent difference: {round(100*((CExogTempRes.loc[2040,"PAKNECONPRVTKN"]-bline.
↪loc[2040,"PAKNECONPRVTKN"])/bline.loc[2040,"PAKNECONPRVTKN"]),2)}')
```

```
Consumption base and shock levels

Real values in 2030
Base value:  27,241,278.        Shocked value: 27,241,278.
Percent difference: 5.54

Real values in 2040
Base value:  38,676,995.        Shocked value: 38,692,817.
Percent difference: 0.04
```

8.1.5 Simulation with Add factors

Add factors are a crucial element of the macromodels of the World Bank and serve multiple purposes.

In simulation, add-factors allow simulations to be conducted **without** de-activating behavioural equations. Such shocks are often referred to as **endogenous** shocks because the equation of the behavioural variable that is shocked remains active throughout.

In some ways they are very similar to a temporary exogenous shock. Both ways of producing the shock allow the shocked variable to respond endogenously in the period after the shock. The main difference between the two approaches is that:

- **Endogenous** shocks (Add-Factor shocks) allow the shocked variable to respond to changed circumstances that occur during the period of the shock.
  - This approach makes most sense for "animal spirits", shocks where the underlying behaviour is expected to change.
  - It also makes sense when actions of one part of an aggregate is likely to impact behaviour of other sectors within an aggregate
  - increased investment by a particular sector would be an example here as the associated increase in activity is likely to increase investment incentives in other sectors, while increased demand for savings will increase interest rates and the cost of capital operating in the opposite direction.
  - Sustained changes in behaviour, for example increased propensity to invest because of improved recognition
- **Exogenous** shocks to endogenous variables fix the level of the shocked variable during the shock period.
  - Changes in government spending policy, something that is often largely an economically exogenous decision.

Simulating the impact of a planned investment

The below simulation uses the add-factor to simulate the impact of a 3 year investment program beginning in 2025 of 1 percent of GDP per year, that is financed through an increase in foreign direct investment. This might reflect a specific large scale plant that is being constructed due to a deal reached by the government with a foreign manufacturer. The add-factor approach is chosen because the additional investment is likely to increase demand for the products of other firms, which is likely to incite them to add to their investments as well.

How to translate the economic shock into a model shock

Add-factors in the MFMod framework are applied to the intercept of an equation (not the level of the dependent variable). This preserves the estimated elasticities of the equation, but makes introduction of an add-factor shock somewhat more complicated than the exogenous approach. Below a step-by-step how-to guide:

1. Identify numerical size of the shock
2. Examine the functional form of the equation, to determine the nature of the add factor. If the equation is expressed as a:
   - **growth rate** then the add-factor will be an addition or subtraction to the growth rate
   - **percent of GDP (or some other level)** then the add-factor will be an addition or subtraction to the share of growth.
   - **Level** then the add-factor will be a direct addition to the level of the dependent variable
3. Convert the economic shock into the units of the add-factor
4. Shock the add-factor by the above amount and run the model
   - Note the add-factor is an exogenous variable in the model, so shocking it follows the well established process for shocking an exogenous variable.

Determine the size of shock

Above we identified the shock as to be a 1 percent of GDP increase in FDI that flows directly into private-sector investment. A first step would be to determine the variables that need to be shocked (FDI) and private investment. To do this we can query the variable dictionary.

```
mpak['*NY*'].des
```

```
PAKNYGDPDISCCN             : GDP Disc., LCU mn
PAKNYGDPDISCKN             : GDP Disc., 2000 LCU mn
PAKNYGDPFCSTCN             : GDP Factor Cost Local Currency units Volumes National␣
 ↪base year
PAKNYGDPFCSTKN             : GDP Factor Cost Local Currency units Volumes National␣
 ↪base year
PAKNYGDPFCSTXN             : GDP Factor Cost Local Currency units Implicit Price␣
 ↪deflator
PAKNYGDPFCSTXN_A           : Add factor:GDP Factor Cost Local Currency units␣
 ↪Implicit Price deflator
PAKNYGDPFCSTXN_D           : Fix dummy:GDP Factor Cost Local Currency units␣
 ↪Implicit Price deflator
PAKNYGDPFCSTXN_FITTED      : Fitted  value:GDP Factor Cost Local Currency units␣
 ↪Implicit Price deflator
PAKNYGDPFCSTXN_X           : Fix value:GDP Factor Cost Local Currency units␣
 ↪Implicit Price deflator
```

(continues on next page)

(continued from previous page)

```
PAKNYGDPGAP_                 : Output Gap (% of Potential GDP)
PAKNYGDPMKTPCD               : GDP, Market Prices, US$ mn
PAKNYGDPMKTPCN               : GDP, Market Prices, LCU mn
PAKNYGDPMKTPCN_VALUE_2010 : PAKNYGDPMKTPCN_VALUE_2010
PAKNYGDPMKTPKD               : GDP, Market Prices, 2000 US$ mn
PAKNYGDPMKTPKN               : Real GDP
PAKNYGDPMKTPKN_VALUE_2010 : PAKNYGDPMKTPKN_VALUE_2010
PAKNYGDPMKTPXN               : GDP, Marker Prices, LCU Price defl., 2000 = 1
PAKNYGDPPOTLKN               : Potential Output, constant LCU
PAKNYGDPTFP                  : Total factor productivity
PAKNYTAXNINDCN               : Net Indirect Taxes Local Currency units Values
PAKNYTAXNINDKN               : Net Indirect Taxes Local Currency units Volumes␣
↪National base year
PAKNYWBFORMSH                : PAKNYWBFORMSH
PAKNYWBINFMSH                : PAKNYWBINFMSH
PAKNYWRTFORMCN               : PAKNYWRTFORMCN
PAKNYWRTFORMCN_A             : Add factor:PAKNYWRTFORMCN
PAKNYWRTFORMCN_D             : Fix dummy:PAKNYWRTFORMCN
PAKNYWRTFORMCN_FITTED        : Fitted  value:PAKNYWRTFORMCN
PAKNYWRTFORMCN_X             : Fix value:PAKNYWRTFORMCN
PAKNYWRTINFMCN               : PAKNYWRTINFMCN
PAKNYWRTINFMCN_A             : Add factor:PAKNYWRTINFMCN
PAKNYWRTINFMCN_D             : Fix dummy:PAKNYWRTINFMCN
PAKNYWRTINFMCN_FITTED        : Fitted  value:PAKNYWRTINFMCN
PAKNYWRTINFMCN_X             : Fix value:PAKNYWRTINFMCN
PAKNYWRTTOTLCN               : PAKNYWRTTOTLCN
PAKNYYGOSOTLCN               : PAKNYYGOSOTLCN
PAKNYYWBFORMCN               : PAKNYYWBFORMCN
PAKNYYWBINFMCN               : PAKNYYWBINFMCN
PAKNYYWBINFMCN_              : PAKNYYWBINFMCN_
PAKNYYWBTOTLCN               : Total Wage Bill
PAKNYYWBTOTLCN_              : Labor Share of Income
```

Identify the functional form(s)

To understand how to shock using the add factor, it is essential to understand how the add-factor enters into the equation.

Addfactor is on the intercept of	Shock needs to be calculated as
a growth equation	a change in th growth rate
Share of GDP	a percent of GDP
Level	as change in the level

Use the .eviews command or .original command to identify the functional forms if the equation to be shocked.

```
# This needs to be rewritten to use the eviews expression when published
mpak['PAKNEGDIFPRVKN'].frml
```

```
PAKNEGDIFPRVKN : FRML <DAMP,STOC> PAKNEGDIFPRVKN = (PAKNEGDIFPRVKN_
↪A*PAKNEGDIKSTKKN(-1)+ (0.00212272413966296+0.970234989019907*(PAKNEGDIFPRVKN(-1)/
↪PAKNEGDIKSTKKN(-2))+(1-0.970234989019907)*(((LOG(PAKNYGDPPOTLKN))-
↪(LOG(PAKNYGDPPOTLKN(-1))))+PAKDEPR)+0.0525240494260597*((LOG(PAKNEKRTTOTLCN/
↪PAKNYGDPFCSTXN))-(LOG(PAKNEKRTTOTLCN(-1)/PAKNYGDPFCSTXN(-1))))) *PAKNEGDIKSTKKN(-
↪1)) * (1-PAKNEGDIFPRVKN_D)+ PAKNEGDIFPRVKN_X*PAKNEGDIFPRVKN_D $
```

(continues on next page)

(continued from previous page)

Calculate the size of the required add factor shock

The shock to be executed is 0.5 percent of GDP.

It is assumed that the financing will come from FDI and that all the money will be spent in one year on private investment.

The private investment equation is written as a share of the capital stock. Therefore, the add-factor needs to be shocked by adding 1 percent of GDP to private investment in 2028 divided by the capital stock in 2028.

```
#Create a DataFrame AFShock that is equal tothe baseline
AFShock=bline

#Display the level of the AF
print("Pre shock levels")
AFShock.loc[2025:2030,['PAKNEGDIFPRVKN_A','PAKNEGDIFPRVKN','PAKNEGDIKSTKKN']]

#print(AFShock.loc[2025:2030,'PAKNEGDIFPRVKN']/AFShock.loc[2025:2030,'PAKNYGDPMKTPKN
↪']*100)
```

```
Pre shock levels
```

```
      PAKNEGDIFPRVKN_A  PAKNEGDIFPRVKN  PAKNEGDIKSTKKN
2025         -0.000458    1.602854e+06    4.730392e+07
2026         -0.000389    1.581104e+06    4.814879e+07
2027         -0.000331    1.569541e+06    4.900980e+07
2028         -0.000281    1.569141e+06    4.989869e+07
2029         -0.000239    1.580577e+06    5.082694e+07
2030         -0.000203    1.604394e+06    5.180590e+07
```

Below the mfcalc routine is used to set the addfactor variable equal to its previous value plus the equivalent of 1 percent of GDP when expressed as a percent of the previous period's level of private investment.

```
AFShock=AFShock.mfcalc("<2028 2028> PAKNEGDIFPRVKN_A = PAKNEGDIFPRVKN_A + (.
↪01*PAKNYGDPMKTPKN/PAKNEGDIKSTKKN)");

print("Post shock levels")
AFShock.loc[2025:2030,'PAKNEGDIFPRVKN_A']
```

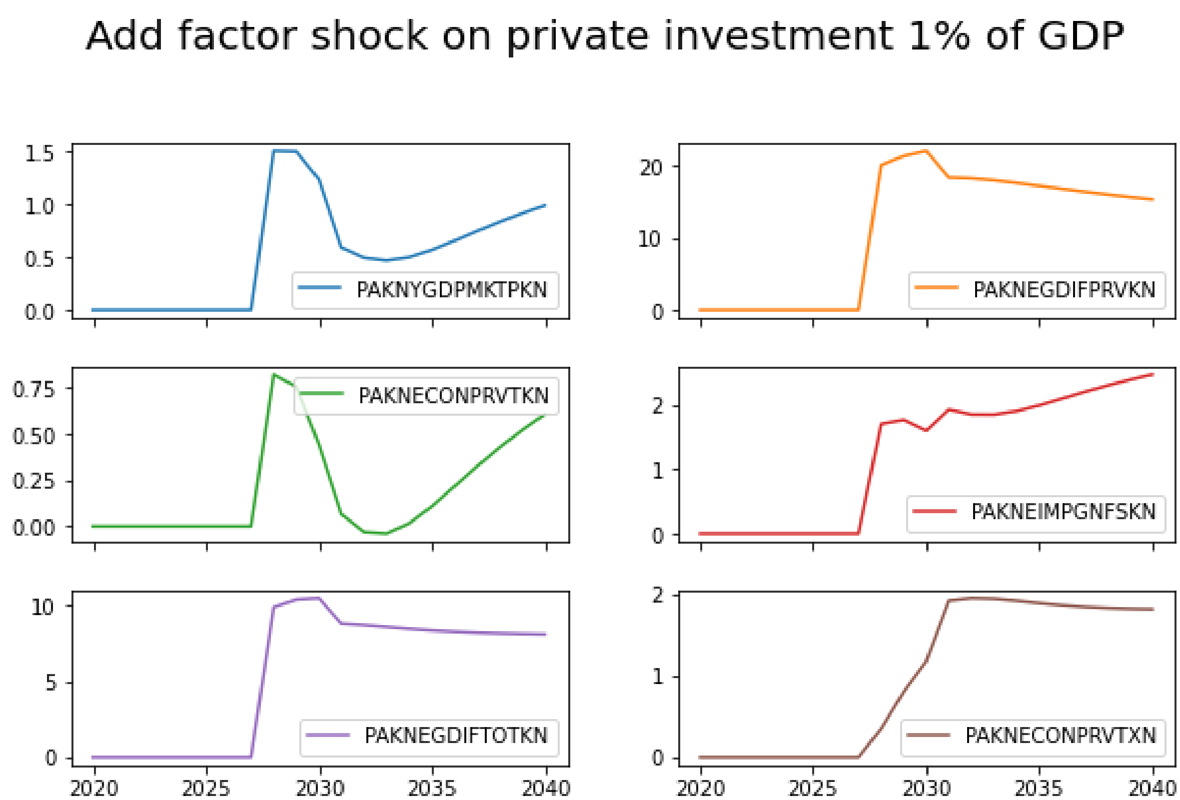
```
Post shock levels
```

```
2025   -0.000458
2026   -0.000389
2027   -0.000331
2028    0.005774
2029   -0.000239
2030   -0.000203
Name: PAKNEGDIFPRVKN_A, dtype: float64
```

Run the shock

```
AFShockRes = mpak(AFShock,2020,2040,keep='1% of GDP increase in FDI and private␣
↪investment (AF shock)')
mpak['PAKNYGDPMKTPKN PAKNEGDIFPRVKN PAKNECONPRVTKN PAKNEIMPGNFSKN PAKNEGDIFTOTKN␣
↪PAKNECONPRVTXN'].difpctlevel.mul100.plot(title="Add factor shock on private␣
↪investment 1% of GDP")
```

```
C:\Users\wb268970\.conda\envs\modelflow\lib\site-packages\IPython\core\pylabtools.
↪py:151: UserWarning: This figure was using constrained_layout, but that is␣
↪incompatible with subplots_adjust and/or tight_layout; disabling constrained_
↪layout.
  fig.canvas.print_figure(bytes_io, **kw)
```



CHAPTER NINE

REPORT WRITING AND SCENARIO RESULTS

ModelFlow, standard pandas routines and other python libraries like Matplotlib and Plotly can be used to visualize and compare dataframes and therefore the results, from scenarios – as indeed has been done in the preceding paragraphs.

In addition, ModelFlow also provides several specific routines that make such comparisons easier.

9.1 The Keep option

The **Keep** option facilitates the comparison of results from different scenarios run on a give model object. In each of the simulations executed above, the keep option was activated. This causes the results from each simulation in a unique DataFrame that can be identified by the descriptor given to it.

9.2 The .keep_plot() method

The keep_plot method can be used to plot and compare results from the various scenarios that had been run earlier using the keep= option.

By default the results across all scenarios for each selected variables will be shown on one chart at a time.

```
mpak.keep_plot('PAKNYGDPMKTPCN PAKNECONPRVTKN PAKNEIMPGNFSKN',  legend=True)
#show for each variable on a separate chart the results from each kept scenario
```

```
{'PAKNYGDPMKTPCN': <Figure size 720x432 with 1 Axes>,
 'PAKNECONPRVTKN': <Figure size 720x432 with 1 Axes>,
 'PAKNEIMPGNFSKN': <Figure size 720x432 with 1 Axes>}
```

9.2.1 keep_plot() options

The **variables** to be displayed are listed as first argument. Variable names can include wildcards (using * for any string and ? for any character).

Transformation of data displayed:

showtype=	Use this operator
'level' (default)	No transformation
'growth'	The growth rate in percent
'change'	The yearly change (Δ)

legend placement

legend=	Use this operator
False (default)	The legends are placed at the end of the corresponding line
True	The legends are places in a legend box

Often it is useful to compare the scenario results with the baseline result. This is done with the diff argument.

diff=	Use this operator
False (default)	All entries in the keep_solution dictionary are displayed
True	The difference to the first entry is shown.

It can also be useful to compare the scenario results with the baseline result **measured in percent**. This is done with the diffpct argument.

diffpct=	Use this operator
False (default)	All entries in the keep_solution dictionary is displayed
True	The difference in percent to the first entry is shown

Note: `'keep_plot()` and `.keep_plot_multi()` return a python object that points to the in memory version of the rendered figure(s). This object can be used to modify the graph (see examples towards the end of this chapter.

`savefig='[path/]<prefix>.<extension>'` Will create a number of files with the charts. The files will be saved location with name `<path>/<prefix><variable name>.<extension>` The extension determines the format of the saved file: `pdf`, `svg` and `png` are the most common extensions.

9.2.2 An example using the diff=TRUE option

When `diff=True` (or 1) results will be shown all of the selected scenarios presented as the change in selected variables with respect to the first scenario – in this instance the scenario saves with the name `baseline`.

```
Note in this instance `baseline` and `basedf` are the same because they were defined
↪that way.  However, there is nothing in the system that guarantees that the first
↪`keep` scenario will be the baseline or the `basedf` scenario.
```

```python
mpak.keep_plot('PAKNYGDPMKTPCN PAKNECONPRVTKN PAKNEIMPGNFSKN', diff=1, legend=True)
```

```
{'PAKNYGDPMKTPCN': <Figure size 720x432 with 1 Axes>,
 'PAKNECONPRVTKN': <Figure size 720x432 with 1 Axes>,
 'PAKNEIMPGNFSKN': <Figure size 720x432 with 1 Axes>}
```

9.2.3 The `showtype` option

In this example the difference with respect first `'keep` scenario `baseline` values are once again shown. This time the `showtype` option has been set to `growth`. As a result the data is displayed as the difference in the growth rate.

```
mpak.keep_plot('PAKNYGDPMKTPCN PAKNEIMPGNFSKN', diff=1,showtype='growth', legend=True)
```

```
{'PAKNYGDPMKTPCN': <Figure size 720x432 with 1 Axes>,
 'PAKNEIMPGNFSKN': <Figure size 720x432 with 1 Axes>}
```

9.2.4 The diffpct option

Setting `diffpct=True` instructs `.keep_plot()` to display the data as a percent deviation from the first `kep` scenario.

```
mpak.keep_plot('PAKNYGDPMKTPCN  PAKNEIMPGNFSKN', diffpct=1,legend="Change in level as␣
↪a % of first keep scenario")
```

```
{'PAKNYGDPMKTPCN': <Figure size 720x432 with 1 Axes>,
 'PAKNEIMPGNFSKN': <Figure size 720x432 with 1 Axes>}
```

Differences in percent of baseline values

In this plot, the same results are presented, but as percent deviations from the baseline values of the displayed data.

```
mpak.keep_plot('PAKNYGDPMKTPCN PAKNECONPRVTKN ', diffpct=1,showtype='level',␣
↪legend=True)
```

```
{'PAKNYGDPMKTPCN': <Figure size 720x432 with 1 Axes>,
 'PAKNECONPRVTKN': <Figure size 720x432 with 1 Axes>}
```

9.2.5 The `.keep_switch()` method

The `.keep_switch()` method restricts the number of scenarios on which subsequent calls to `.keep_plot()` (and `.keep_plot_multi()`) are executed on. `.keep_switch()` can be passed a list of scenarios or using a wildcard selector.

The `.keep_solutions.keys()` method

The `.keep_solutions.keys()` method generates a list of the solutions that have been kept previously.

```
mpak.keep_solutions.keys()
```

```
dict_keys(['Baseline', '$25 increase in oil prices 2025-27', '2.5% increase in C␣
↪2025-40', '2.5% increase in C 2025-27 -- exog whole period', '2.5% increase in C␣
↪2025-27 -- exog whole period --KG=True', '2.5% increase in C 2025-27 --␣
↪temporarily exogenized', '1% of GDP increase in FDI and private investment (AF␣
↪shock)'])
```

(continues on next page)

(continued from previous page)

To specify exactly which scenarios to show in a keep_plot, the `scenarios=` option of `.keepswitch()` must be initialized with a "|" delimited string of the names of the scenarios (retrieved above) that are to be displayed.

By placing the `.keepswitch()` in a `with` clause the scenario restriction will only apply to indented lines that follow the with construct.

```
with mpak.keepswitch(scenarios='2.5% increase in C 2025-40|2.5% increase in C 2025-27␣
↪-- exog whole period|2.5% increase in C 2025-27 -- exog whole period --KG=True|2.5%␣
↪increase in C 2025-27 -- temporarily exogenized'):
    mpak.keep_plot('PAKNYGDPMKTPKN PAKGGBALOVRLCN PAKGGDEBTTOTLCN',diff=False,
↪showtype='growth',legend=True);
```

Keepswitch with wildcard selection

Below we generate a series of plots using

```
with mpak.keepswitch(scenarios='*2025*'):
    mpak.keep_plot('PAKNYGDPMKTPKN PAKNECONPRVTKN',showtype='growth',legend=True);
```

9.3 The `.keep_plot_multi()` method

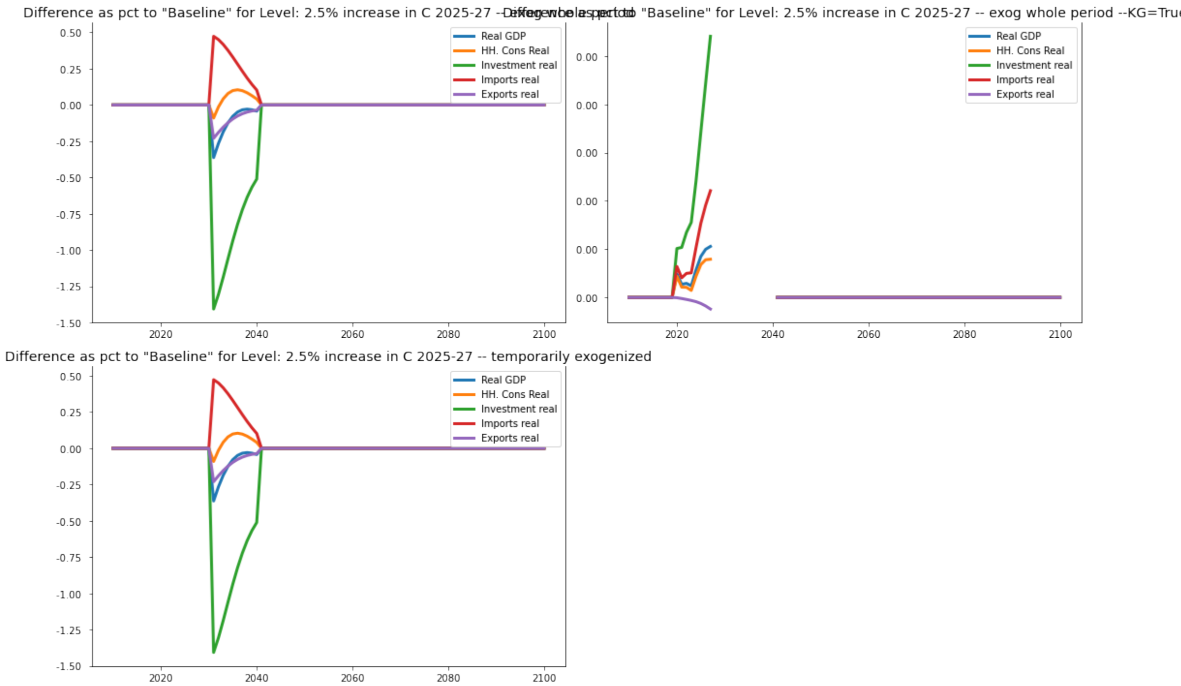
The `.keep_plot_multi()` method allows several charts to be displayed in a grid. The size of each chart can be set with the `size=(w,h)` option, where the units of with and height are in centimetres.

```
with mpak.set_smpl(2000,2040):
    with mpak.keepswitch(scenarios="baseline *exog*"):
        var_figs = mpak.keep_plot_multi('PAKNYGDPMKTPKN PAKNECONPRVTKN PAKNEGDIFTOTKN␣
↪PAKNEIMPGNFSKN PAKNEEXPGNFSKN',2010,2100,keep_dim=0,legend=1
                                       ,size=(20,20) ,diffpct=True,title=''  );
```

As indicated earlier both `keep_plot()` and `keep_plot_multi()` return a variable that can be used to embellish or modify the figures produced by the automatic routines.

For example the charts can be resized.

```
var_figs.set_size_inches(15,10)
var_figs
```



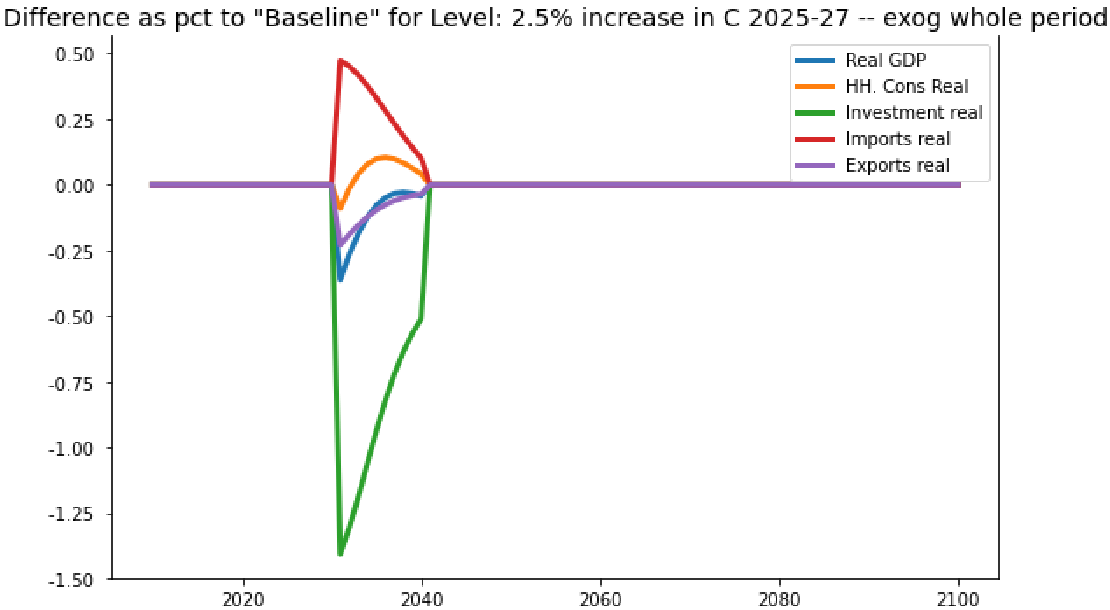
Individual charts can be deleted from the grid.

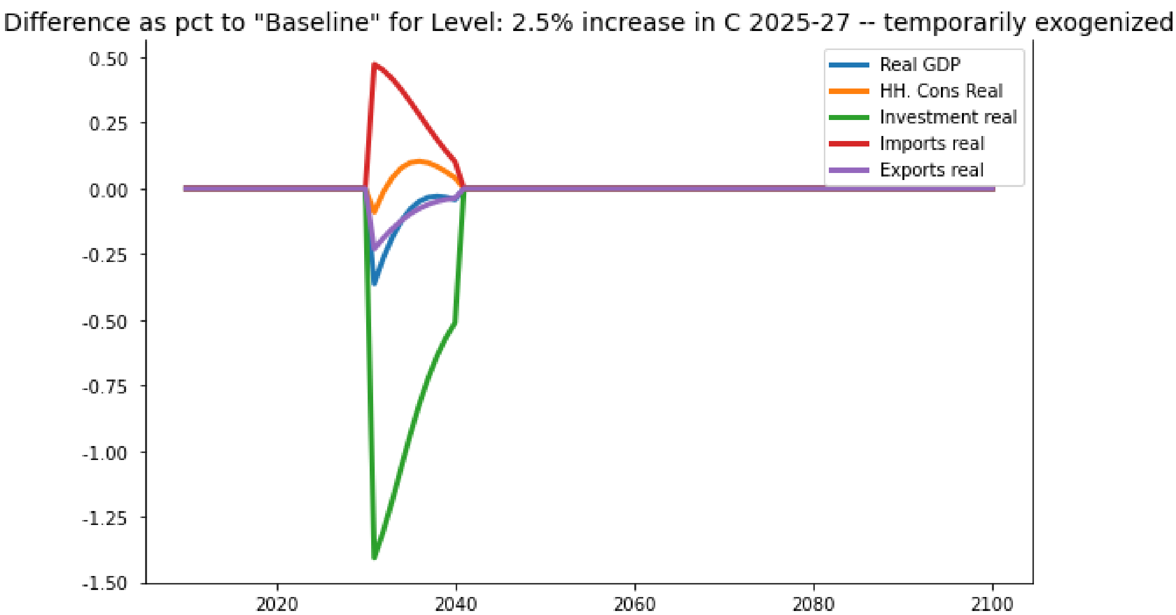
Note: The grid representation of the individual charts is returned as a 0-based vector of charts. Thus the first figure is the zeroeth and the second is the first.

9.3.1 Delete a chart from th grid

A chart can be deleted from the grid by referencing it and calling the `.remove()` method.

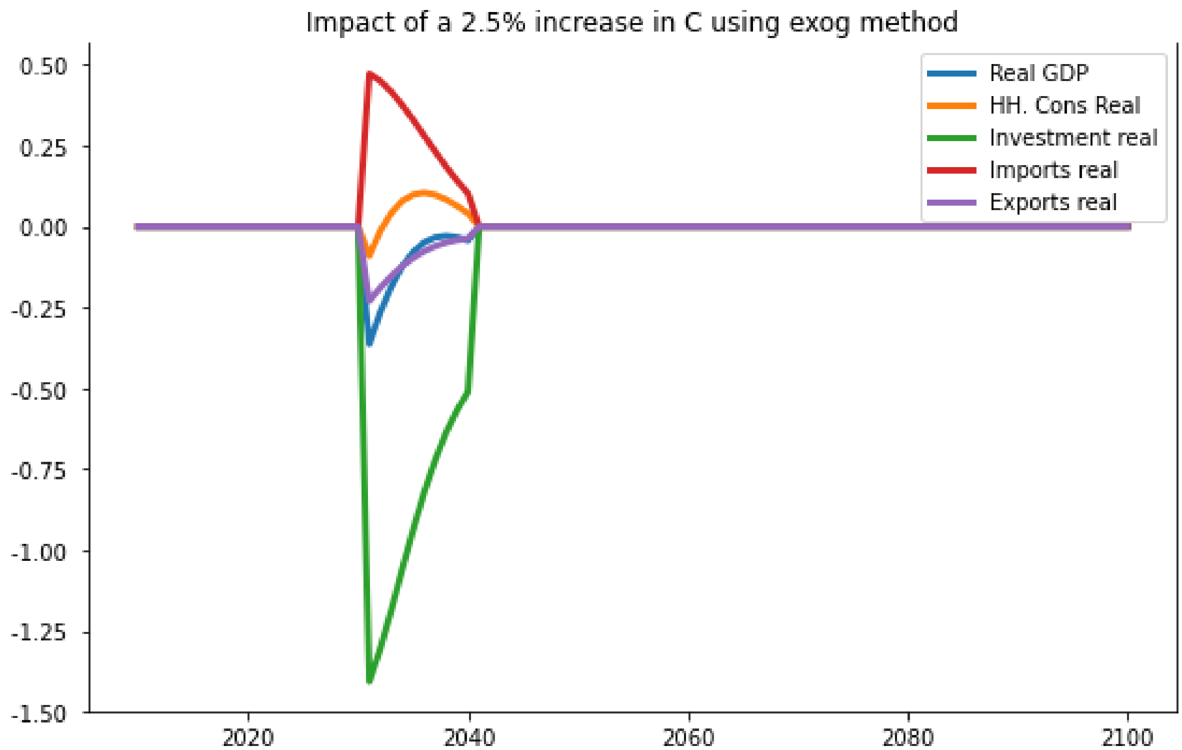
```
var_figs.axes[1].remove()
var_figs
```

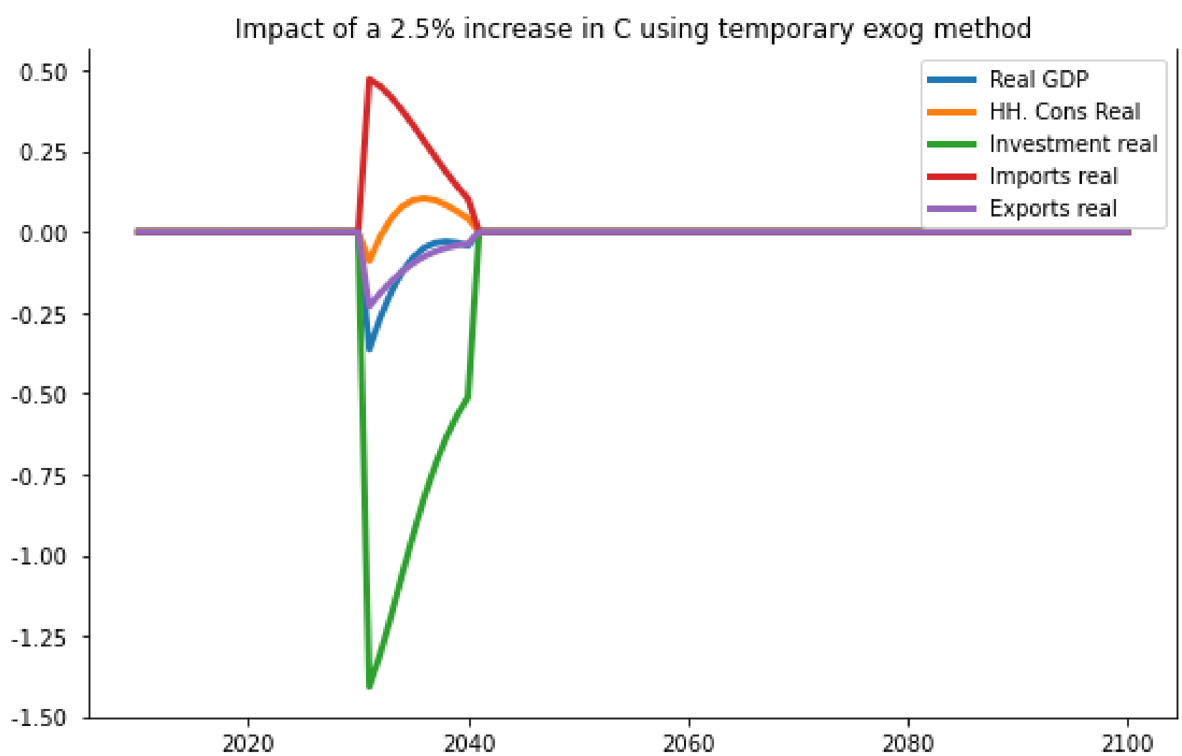




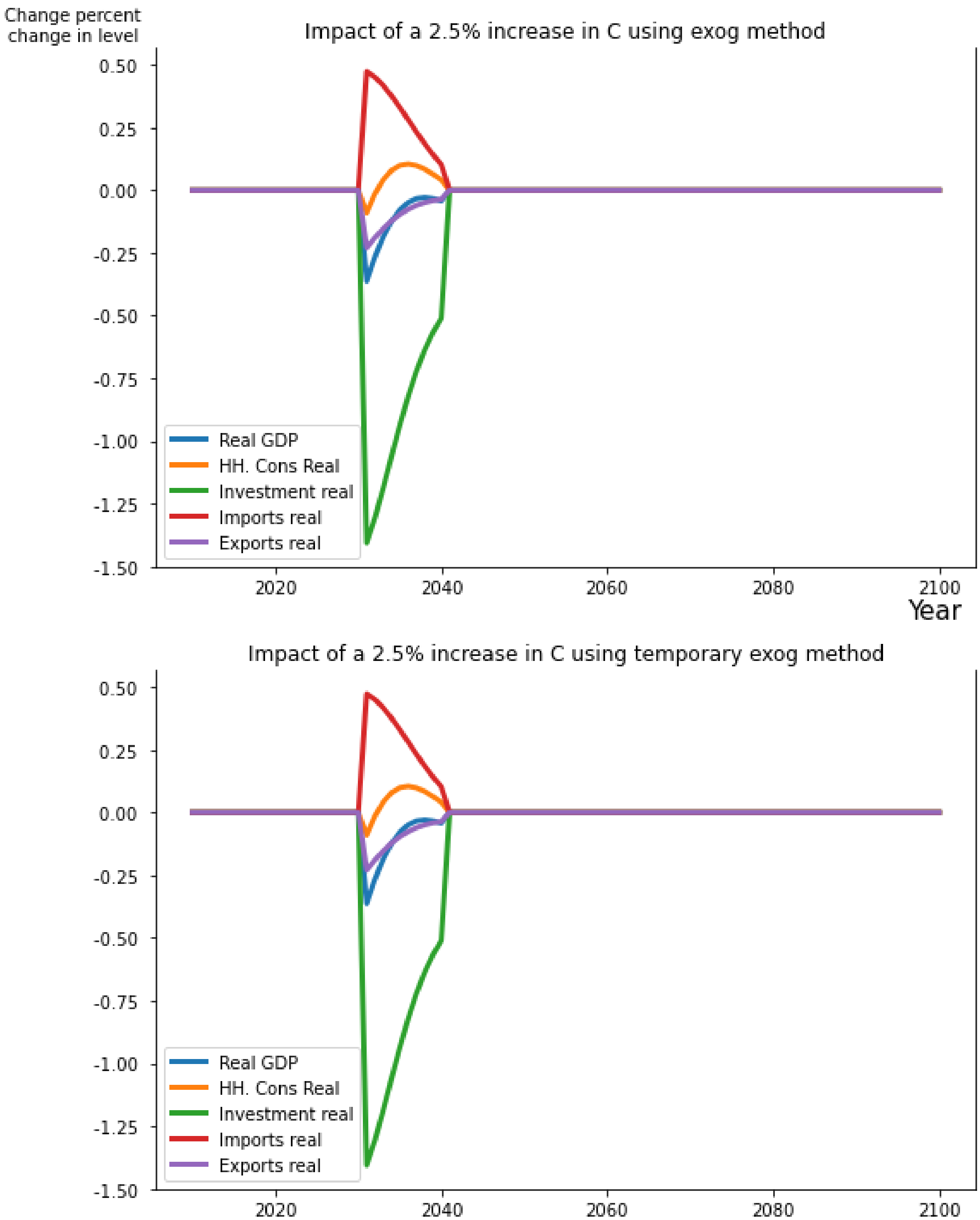
The same mechanism can be used to revise the titles of the indidividual charts and annotate them.

```
var_figs.axes[0].set_title('Impact of a 2.5% increase in C using exog method');     #
↪many properties can be set afterward
var_figs.axes[1].set_title('Impact of a 2.5% increase in C using temporary exog method
↪');
var_figs
```





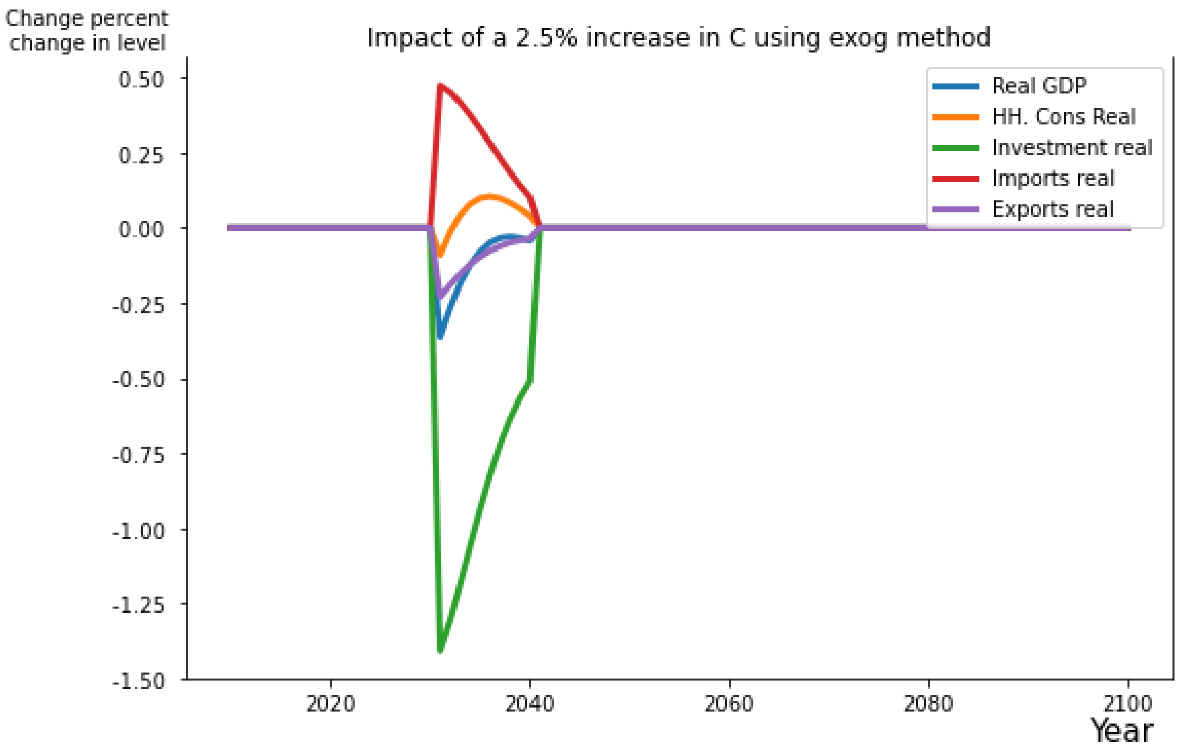

```
var_figs.axes[0].set_xlabel('Year')
var_figs.axes[0].set_ylabel('Change percent\nchange in level',fontsize=10)
var_figs.axes[0].yaxis.set_label_coords(-0.1,1.02)
var_figs.axes[0].xaxis.set_label_coords(.95,-.06)
var_figs
```

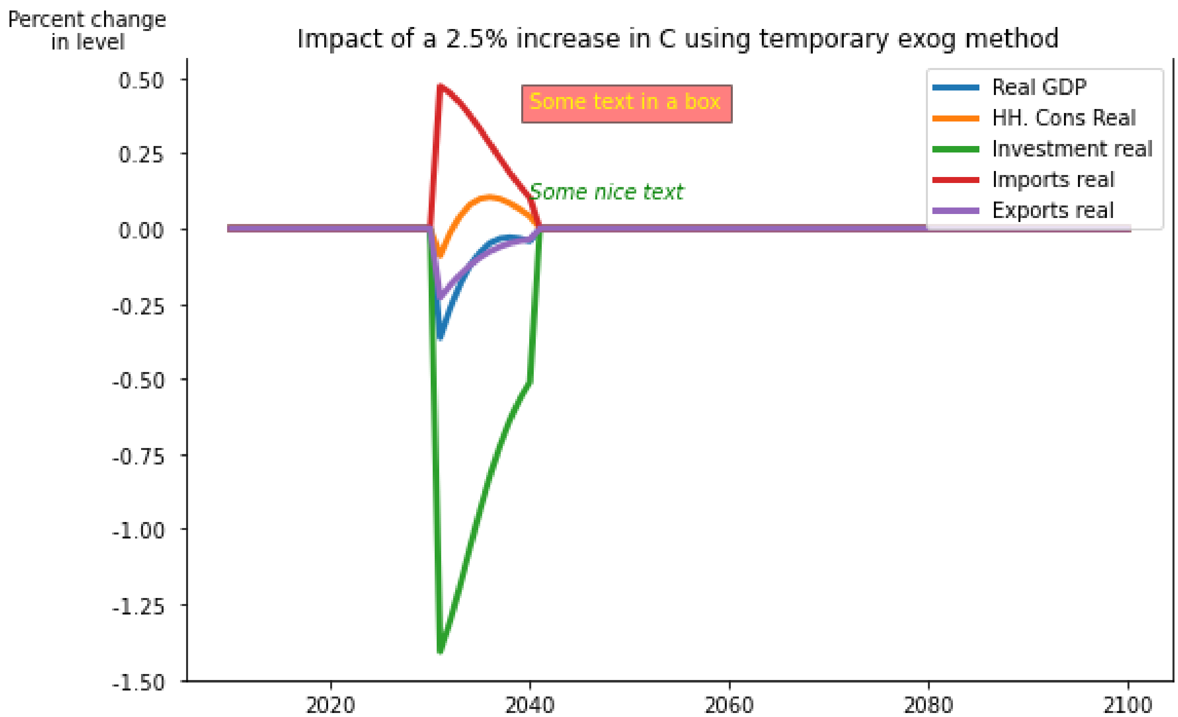


Variables pointing to the individual charts can be defined and used to make modifications to individual charts within the overall figure.

```
fig1=var_figs.axes[0]
fig2=var_figs.axes[1]
```

```
fig2.set_ylabel('Percent change\nin level',fontsize=10)
fig2.yaxis.set_label_coords(-0.1,1.02) #place axes labels
fig2.xaxis.set_label_coords(.95,-.06)



fig2.text(2040.,0.4, 'Some text in a box',
          color='yellow',bbox=dict(facecolor='red', alpha=0.5));
fig2.text(2040.,0.1, 'Some nice text',
          style='italic',color='green');

var_figs
```


Impact of a 2.5% increase in C using exog method
Change percent
change in level
0.50
0.25
0.00
-0.25
-0.50
-0.75
-1.00
-1.25
-1.50
2020
2040
2060
2080
2100
Year
Real GDP
HH. Cons Real
Investment real
Imports real
Exports real


Impact of a 2.5% increase in C using temporary exog method
Percent change
in level
Some text in a box
Some nice text
0.50
0.25
0.00
-0.25
-0.50
-0.75
-1.00
-1.25
-1.50
2020
2040
2060
2080
2100
Real GDP
HH. Cons Real
Investment real
Imports real
Exports real

9.3.2 The results visualization widget view

When working in Jupyter Notebook, referencing a selection of series will cause a data visualization widget to be generated that allows you to look at results (`basesdf` vs `latestdf`) for the selected variables as tables or charts, as levels, as growth rates and as percent differences from baseline.

```
mpak['PAKNYGDPMKTPCN PAKNYGDPMKTPKN PAKGGEXPTOTLCN PAKGGREVTOTLCN PAKNECONGOVTKN']
```

```
Tab(children=(Tab(children=(HTML(value='<?xml version="1.0" encoding="utf-8"␣
↪standalone="no"?>\n<!DOCTYPE svg …
```

```
%matplotlib inline
```

CHAPTER

TEN

MORE COMPLEX SCENARIOS

The preceding chapter introduced four different ways of preparing a solution and the forms the backbone of running simulations on World Bank models in `modelflow`. This chapter builds on those examples and delves into some of the challenges involved in translating a real-world policy challenge into the model-world and then back again.

The particular problem to be examined is in the introduction of a Carbon Tax. The model used and the example presented are both taken from the model of Pakistan presented here: .

10.1 Setting up the environment

As always the modelflow and other python libraries that are to be used must be imported into the current session.

```
from modelclass import model
model.widescreen()
model.scroll_off()
%load_ext autoreload
%autoreload 2
```

```
<IPython.core.display.HTML object>
```

10.2 Load a pre-existing model, data and descriptions

Load the Pakistan model, which is comprised of the model object, its estimated equations and the data. The `pcim` file was created by the World Bank from the original EViews model used in the paper .

```
mpak,baseline = model.modelload('../models/pak.pcim',alfa=0.7,run=1,keep="Baseline")
```

```
file read:  C:\mflow\modelflow-manual\papers\mfbook\content\models\pak.pcim
```

10.3 The policy problem

The objective of this chapter is to produce a simulation of the economic and climate effects of the introduction of a carbon tax in Pakistan.

The variable `mpak` loaded above contains the model instance, the variables, equations and the data for the model. On load the model was solved, and the results of that initiaql solve was assigned to the `DataFrame` `baseline`.

The Pakistan model contains three carbon tax variables:

Mnemonic	Meaning
PAKGGREVCO2CER	The effective carbon tax rate on Coal
PAKGGREVCO2GER	The effective carbon tax rate on Gas
PAKGGREVCO2OER	The effective carbon tax rate on Crude Oil

As discussed in earlier chapters the meaning of the mnemonics can be retrieved from the model:

```
mpak['PAKGGREVCO2*ER'].des
```

```
PAKGGREVCO2CER : Carbon tax on coal (USD/t)
PAKGGREVCO2GER : Carbon tax on gas (USD/t)
PAKGGREVCO2OER : Carbon tax on oil (USD/t)
```

Alternatively one can search on the variable descriptions to retrieve the mnemonics of variables. Below the exclamation sign at the beginning of the string notifies the matching algorithm to search the variables descriptions (not he mnemonics) and return all variables that match.

```
mpak['!*tco2*'].des
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [6], in <cell line: 1>()
----> 1 mpak['!*tco2*'].des

File ~\.conda\envs\modelflow\lib\site-packages\ModelFlow-1.0.8-py3.9.egg\modelvis.
↪py:153, in vis.des(self)
    151 def getdes(var,l):
    152     return f'{var:<{l}} : {self.model.var_description[var]}'
--> 153 mlength = max([len(v) for v in self.names])
    155 out = '\n'.join(getdes(var,mlength) for var in self.names)
    156 print(out)

ValueError: max() arg is an empty sequence
```

10.4 Add variable descriptions

A modelflow model imported from EViews will inherit the variable descriptors coming from Eviews. Not all EViews variables will necessarily have a description so such descriptions can be added to the existing using the `.set_var_descri[tion()` method as below.

As coded, the call

`mpak.set_var_description({**mpak.var_description,**extra_description})`

adds the extra_description dictionary to the pre-existing mpak,var_description dictionary.

Several `modelflow` methods include a `rename` option, which if set to True will substitute the description for the variable name in any outputs. Variables can also be selected for by using the `mpak['!*subtext*']` syntax, where subtext is some text that appears in the variable descriptor.

```
extra_description = {'PAKNYGDPMKTPKN': 'GDP',
'EMISCOAL'         : 'Coal emissions',
'EMISGAS'          : 'Gas Emissions',
'EMISOIL'          : 'Gas Emissions',
'PAKCCEMISCO2CKN' : 'Coal emissions, tCO2e',
'PAKCCEMISCO2GKN' : 'Natural Gas emissions, tCO2e',
'PAKCCEMISCO2OKN' : 'Crude Oil emissions, tCO2e',
'PAKCCEMISCO2TKN' : 'Total emissions, tCO2e',
'PAKGGREVEMISCN'  : 'Revenue from emissions taxes',
 'PAKLMUNRTOTLCN': 'Unemployment rate',
 'PAKGGDBTTOTLCN_': 'Debt (%GDP)',
 'PAKGGREVTOTLCN': 'Fiscal revenues',
 'PAKWDL': 'Working days lost due to pollution'}
mpak.set_var_description({**mpak.var_description,**extra_description})
```

10.5 Simulating the impact of a imposing a carbon price

To run a simulation, the following steps must invariably be followed.

1. Create a new DataFrame, typically a copy of an existing one.
2. Change the value in the new df of the variable(s) to be shocked.
3. Solve the model using the newly altered df as the input df.

```
# Create copy of the baseline df
alternative_df = baseline.copy()
#set the effective carbon tax of all three carbon tax variables equal to 30 USD
alternative_df.loc[2025:2100,['PAKGGREVCO2CER','PAKGGREVCO2GER', 'PAKGGREVCO2OER']] =
↪30
```

The above used the `pandas` function `.loc[]` to change the Carbon Tax rate variables.

The `modelflow` method `.upd()` could be used to perform the same change.

```
# This modelflow command is equivalent to the previous standard pandas command abive
↪that used the .loc[] syntax
CT30df  =  baseline.upd("<2025 2100> PAKGGREVCO2CER PAKGGREVCO2GER PAKGGREVCO2OER = 30
↪")
```

10.5.1 Solve the model

Solving the model is as simple as calling the mpak function with the altered DataFrame and assigning the results to dataframe (resultsdf in this instance). The keep option causes a copy of the dataframe to be stores within the mpak model obkect.

```
resultsdf = mpak(CT30df,2020,2100,keep="Nominal $30USD Carbon tax") # simulates the␣
↪model
```

Examining the results

Every time the model is solved the results of the simulation are assigned to a variable on the left hands side of the solve call (resultdf in the example above. The results of the most recent scenario are also always stored in the .lastdf DataFrame that is part of the properties of any modelflow model. the basedf property of mpak (an instantiation of a modelflow model object) contains a copy of the initial DataFrame from which the model was built.

The DataFrames Baseline and Resultsdf were created by us when we solved the model (initially on load) and now with the simulation. Currently their contents are the same as, but separate from the contents of basedf and lastdf.

Note: The standard dataframes are part of the modelflow object and managed by it.

- **mpak.basedf**: Dataframe with the values for baseline
- **mpak.lastdf**: Dataframe with the values from the most recent simulation

---

The impact of the imposition of the carbon tax in the model is relatively quick, resulting in an overall decline in emissions of 21.8% in the first year, with coal emissions recording the biggest hit at -40.5 percent.

Abstracting from the fact that the impact is occurring too quickly (it would take time for the substitution towards alternative sources of power to occur), the fact that impacts are fading with time suggests an error in the specification of the shock. High domestic inflation means that the relative price change of a given Carbon price is declining over time.

```
with mpak.set_smpl(2023,2030):
    print(round(mpak['PAKCCEMISCO2*'].difpctlevel.mul100.df,2));
```

```
mpak['PAKCCEMISCO2?KN'].difpctlevel.mul100.plot(title="Emissions impact of a $30 USD␣
↪Carbon tax",showfig=True)
```

Abstracting from the fact that the impact is occurring too quickly (it would take time for the substitution towards alternative sources of power to occur), the fact that impacts are fading with time suggests an error in the specification of the shock. High domestic inflation means that the relative price change of a given Carbon price is declining over time.

10.6 Re-thinking the shock as an ex-ante real shock

Inflation in Pakistan is relatively high so a $30 shock quickly loses its relative price effect. Increasing the nominal value of the Carbon Tax by the amount of domestic inflation (converted into USD each year) would resolve the problem.

Below a new dataframe is created as acopy of the baseline and the three Carbon taxes are first set to $30 in 2025 and then grown at the rate of domestic inflation to keep the relative price of the Carbon Tax constant.

Finally the model is re-solved.

```
import modelmf  # import the mfcalc functionality and append it to standard pandas
CT30realdf  =  baseline.copy()
CT30realdf=CT30realdf.upd("<2025 2025> PAKGGREVCO2CER PAKGGREVCO2OER PAKGGREVCO2GER =␣
↪30")

CT30realdf=CT30realdf.mfcalc('''
                              <2026 2100> PAKGGREVCO2CER = PAKGGREVCO2CER(-
↪1)*(PAKNECONPRVTXN*PAKPANUSATLS)/(PAKNECONPRVTXN(-1)*PAKPANUSATLS(-1))
                                      PAKGGREVCO2OER = PAKGGREVCO2OER(-
↪1)*(PAKNECONPRVTXN*PAKPANUSATLS)/(PAKNECONPRVTXN(-1)*PAKPANUSATLS(-1))
                                      PAKGGREVCO2GER = PAKGGREVCO2CER(-
↪1)*(PAKNECONPRVTXN*PAKPANUSATLS)/(PAKNECONPRVTXN(-1)*PAKPANUSATLS(-1))
                          ''')

CT30realdf.loc[2023:2030,'PAKGGREVCO2CER']


resultsdf = mpak(CT30realdf,2020,2100,keep="Real $30USD Carbon tax") # simulates the␣
↪model
```

```
with mpak.set_smpl(2023,2030):
    print(mpak['PAKGG*ER'].df)
```

```
mpak['PAKCCEMISCO2?KN'].difpctlevel.mul100.plot(title="Emissions impact of a $30 USD␣
↪Carbon tax",showfig=True)
```

These results are better, but still there is an erosion of the effect of the tax.

On introspection, this is likely due to the fact that the carbon tax itself is inflationary. As a result, prices probably rose to a higher level than supposed by the ex ante calculation.

To deal with this, a different approach is needed. Rather than maintaining the carbon price as an exogenous variable, instead it should be made an endogenous variable by changing the model and adding equations for all of the carbon tax variables.

10.7 Changing the model – modifying and or adding equations

To endogenize the carbon price, an equation for each carbon price has to be added to the model. This cane be done with the `.equpdate()` method.

```
mpak1,baseline = model.modelload('../models/pak.pcim',alfa=0.7,run=1,keep="Baseline")
mpakreal,baselinereal = mpak1.equpdate('''
<fixable> PAKGGREVCO2CER = PAKGGREVCO2CER(-1) * (PAKNECONPRVTXN*PAKPANUSATLS) /␣
↪(PAKNECONPRVTXN(-1)*PAKPANUSATLS(-1))
```

(continues on next page)

(continued from previous page)

```
<fixable> PAKGGREVCO2OER = PAKGGREVCO2OER(-1) * (PAKNECONPRVTXN*PAKPANUSATLS) /␣
↪(PAKNECONPRVTXN(-1)*PAKPANUSATLS(-1))
<fixable> PAKGGREVCO2GER = PAKGGREVCO2GER(-1) * (PAKNECONPRVTXN*PAKPANUSATLS) /␣
↪(PAKNECONPRVTXN(-1)*PAKPANUSATLS(-1))
''',add_add_factor=False, calc_add=False,newname='Pak model, with real Carbon price␣
↪equations')
```

```
file read:  C:\mflow\modelflow-manual\papers\mfbook\content\models\pak.pcim

The model:"PAK" got new equations, new model name is:"Pak model, with real Carbon␣
↪price equations"
New equation for For PAKGGREVCO2CER
Old frml   :new endogeneous variable
New frml   :FRML <fixable> PAKGGREVCO2CER = (PAKGGREVCO2CER(-
↪1)*(PAKNECONPRVTXN*PAKPANUSATLS)/(PAKNECONPRVTXN(-1)*PAKPANUSATLS(-1)))* (1-
↪PAKGGREVCO2CER_D)+ PAKGGREVCO2CER_X*PAKGGREVCO2CER_D$
Adjust calc:No frml for adjustment calc

New equation for For PAKGGREVCO2OER
Old frml   :new endogeneous variable
New frml   :FRML <fixable> PAKGGREVCO2OER = (PAKGGREVCO2OER(-
↪1)*(PAKNECONPRVTXN*PAKPANUSATLS)/(PAKNECONPRVTXN(-1)*PAKPANUSATLS(-1)))* (1-
↪PAKGGREVCO2OER_D)+ PAKGGREVCO2OER_X*PAKGGREVCO2OER_D$
Adjust calc:No frml for adjustment calc

New equation for For PAKGGREVCO2GER
Old frml   :new endogeneous variable
New frml   :FRML <fixable> PAKGGREVCO2GER = (PAKGGREVCO2GER(-
↪1)*(PAKNECONPRVTXN*PAKPANUSATLS)/(PAKNECONPRVTXN(-1)*PAKPANUSATLS(-1)))* (1-
↪PAKGGREVCO2GER_D)+ PAKGGREVCO2GER_X*PAKGGREVCO2GER_D$
Adjust calc:No frml for adjustment calc
```

As written, the `.equpdate()` command creates a new model, which is a copy of the existing model with three new equations.

Each equation grows the nominal rate of the carbon tax at the same rate as inflation (`PAKNECONPRVTXN`) converted into USD via the exchange rate `PAKPANUSATLS`. The equations are introduced as exogenizable equations (as distinct from an identity which must always hold), by adding the <fixable> prefix to each equation. The equations are not estimated, so no add-factors are included in the equations.

The output for the `.equpdate()` reports the actual formulae included in the model.

```
New equation for For PAKGGREVCO2CER
Old frml   :new endogeneous variable
New frml   :FRML <fixable> PAKGGREVCO2CER = (PAKGGREVCO2CER(-
↪1)*(PAKNECONPRVTXN*PAKPANUSATLS)/(PAKNECONPRVTXN(-1)*PAKPANUSATLS(-1)))* (1-
↪PAKGGREVCO2CER_D)+ PAKGGREVCO2CER_X*PAKGGREVCO2CER_D$
Adjust calc:No frml for adjustment calc
```

Note that because the equations are to be fixable, an _X and _D variable are added to the specified equations. Combined they effectively split each equation into two:

1. the specified equations when _D equals zero
2. equal to _X when the _D equals one.

The newly created model is given the name mpakreal and is given a text description.

Following the addition of the equations, the new variables (_D and _X) must be initialized. The _X variables are made equal to the current values of the various tax rates, while the _D is set to 1 everywhere – effectively turning the equation off and re-creating the same situation as the initial model where the tax rates are fully exogenous.

```
#Exogenizes the newly added equations and sets the dummy =1 amd the _x to the current␣
↪value of the dependent variable
baseline_real=mpakreal.fix(baselinereal,'PAKGGREVCO2CER PAKGGREVCO2GER PAKGGREVCO2OER
↪')
```

The fix command above effectively does in one line all of the following code.

```
baseline_real=baselinereal.copy()


#Create the _X variables if we exogenize the equation
baseline_real = baseline_real.mfcalc('''
PAKGGREVCO2CER_X = PAKGGREVCO2CER
PAKGGREVCO2GER_X = PAKGGREVCO2GER
PAKGGREVCO2OER_X = PAKGGREVCO2OER
''')

#create the _D varibale so we can exogenize the equations (set _D=1)-- currently it␣
↪is exogenized
baseline_real = baseline_real.upd('''
<-0 -1>
PAKGGREVCO2CER_D PAKGGREVCO2GER_D PAKGGREVCO2OER_D = 1
''')
```

Finally the new model is solved, the result is kept in a new baseline and a quick check ensures that the model did indeed reproduce the data that it was originally fed, including the initial Carbon Tax levels.

```
#Solve the model for the new baseline
res = mpakreal(baseline_real,2021,2100,alfa=0.5,keep='Baseline - adjusted model')

mpakreal['PAKNYGDPMKTPKN PAKNECONPRVTXN PAKGGBALOVRL PAKGGREVCO2CER PAKCCEMISCO2TKN'].
↪difpctlevel.mul100.df
```

```
      PAKNYGDPMKTPKN  PAKNECONPRVTXN  PAKGGREVCO2CER  PAKCCEMISCO2TKN
2021             0.0             0.0            -0.0              0.0
2022             0.0             0.0            -0.0              0.0
2023             0.0             0.0            -0.0              0.0
2024             0.0             0.0            -0.0              0.0
2025             0.0             0.0            -0.0              0.0
...              ...             ...             ...              ...
2096             0.0             0.0            -0.0              0.0
2097             0.0             0.0            -0.0              0.0
2098             0.0             0.0            -0.0              0.0
2099             0.0             0.0            -0.0              0.0
2100             0.0             0.0            -0.0              0.0

[80 rows x 4 columns]
```

10.7.1 Solving the revised model

With the new model generated, it can now be solves with the real tax rate endogenized in the forecast period. This involves three steps.

1. Set the nominal tax rate to 30 in 2024
2. Now Endogenize the equation for the rest of the period
3. Solve the model.

```
scenario_real_CTax = baseline_real.upd('''
<2024 2024>
PAKGGREVCO2CER_x PAKGGREVCO2GER_x PAKGGREVCO2OER_x = 30 # Sets the exogenous value to␣
↪29 in 2024
<2025 2100 >
PAKGGREVCO2CER_D PAKGGREVCO2GER_D PAKGGREVCO2OER_d = 0   # Endogenizes the new␣
↪equations for the rest of time so that the real-rate stays at 30USD
''')


_ = mpakreal(scenario_real_CTax,2021,2100,alfa=0.5,keep='Real model real tax = 30 in␣
↪2022 currency units')
```

Initially the Carbon tax comes in at 30 but gradually its rate in USD rises in line with inflation such that it reaches $1500 by 2100.

```
round(mpakreal['PAKGGREVCO2?ER PAKNECONPRVTXN PAKPANUSATLS'].df,1)
```

```
      PAKGGREVCO2CER  PAKGGREVCO2GER  PAKGGREVCO2OER  PAKNECONPRVTXN  \
2021            -5.5           -41.0            -8.7             1.8
2022            -5.5           -41.0            -8.7             2.0
2023            -5.5           -41.0            -8.7             2.1
2024            30.0            30.0            30.0             2.4
2025            32.1            32.1            32.1             2.5
...              ...             ...             ...             ...
2096          1204.1          1204.1          1204.1           106.8
2097          1270.1          1270.1          1270.1           112.9
2098          1339.8          1339.8          1339.8           119.3
2099          1413.2          1413.2          1413.2           126.1
2100          1490.7          1490.7          1490.7           133.3

      PAKPANUSATLS
2021         107.0
2022         106.8
2023         106.7
2024         106.3
2025         106.2
...            ...
2096          94.3
2097          94.1
2098          93.9
2099          93.7
2100          93.5

[80 rows x 5 columns]
```

This seemingly very high level is just a reflection of the 75 years of inflation that compounded require a much higher nominal Carbon tax rate to have the same relative price effect. The cumulative effect of inflation in the range of 5.5 percent per annum causes the price level to increase 74 times (7400 percent increase 133/1.8 from fourth data column in the above table).

The table below shows the same data but in growth rate terms – indicating that the Carbon tax effective rate is gradually rising each year broadly in line with inflation.

```
mpakreal['PAKGGREVCO2?ER PAKNECONPRVTXN PAKPANUSATLS'].pct.mul100.df
```

```
      PAKGGREVCO2CER  PAKGGREVCO2GER  PAKGGREVCO2OER  PAKNECONPRVTXN  \
2021        0.000000        0.000000        0.000000        9.476828   
2022        0.000000        0.000000        0.000000        8.776453   
2023        0.000000        0.000000        0.000000        7.978008   
2024     -640.556268     -173.169155     -444.405991       10.081669   
2025        6.922650        6.922650        6.922650        7.066610   
...              ...             ...             ...             ...   
2096        5.482659        5.482659        5.482659        5.695883   
2097        5.482423        5.482423        5.482423        5.695627   
2098        5.482121        5.482121        5.482121        5.695299   
2099        5.481761        5.481761        5.481761        5.694908   
2100        5.481351        5.481351        5.481351        5.694461   

      PAKPANUSATLS  
2021     -0.158806  
2022     -0.155332  
2023     -0.137261  
2024     -0.328810  
2025     -0.134458  
...            ...  
2096     -0.201734  
2097     -0.201715  
2098     -0.201691  
2099     -0.201662  
2100     -0.201629  

[80 rows x 5 columns]
```

10.7.2 Results

The results from the simulation with the Carbon Tax rate endogenized so as to maintain its real value over time, are broadly consistent with the results from the ex ante real scenario performed above.

```
mpakreal['PAKCCEMISCO2?KN'].difpctlevel.mul100.df
```

```
      PAKCCEMISCO2CKN  PAKCCEMISCO2GKN  PAKCCEMISCO2OKN  PAKCCEMISCO2TKN
2021         0.000000         0.000000         0.000000         0.000000
2022         0.000000         0.000000         0.000000         0.000000
2023         0.000000         0.000000         0.000000         0.000000
2024       -42.758320       -28.536509       -11.513706       -23.275423
2025       -43.001668       -27.652364       -12.167191       -23.390856
...               ...              ...              ...              ...
2096       -26.343289       -12.356617        -5.804034       -12.147898
2097       -26.200340       -12.274552        -5.753062       -12.067369
2098       -26.057915       -12.192940        -5.702492       -11.987293
```

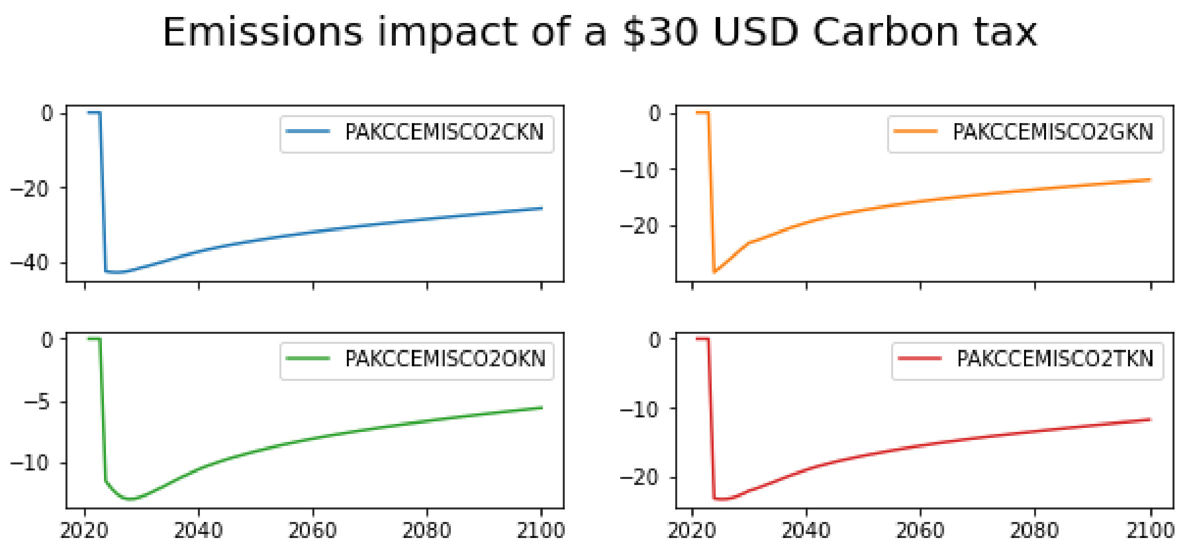
(continues on next page)

(continued from previous page)

```
2099        -25.915998       -12.111773        -5.652313       -11.907659
2100        -25.774578       -12.031041        -5.602516       -11.828457

[80 rows x 4 columns]
```

```
mpakreal['PAKCCEMISCO2?KN'].difpctlevel.mul100.plot(title="Emissions impact of a $30␣
↪USD Carbon tax",showfig=True)
```

```
C:\Users\wb268970\.conda\envs\modelflow\lib\site-packages\IPython\core\pylabtools.
↪py:151: UserWarning: This figure was using constrained_layout, but that is␣
↪incompatible with subplots_adjust and/or tight_layout; disabling constrained_
↪layout.
  fig.canvas.print_figure(bytes_io, **kw)
```



Part IV

Model Analytics

```
%matplotlib inline
```

CHAPTER

ELEVEN

MODEL EIGENVALUES

Eigenvalues are a fundamental concept in dynamic models. In simple terms, they summarize the adjustment process within a model. In the context of dynamic models, the eigenvalues of the model describe the behavior of the system over time. The sign and magnitude of the eigenvalues determine whether a system of equations will converge to a stable equilibrium, oscillate, or diverge. For macro models they determine whether the model is stable, marginally stable, or unstable.

In the case of a macromodel, which is effectively a system of differential equations, the eigenvalues of the coefficient matrix determine whether the system is stable or unstable. If all the eigenvalues have negative real parts, then the system is stable and will converge to a steady state over time. If at least one eigenvalue has a positive real part, then the system is unstable, and the solutions will diverge over time.

This Notebook uses a model for Pakistan described here:

11.1 Imports

Modelflow's modelclass includes most of the methods needed to manage a model in Modelflow.

```
from modelclass import model
from modelnewton import newton_diff
import modelmf
model.widescreen()
model.scroll_off()
```

```
<IPython.core.display.HTML object>
```

11.2 Load a pre-existing model, data and descriptions

The file `pak.pcim` contains a dump of model equations, dataframe, simulation options and variable descriptions. The file has been created when onboarding the model. Examples can be found here

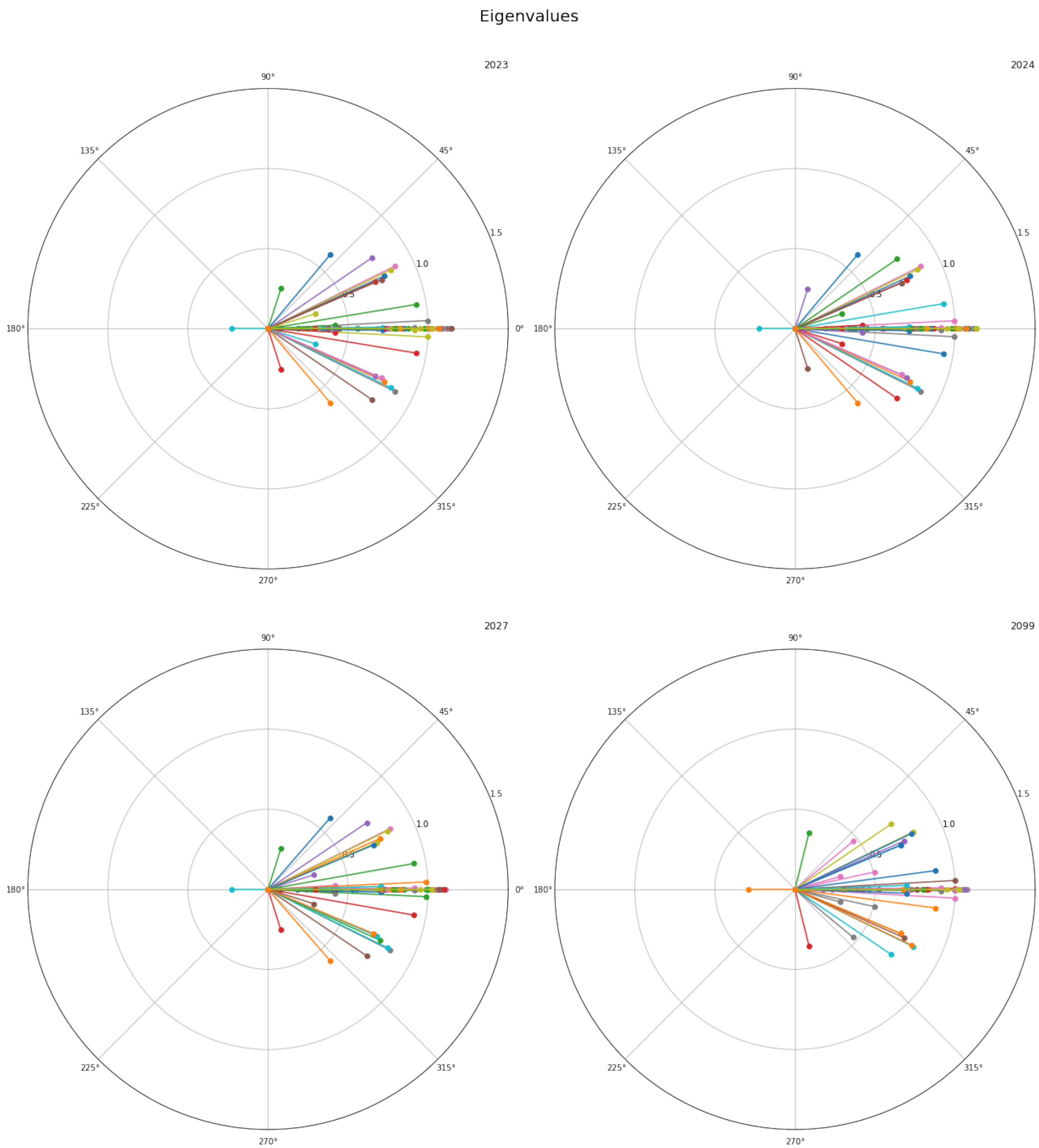
```
mpak,baseline = model.modelload('../../models/pak.pcim',alfa=0.7,run=1)
```

```
Open file from URL:  https://raw.githubusercontent.com/IbHansen/modelflow-manual/
↪main/model_repo/pak.pcim
```

```
mpak_newton = newton_diff(mpak,forcenum=0)       # create a newton_diff instance which␣
↪contains derivatives
```

```
eig_dic = mpak_newton.get_eigenvectors(filnan = True,periode= (2023,2024,2027,2099),
 ↪silent=True) #
mpak_newton.eigplot_all(eig_dic,size=(3,3));
```

```
C:\Users\wb268970\.conda\envs\modelflow\lib\site-packages\IPython\core\events.
 ↪py:89: UserWarning: This figure was using constrained_layout, but that is␣
 ↪incompatible with subplots_adjust and/or tight_layout; disabling constrained_
 ↪layout.
  func(*args, **kwargs)
```



```
help(mpak_newton)
```

```
Help on newton_diff in module modelnewton object:

class newton_diff(builtins.object)
 |  newton_diff(mmodel, df=None, endovar=None, onlyendocur=False, timeit=False,
↪silent=True, forcenum=False, per='', ljit=0, nchunk=None, endoandexo=False)
 |
 |  Class to handle newron solving
 |  this is for un-nomalized or normalized models ie models of the forrm
 |
 |  0 = G(y,x)
 |  y = F(y,x)
 |
 |  Methods defined here:
 |
 |  __init__(self, mmodel, df=None, endovar=None, onlyendocur=False, timeit=False,
↪silent=True, forcenum=False, per='', ljit=0, nchunk=None, endoandexo=False)
 |      Args:
 |          mmodel (TYPE): Model to analyze.
 |          df (TYPE, optional): Dataframe. if None mmodel.lastdf will be used
 |          endovar (TYPE, optional): if set defines which endogeneous to include .
↪ Defaults to None.
 |          onlyendocur (TYPE, optional): Only calculate for the curren
↪endogeneous variables. Defaults to False.
 |          timeit (TYPE, optional): writeout time informations . Defaults to
↪False.
 |          silent (TYPE, optional):  Defaults to True.
 |          forcenum (TYPE, optional): Force differentiation to be numeric else
↪try sumbolic (slower)  Defaults to False.
 |          per (TYPE, optional): Period for which to calculte the jacobi .
↪Defaults to ''.
 |          ljit (TYPE, optional): Trigger just in time compilation of the
↪differential coiefficient. Defaults to 0.
 |          nchunk (TYPE, optional): Chunks for which the model is written  -
↪relevant if ljit == True. Defaults to None.
 |          endoandexo (TYPE, optional): Calculate for both endogeneous and
↪exogeneous . Defaults to False.
 |
 |      Returns:
 |          None.
 |
 |  eigenvector_plot(self, per=None, size=(4, 3), top=0.9)
 |
 |  eigplot(self, eig_dic=None, per=None, size=(4, 3), top=0.9)
 |
 |  eigplot_all(self, eig_dic, size=(4, 3), maxfig=6)
 |
 |  eigplot_all0(self, eig_dic, size=(4, 3))
 |
 |  get_diff_df_1per(self, df=None, periode=None)
 |
 |  get_diff_df_tot(self, periode=None, df=None)
 |
 |  get_diff_mat_1per(self, periode=None, df=None)
 |      fetch a dict of one periode sparse jacobimatrices
```

(continues on next page)

(continued from previous page)

```
|
|  get_diff_mat_all_1per(self, periode=None, df=None, asdf=False)
|
|  get_diff_mat_tot(self, df=None)
|      Fetch a stacked jacobimatrix for the whole model.current_per
|
|      Returns a sparse matrix.
|
|  get_diff_melted(self, periode=None, df=None)
|      returns a tall matrix with all values to construct jacobimatrix(es)
|
|  get_diff_melted_var(self, periode=None, df=None)
|      makes dict with all  derivative matrices for all lags
|
|  get_diff_values_all(self, periode=None, df=None, asdf=False)
|      stuff the values of derivatives into nested dic
|
|  get_diffmodel(self)
|      Returns a model which calculates the partial derivatives of a model
|
|  get_eigenvectors(self, periode=None, asdf=True, filnan=False, silent=False)
|
|  get_solve1per(self, df=None, periode=None)
|
|  get_solve1perlu(self, df='', periode='')
|
|  get_solvestacked(self, df='')
|
|  get_solvestacked_it(self, df='', solver=<function bicg at 0x000001DC1DC938B0>)
|
|  modeldiff(self)
|      Differentiate relations for self.enovar with respect to endogeneous␣
↪variable
|      The result is placed in a dictory in the model instanse: model.diffendocur
|
|  show_diff(self, pat='*')
|      Displays espressions for differential koifficients for a variable
|      if var ends with * all matchning variables are displayes
|
|  show_diff_latex(self, pat='*', show_expression=True, show_values=True,␣
↪maxper=5)
|
|  show_stacked_diff(self, time=None, lhs='', rhs='', dec=2, show=True)
|      Parameters
|      ----------
|      time : list, optional
|          DESCRIPTION. The default is None. Time for which to retrieve stacked␣
↪jacobi
|      lhs : string, optional
|          DESCRIPTION. The default is ''. Left hand side variables
|      rhs : TYPE, optional
|          DESCRIPTION. The default is ''. Right hand side variabnles
|      dec : TYPE, optional
|          DESCRIPTION. The default is 2.
|      show : TYPE, optional
|          DESCRIPTION. The default is True.
```

(continues on next page)

(continued from previous page)

```
 |
 |      Returns
 |      -------
 |      selected rows and columns of stacked jacobi as dataframe .
 |
 |  ----------------------------------------------------------------------
 |  Static methods defined here:
 |
 |  get_feedback(eig_dic, per=None)
 |      Returns a dict of max abs eigenvector and the sign
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

CHAPTER

TWELVE

ATTRIBUTION

When working with a model it is often useful to have a better sense of the contribution of different channels to a final result. For example, an increase in interest rates will tend to reduce investment and consumer demand – contributing to a reduction in GDP. At the same time, lower inflation as the higher interest rate takes effect will tend to work in the opposite direction.

The modelflow method `.attributon` calculates the contributions of each channel to the overall result.

12.1 Prepare the workspace

As always before running `modelflow` the python environment needs to be initialized and libraries to be used imported.

12.1.1 Load the pre-existing model, data and descriptions

The file `pak.pcim` contains a dump of model equations, dataframe, simulation options and variable descriptions:

- Loads model and simulates to establish a baseline.
- Creates a dataframe with a tax rate of 29 USD/Ton for carbon emission for 3 sectors.
- Simulates the new experiment.

```
mpak,baseline = model.modelload('../models/pak.pcim',alfa=0.7,run=1,keep='Business as␣
↪Usual')
alternative  =  baseline.upd("<2020 2100> PAKGGREVCO2CER PAKGGREVCO2GER␣
↪PAKGGREVCO2OER = 29")
result = mpak(alternative,2020,2100,keep='Carbon tax nominal 29') # simulates the␣
↪model
```

```
file read:  C:\mflow\modelflow-manual\papers\mfbook\content\models\pak.pcim
```

12.2 The mathematics of attribution

At its root the idea of attribution is simply taking the total derivative of the model to identify the sensitivity of the equation we are interested in to changes elsewhere in the model and then combine that with the changes in other variables.

Take a variable y that is a function of two other variables a and b. In the model the relationship might be written as:

$y = f(a, b)$

If there are two observations

$$y_0 = f(a_0, b_0) \tag{12.1}$$
$$y_1 = f(a_1, b_1) \tag{12.2}$$

then we also have the change in all three variables $\Delta y, \Delta a, \Delta b$ and the total derivative of y can be written as:

$\Delta y = \underbrace{\Delta a \frac{\partial f}{\partial a}(a, b)}_{\Omega a} + \underbrace{\Delta b \frac{\partial f}{\partial b}(a, b)}_{\Omega b} + Residual$

The first expresion can be called Ω_a or the contribution of changes in a to changes in y, and the second Ω_b or the contribution fo changes in b to changes in y.

The `modelflow` method `.totdif()` is used to calculate attributions. It performs a numerical approximation of Ω_a and Ω_b by performing two runs of the model:

$$y_0 \quad = \quad f(a_0, b_0) \tag{12.3}$$
$$y_1 \quad = \quad f(a_0 + \Delta a, b_0 + \Delta b) \tag{12.4}$$

and calculates Ω_a and Ω_b as:

$$\Omega f_a \quad = \quad f(a_1, b_1) - f(a_1 - \Delta a, b_1) \tag{12.5}$$
$$\Omega f_b \quad = \quad f(a_1, b_1) - f(a_1, b_1 - \Delta b) \tag{12.6}$$

And:

$$residual = \Omega f_a + \Omega f_b - (y_1 - y_0) \tag{12.7}$$

If the model is fairly linear, the residual will be small.

12.3 Model attribution or single equation attribution?

Above the relationship between y, a, and b was summarized by the function f().

$f(a, b)$ could represent a single equation in the model or it could represent the entire model. In the first instance, Δa and Δb would be treated as exogenous variables in the attribution calculation. In the second instance, they would be all of the endogenous and exoegnous variables that directly and indirectly impact y.

Assume the simple equation example such that a and b are simple variables. When Δy, Δa and Δb reflect the difference across scenarios (say the value of the three variables in `.lastdf` less the value in `.basedf` then;

Ω_a, Ω_b are the absolute contribution of a and b to the change in y, and $100 * \left[\frac{\Omega_a}{\Delta y}\right]$ $100 * \left[\frac{\Omega_b}{\Delta y}\right]$ are the share of the change in y explained by a and b respectively.

If Δy, Δa and Δb are the changes over time ($\Delta y_t = y_t - y_{t-1}$), then Ω_a, Ω_b are the contributions of a and b to the rate of growth of y, while $100 * \left[\frac{\Omega_a}{\Delta y_{t-1}}\right]$ $100 * \left[\frac{\Omega_b}{\Delta y_{t-1}}\right]$ are are the contributions of a and b to the rate of growth of y.

12.4 Formula attribution

Attribution analysis on the formula level is performed by the method **.dekomp**.

This method utilizes that two attributes .basedf and .lastdf containing the first and the last run are contained in the model instance. Also all the formulas are contained in the instance. Therefore a model, just with one formula - is created. Then experiments mentioned above is run for each period and each right hand side variable.

12.4.1 Single equation attribution output

The `dekomp()` method calculates the contribution to changes in the level of the dependent variable in a given equation. In the example below the contribution to the change in Total emissions is decomposed into the contribution from each of three sources in the model, the consumption of Crude Oil, Natural Gas and Coal. As the equation for total emissions is just the sum of the three this is a fairly trivial decomposition, but it provides an easily understood illustration of the process at work.

The results of the `.dekomp` command are divided into 4 separate tables.

1. The first table of output shows the changes that are to be explained/ **The changes are always drawn from the most solution, i.e. from the `.basedf` and `.lastdf` DataFrames** of the model object.
2. The seond table shows the changes between the contributions of the LHS variables to the changes in the RHS variable. Because the equation is just and additive identity, these amount to the changes in each of the variables themselves.
3. the third table expresses these changes as a share of the total change.
4. The last shows the contributions of these changes to the change in the growth rate of the deopendent variable (these results would need to be multiplied by 100 to see that they add to the totals in table 1.

```
with mpak.set_smpl(2020,2025):
    mpak['PAKCCEMISCO2TKN'].dekomp()
```

```
Formula        : FRML <IDENT> PAKCCEMISCO2TKN =␣
↪PAKCCEMISCO2CKN+PAKCCEMISCO2OKN+PAKCCEMISCO2GKN $

                        2020         2021         2022         2023         2024  ␣
↪      2025
Variable    lag
Base         0   213515545.24 217548186.56 221072469.97 225253519.79 230370294.27␣
↪236240658.85
Alternative 0   154655290.66 159024407.00 163163619.20 168028201.90 173929277.95␣
↪180696656.18
Difference  0   -58860254.58 -58523779.56 -57908850.77 -57225317.88 -56441016.32 -
↪55544002.66
Percent     0         -27.57       -26.90       -26.19       -25.40       -24.50  ␣
↪    -23.51

 Contributions to differende for  PAKCCEMISCO2TKN
                            2020         2021         2022         2023        ␣
↪2024         2025
Variable        lag
PAKCCEMISCO2CKN 0   -23834819.21 -23889797.62 -23777548.63 -23622076.35 -23444796.
↪31 -23239969.29
PAKCCEMISCO2OKN 0   -13887796.96 -14589014.38 -15070531.59 -15363597.05 -15456777.
↪50 -15387859.25
```

(continues on next page)

(continued from previous page)

```
PAKCCEMISCO2GKN 0   -21137638.41 -20044967.56 -19060770.55 -18239644.49 -17539442.
↪51 -16916174.12

 Share of contributions to differende for  PAKCCEMISCO2TKN
                           2020        2021        2022        2023        2024   ␣
↪    2025
Variable        lag
PAKCCEMISCO2CKN 0           40%         41%         41%         41%         42%   ␣
↪     42%
PAKCCEMISCO2GKN 0           36%         34%         33%         32%         31%   ␣
↪     30%
PAKCCEMISCO2OKN 0           24%         25%         26%         27%         27%   ␣
↪     28%
Total           0          100%        100%        100%        100%        100%   ␣
↪    100%
Residual        0            0%          0%          0%          0%          0%   ␣
↪      0%

 Contribution to growth rate PAKCCEMISCO2TKN
                           2020        2021        2022        2023        2024   ␣
↪    2025
Variable        lag
PAKCCEMISCO2CKN 0         -0.1%       -0.2%       -0.1%       -0.1%       -0.1%   ␣
↪   -0.1%
PAKCCEMISCO2OKN 0         -0.1%       -0.1%       -0.1%       -0.1%       -0.1%   ␣
↪   -0.1%
PAKCCEMISCO2GKN 0         -0.1%       -0.1%       -0.1%       -0.1%       -0.1%   ␣
↪   -0.1%
```

The following single-equation decomposition looks to the impact on inflation. The inflation equation is more complex and has more direct causal variables, so here the decomposition is more helpful.

Recall the inflation equation is given by the `.frml` method for its normalized version and `.eviews` for its original specification. The equation for the consumer price level (PAKNECONPRVTXN) was originally specified in eviews as:

```
mpak['PAKNECONPRVTXN'].eviews
```

```
PAKNECONPRVTXN : @IDENTITY PAKNECONPRVTXN  = ((PAKNECONENGYSH^PAKCESENGYCON)  *␣
↪PAKNECONENGYXN^(1  - PAKCESENGYCON)  + (PAKNECONOTHRSH^PAKCESENGYCON)  *␣
↪PAKNECONOTHRXN^(1  - PAKCESENGYCON))^(1  / (1  - PAKCESENGYCON))
```

When normalized the equation solves for the **level** of the price deflator. It is this normalized equation that is:

```
mpak['PAKNECONPRVTXN'].frml
```

```
PAKNECONPRVTXN : FRML <IDENT> PAKNECONPRVTXN =␣
↪((PAKNECONENGYSH**PAKCESENGYCON)*PAKNECONENGYXN**(1-
↪PAKCESENGYCON)+(PAKNECONOTHRSH**PAKCESENGYCON)*PAKNECONOTHRXN**(1-
↪PAKCESENGYCON))**(1/(1-PAKCESENGYCON)) $
```

Because the normalized equation solves for the level of the price deflator, te decomposition will show the contributions of each explanatory variable to the increase in the price level (not that of the inflation rate).

Note in the Pakistan model, consumer inflation is derived as a CET aggregation of the price of energy goods(PAKNECONENGYXN) and non-energy goods (PAKNECONOTHRXN).

```python
with mpak.set_smpl(2020,2025):
    mpak['PAKNECONPRVTXN'].dekomp()
```

```
Formula        : FRML <IDENT> PAKNECONPRVTXN =␣
↪((PAKNECONENGYSH**PAKCESENGYCON)*PAKNECONENGYXN**(1-
↪PAKCESENGYCON)+(PAKNECONOTHRSH**PAKCESENGYCON)*PAKNECONOTHRXN**(1-
↪PAKCESENGYCON))**(1/(1-PAKCESENGYCON)) $

                      2020      2021      2022      2023      2024      2025
Variable    lag
Base        0         1.67      1.82      1.98      2.14      2.30      2.45
Alternative 0         1.72      1.89      2.06      2.23      2.39      2.55
Difference  0         0.06      0.07      0.08      0.09      0.09      0.10
Percent     0         3.34      3.77      4.04      4.14      4.11      3.99

 Contributions to differende for  PAKNECONPRVTXN
                         2020      2021      2022      2023      2024      ␣
↪2025
Variable          lag
PAKNECONENGYSH 0        -0.00     -0.00     -0.00     -0.00     -0.00     -0.
↪00
PAKCESENGYCON  0        -0.00     -0.00     -0.00     -0.00     -0.00     -0.
↪00
PAKNECONENGYXN 0         0.01      0.01      0.01      0.01      0.01      0.
↪01
PAKNECONOTHRSH 0        -0.00     -0.00     -0.00     -0.00     -0.00     -0.
↪00
PAKNECONOTHRXN 0         0.04      0.06      0.07      0.08      0.08      0.
↪08

 Share of contributions to differende for  PAKNECONPRVTXN
                          2020        2021        2022        2023        2024    ␣
↪   2025
Variable          lag
PAKNECONOTHRXN 0           77%         81%         83%         85%         86%    ␣
↪     86%
PAKNECONENGYXN 0           24%         20%         17%         16%         15%    ␣
↪     14%
PAKNECONENGYSH 0           -0%         -0%         -0%         -0%         -0%    ␣
↪     -0%
PAKCESENGYCON  0           -0%         -0%         -0%         -0%         -0%    ␣
↪     -0%
PAKNECONOTHRSH 0           -0%         -0%         -0%         -0%         -0%    ␣
↪     -0%
Total          0          101%        101%        101%        101%        101%    ␣
↪    100%
Residual       0            1%          1%          1%          1%          1%    ␣
↪      0%

 Contribution to growth rate PAKNECONPRVTXN
                          2020        2021        2022        2023        2024    ␣
↪   2025
Variable          lag
PAKNECONENGYSH 0         -0.0%       -0.0%       -0.0%       -0.0%       -0.0%    ␣
↪   -0.0%
PAKCESENGYCON  0         -0.0%       -0.0%       -0.0%       -0.0%       -0.0%    ␣
↪   -0.0%
```

(continues on next page)

(continued from previous page)

```
PAKNECONENGYXN 0           0.0%        0.0%        0.0%        0.0%        0.0%    ␣
↪    0.0%
PAKNECONOTHRSH 0          -0.0%       -0.0%       -0.0%       -0.0%       -0.0%    ␣
↪   -0.0%
PAKNECONOTHRXN 0           0.0%        0.0%        0.0%        0.0%        0.0%    ␣
↪    0.0%
```

Interestingly only 25% of the increase in the price level each period is due to the direct channel (the impact on energy prices), the bulk of the increase comes indirectly through other prices. Indeed as time progresses this share rises from 75% in the first year of the price change (2020) to 86% by 2024.

Below is the formula for nonenergy consumer prices and its decomposition. This equation is written out as a more standard inflation equation reflecting changes in the cost of local goods production (PAKNYGDPFCSTXN), Government taxes on goods and services (PAKGGREVGNFSXN), the price of imports (PAKNEIMPGNGSXN) and the influence of the economic cycle (PAKNYGDPGAP_).

```
mpak['PAKNECONOTHRXN'].eviews
```

```
PAKNECONOTHRXN : DLOG(PAKNECONOTHRXN) = 0.590372627657176*DLOG(PAKNYGDPFCSTXN) +␣
↪D(PAKGGREVGNFSXN/100) + (1 - 0.590372627657176)*DLOG(PAKNEIMPGNFSXN) + 0.
↪2*PAKNYGDPGAP_/100
```

```
with mpak.set_smpl(2020,2025):
    mpak['PAKNECONOTHRXN'].dekomp()
```

```
Formula        : FRML <DAMP,STOC> PAKNECONOTHRXN = (PAKNECONOTHRXN(-
↪1)*EXP(PAKNECONOTHRXN_A+ (0.590372627657176*((LOG(PAKNYGDPFCSTXN))-
↪(LOG(PAKNYGDPFCSTXN(-1))))+((PAKGGREVGNFSXN/100)-(PAKGGREVGNFSXN(-1)/100))+(1-0.
↪590372627657176)*((LOG(PAKNEIMPGNFSXN))-(LOG(PAKNEIMPGNFSXN(-1))))+0.
↪2*PAKNYGDPGAP_/100) )) * (1-PAKNECONOTHRXN_D)+ PAKNECONOTHRXN_X*PAKNECONOTHRXN_D␣
↪ $

                      2020       2021       2022       2023       2024       2025
Variable    lag
Base        0         1.70       1.86       2.02       2.18       2.34       2.50
Alternative 0         1.74       1.92       2.09       2.26       2.43       2.59
Difference  0         0.04       0.06       0.07       0.08       0.08       0.09
Percent     0         2.63       3.12       3.44       3.59       3.60       3.51

 Contributions to differende for  PAKNECONOTHRXN
                           2020       2021       2022       2023       2024      ␣
↪2025
Variable           lag
PAKNECONOTHRXN     -1     -0.00       0.05       0.06       0.08       0.08      ␣
↪0.09
PAKNECONOTHRXN_A    0     -0.00      -0.00      -0.00      -0.00      -0.00      -
↪0.00
PAKNYGDPFCSTXN      0      0.00       0.01       0.01       0.02       0.02      ␣
↪0.02
                   -1     -0.00      -0.00      -0.01      -0.01      -0.02      -
↪0.02
PAKGGREVGNFSXN      0     -0.00      -0.00      -0.00      -0.00      -0.00      -
↪0.00
                   -1     -0.00      -0.00      -0.00      -0.00      -0.00      -
↪0.00
```

(continues on next page)

(continued from previous page)

```
PAKNEIMPGNFSXN     0         0.04        0.04        0.05        0.05        0.05       ␣
↪0.05
                  -1        -0.00       -0.05       -0.05       -0.05       -0.05       -
↪0.05
PAKNYGDPGAP_       0         0.00        0.00        0.00        0.00        0.00       -
↪0.00
PAKNECONOTHRXN_D   0        -0.00       -0.00       -0.00       -0.00       -0.00       -
↪0.00
PAKNECONOTHRXN_X   0        -0.00       -0.00       -0.00       -0.00       -0.00       -
↪0.00

 Share of contributions to differende for  PAKNECONOTHRXN
                            2020        2021        2022        2023        2024  ␣
↪      2025
Variable          lag
PAKNECONOTHRXN    -1          -0%         85%         91%         96%        100%  ␣
↪      102%
PAKNEIMPGNFSXN     0          92%         75%         66%         61%         58%  ␣
↪       56%
PAKNYGDPFCSTXN     0           2%         13%         19%         23%         26%  ␣
↪       28%
PAKNECONOTHRXN_A   0          -0%         -0%         -0%         -0%         -0%  ␣
↪       -0%
PAKGGREVGNFSXN     0          -0%         -0%         -0%         -0%         -0%  ␣
↪       -0%
                  -1          -0%         -0%         -0%         -0%         -0%  ␣
↪       -0%
PAKNECONOTHRXN_D   0          -0%         -0%         -0%         -0%         -0%  ␣
↪       -0%
PAKNECONOTHRXN_X   0          -0%         -0%         -0%         -0%         -0%  ␣
↪       -0%
PAKNYGDPGAP_       0           7%          7%          4%          2%          0%  ␣
↪       -1%
PAKNYGDPFCSTXN    -1          -0%         -2%        -12%        -19%        -24%  ␣
↪      -27%
PAKNEIMPGNFSXN    -1          -0%        -79%        -70%        -65%        -62%  ␣
↪      -60%
Total              0         100%         99%         99%         99%         99%  ␣
↪       99%
Residual           0           0%         -1%         -1%         -1%         -1%  ␣
↪       -1%

 Contribution to growth rate PAKNECONOTHRXN
                            2020        2021        2022        2023        2024  ␣
↪      2025
Variable          lag
PAKNECONOTHRXN    -1         -0.0%       -0.0%       -0.0%       -0.0%       -0.0%  ␣
↪     -0.0%
PAKNECONOTHRXN_A   0         -0.0%       -0.0%       -0.0%       -0.0%       -0.0%  ␣
↪     -0.0%
PAKNYGDPFCSTXN     0          0.0%        0.0%        0.0%        0.0%        0.0%  ␣
↪      0.0%
                  -1         -0.0%       -0.0%       -0.0%       -0.0%       -0.0%  ␣
↪     -0.0%
PAKGGREVGNFSXN     0         -0.0%       -0.0%       -0.0%       -0.0%       -0.0%  ␣
↪     -0.0%
```

(continues on next page)

(continued from previous page)

```
                  -1         -0.0%       -0.0%       -0.0%       -0.0%       -0.0%  ␣
↪     -0.0%
PAKNEIMPGNFSXN     0          0.0%        0.0%        0.0%        0.0%        0.0%  ␣
↪      0.0%
                  -1         -0.0%       -0.0%       -0.0%       -0.0%       -0.0%  ␣
↪     -0.0%
PAKNYGDPGAP_       0          0.0%        0.0%        0.0%        0.0%        0.0%  ␣
↪     -0.0%
PAKNECONOTHRXN_D   0         -0.0%       -0.0%       -0.0%       -0.0%       -0.0%  ␣
↪     -0.0%
PAKNECONOTHRXN_X   0         -0.0%       -0.0%       -0.0%       -0.0%       -0.0%  ␣
↪     -0.0%
```

These results are more interesting and indicate that much of the initial impact on prices is coming the increase in the price of imported goods (which includes a large fuel component). As time progresses the imported inflation component declines and the lagged consumption effect dominates. Other factors such as the cost of domestically produced goods play a larger role and the net impact of imported prices (the total of the contemporaneous and lagged value) approaches zero. Cyclical pressure are initially adding to inflation before declining and eventually turning negative.

```
mpak['PAKNECONOTHRXN'].dif.df
```

```
      PAKNECONOTHRXN
2020        0.044582
2021        0.058001
2022        0.069577
2023        0.078417
2024        0.084378
...              ...
2096        0.424055
2097        0.443432
2098        0.463762
2099        0.485087
2100        0.507454

[81 rows x 1 columns]
```

The dekomp_plot with option pct=False shows the change in the level of the LHS variable decomposed into the contribution of

```
control,delta,contributions=mpak.dekomp('PAKNECONOTHRXN',lprint=False)
mpak.dekomp('PAKNECONOTHRXN',lprint=False)
control
```

```
                      2020      2021      2022      2023      2024      2025  \
Variable    lag
Base        0     1.695682  1.857535  2.021492  2.183344  2.342145   2.49833
Alternative 0     1.740265  1.915535  2.091069  2.261761  2.426523  2.586135
Difference  0     0.044582  0.058001  0.069577  0.078417  0.084378  0.087805
Percent     0     2.629162  3.122452  3.441877  3.591603   3.60259  3.514546

                      2026      2027      2028      2029  ...       2091  \
Variable    lag                                         ...
Base        0     2.652585    2.8055  2.957674  3.109926  ...  79.631396
Alternative 0     2.741785  2.894545  3.045414  3.195536  ...   79.97147
```

(continues on next page)

(continued from previous page)

```
Difference  0      0.0892  0.089045   0.08774   0.08561  ...   0.340074
Percent     0    3.362743  3.173934  2.966501  2.752779  ...    0.42706

                     2092       2093       2094       2095        2096  \
Variable    lag
Base        0    84.163096  88.952875  94.015343  99.365938  105.020967
Alternative 0    84.518369   89.32411  94.403339  99.771527  105.445022
Difference  0     0.355273   0.371235   0.387995   0.405589    0.424055
Percent     0     0.422124   0.417339   0.412693   0.408177    0.403781

                      2097        2098        2099        2100
Variable    lag
Base        0    110.997662   117.31423  123.989906  131.045014
Alternative 0    111.441094  117.777991  124.474993  131.552467
Difference  0      0.443432    0.463762    0.485087    0.507454
Percent     0      0.399497    0.395316    0.391231    0.387236

[4 rows x 81 columns]
```

```
delta
```

```
                          2020      2021      2022      2023      2024  \
Variable         lag
PAKNECONOTHRXN   -1       -0.0  0.049072  0.063316  0.075257   0.08413
PAKNECONOTHRXN_A  0       -0.0      -0.0      -0.0      -0.0      -0.0
PAKNYGDPFCSTXN    0   0.000827  0.007618   0.01354  0.018401  0.022052
                 -1       -0.0 -0.000911 -0.008349 -0.014741 -0.019903
PAKGGREVGNFSXN    0       -0.0      -0.0      -0.0      -0.0      -0.0
                 -1       -0.0      -0.0      -0.0      -0.0      -0.0
PAKNEIMPGNFSXN    0   0.040852  0.043658  0.045895  0.047511  0.048539
                 -1       -0.0 -0.046047  -0.04877 -0.050755 -0.052066
PAKNYGDPGAP_      0   0.002994  0.003799  0.003018  0.001664  0.000411
PAKNECONOTHRXN_D  0       -0.0      -0.0      -0.0      -0.0      -0.0
PAKNECONOTHRXN_X  0       -0.0      -0.0      -0.0      -0.0      -0.0

                          2025      2026      2027      2028      2029  ...  \
Variable         lag                                                    ...
PAKNECONOTHRXN   -1   0.089928   0.09309   0.09417  0.093686  0.092065  ...
PAKNECONOTHRXN_A  0       -0.0      -0.0      -0.0      -0.0      -0.0  ...
PAKNYGDPFCSTXN    0   0.024598   0.02621  0.027065  0.027309  0.027066  ...
                 -1  -0.023718 -0.026328 -0.027938 -0.028744 -0.028915  ...
PAKGGREVGNFSXN    0       -0.0      -0.0      -0.0      -0.0      -0.0  ...
                 -1       -0.0      -0.0      -0.0      -0.0      -0.0  ...
PAKNEIMPGNFSXN    0   0.049082  0.049244   0.04911  0.048747   0.04821  ...
                 -1  -0.052788 -0.053043 -0.052938 -0.052561 -0.051982  ...
PAKNYGDPGAP_      0  -0.000602 -0.001313 -0.001753 -0.001981 -0.002051  ...
PAKNECONOTHRXN_D  0       -0.0      -0.0      -0.0      -0.0      -0.0  ...
PAKNECONOTHRXN_X  0       -0.0      -0.0      -0.0      -0.0      -0.0  ...

                          2091      2092      2093      2094      2095  \
Variable         lag
PAKNECONOTHRXN   -1   0.344117   0.35941  0.375474  0.392345  0.410058
PAKNECONOTHRXN_A  0       -0.0      -0.0      -0.0      -0.0      -0.0
PAKNYGDPFCSTXN    0   0.155984  0.163537  0.171467  0.179792  0.188531
                 -1  -0.157562 -0.165175 -0.173171 -0.181566 -0.190379
```

(continues on next page)

(continued from previous page)

```
PAKGGREVGNFSXN     0        -0.0       -0.0       -0.0       -0.0       -0.0
                  -1        -0.0       -0.0       -0.0       -0.0       -0.0
PAKNEIMPGNFSXN     0    0.041552   0.041988    0.04245   0.042939   0.043457
                  -1   -0.043505  -0.043937  -0.044397  -0.044885  -0.045402
PAKNYGDPGAP_       0   -0.000859  -0.000908  -0.000959  -0.001015  -0.001075
PAKNECONOTHRXN_D   0        -0.0       -0.0       -0.0       -0.0       -0.0
PAKNECONOTHRXN_X   0        -0.0       -0.0       -0.0       -0.0       -0.0

                            2096       2097       2098       2099       2100
Variable          lag
PAKNECONOTHRXN    -1    0.428653   0.448169   0.468647   0.490132   0.512668
PAKNECONOTHRXN_A   0        -0.0       -0.0       -0.0       -0.0       -0.0
PAKNYGDPFCSTXN     0    0.197701   0.207322   0.217415   0.228002   0.239103
                  -1   -0.199629  -0.209335   -0.21952  -0.230203  -0.241408
PAKGGREVGNFSXN     0        -0.0       -0.0       -0.0       -0.0       -0.0
                  -1        -0.0       -0.0       -0.0       -0.0       -0.0
PAKNEIMPGNFSXN     0    0.044005   0.044583   0.045193   0.045835   0.046511
                  -1   -0.045949  -0.046527  -0.047137  -0.047781  -0.048459
PAKNYGDPGAP_       0   -0.001139  -0.001207  -0.001279  -0.001357  -0.001439
PAKNECONOTHRXN_D   0        -0.0       -0.0       -0.0       -0.0       -0.0
PAKNECONOTHRXN_X   0        -0.0       -0.0       -0.0       -0.0       -0.0

[11 rows x 81 columns]
```

```
contributions # as a percent of the total change (the omegas)
```

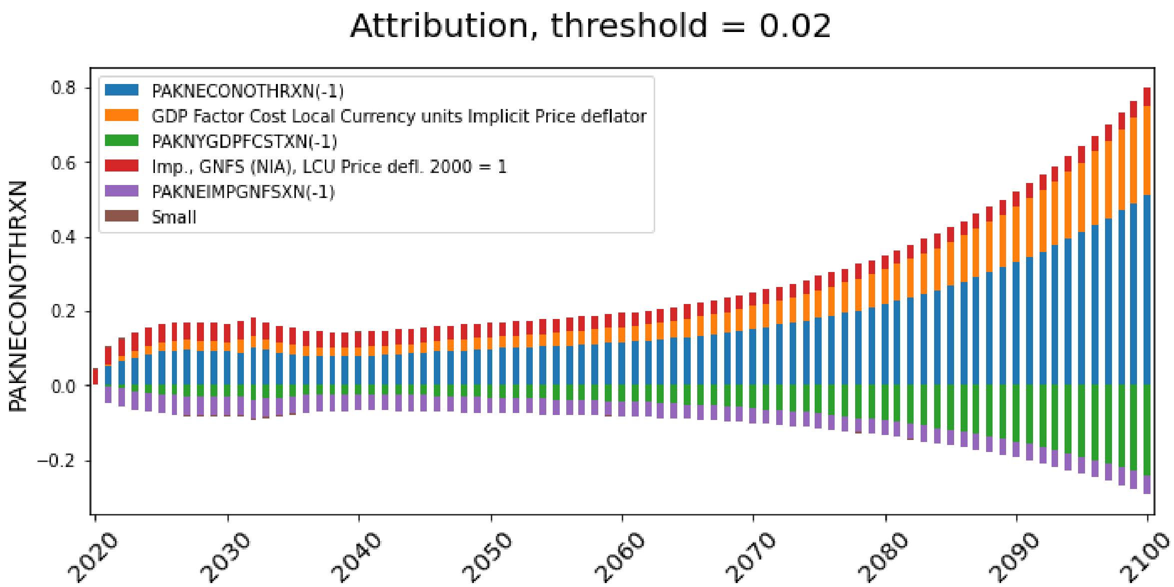
```
                              2020          2021          2022          2023  \
Variable          lag
PAKNECONOTHRXN    -1     -0.000002  8.460662e+01  9.100048e+01  9.596994e+01
PAKNYGDPFCSTXN     0      1.856042  1.313438e+01  1.946052e+01  2.346548e+01
PAKNEIMPGNFSXN     0     91.632877  7.527132e+01  6.596239e+01  6.058777e+01
PAKNECONOTHRXN_A   0     -0.000002 -8.349738e-07 -4.609774e-07 -8.605964e-07
PAKGGREVGNFSXN     0     -0.000002 -8.349738e-07 -4.609774e-07 -8.605964e-07
                  -1     -0.000002 -8.349738e-07 -4.609774e-07 -8.605964e-07
PAKNECONOTHRXN_D   0     -0.000002 -8.349738e-07 -4.609774e-07 -8.605964e-07
PAKNECONOTHRXN_X   0     -0.000002 -8.349738e-07 -4.609774e-07 -8.605964e-07
PAKNYGDPGAP_       0      6.715406  6.549233e+00  4.337007e+00  2.122444e+00
PAKNEIMPGNFSXN    -1     -0.000002 -7.939116e+01 -7.009483e+01 -6.472465e+01
PAKNYGDPFCSTXN    -1     -0.000002 -1.571084e+00 -1.200007e+01 -1.879798e+01
Total              0    100.204310  9.859930e+01  9.866549e+01  9.862301e+01
Residual           0      0.204310 -1.400698e+00 -1.334509e+00 -1.376992e+00

                                2024        2025        2026        2027  \
Variable          lag
PAKNECONOTHRXN    -1    9.970561e+01  102.418017  104.360990  105.755205
PAKNYGDPFCSTXN     0    2.613470e+01   28.013805   29.383646   30.394521
PAKNEIMPGNFSXN     0    5.752562e+01   55.898697   55.206151   55.151888
PAKNECONOTHRXN_A   0   -8.446033e-07   -0.000002   -0.000003   -0.000003
PAKGGREVGNFSXN     0   -8.446033e-07   -0.000002   -0.000003   -0.000003
                  -1   -8.446033e-07   -0.000002   -0.000003   -0.000003
PAKNECONOTHRXN_D   0   -8.446033e-07   -0.000002   -0.000003   -0.000003
PAKNECONOTHRXN_X   0   -8.446033e-07   -0.000002   -0.000003   -0.000003
PAKNYGDPGAP_       0    4.870008e-01   -0.685120   -1.471705   -1.968265
PAKNEIMPGNFSXN    -1   -6.170561e+01  -60.119200  -59.465019  -59.451158
PAKNYGDPFCSTXN    -1   -2.358831e+01  -27.012130  -29.516228  -31.374669
```

(continues on next page)

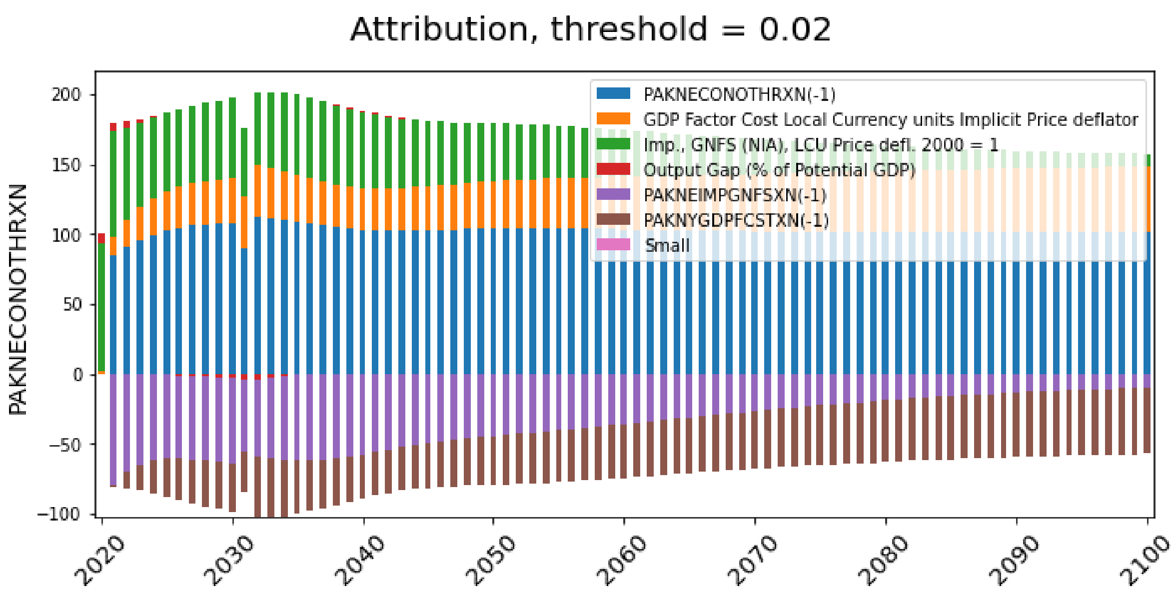
(continued from previous page)

```
Total              0   9.855901e+01   98.514058   98.497819   98.507506
Residual           0  -1.440990e+00   -1.485942   -1.502181   -1.492494

                           2028        2029  ...        2091        2092  \
Variable          lag                        ...
PAKNECONOTHRXN    -1    106.777404  107.540159  ...  101.188729  101.164358
PAKNYGDPFCSTXN     0     31.125292   31.616110  ...   45.867719   46.031293
PAKNEIMPGNFSXN     0     55.558763   56.313837  ...   12.218479   11.818377
PAKNECONOTHRXN_A  0     -0.000003   -0.000004  ...   -0.000002   -0.000002
PAKGGREVGNFSXN     0     -0.000003   -0.000004  ...   -0.000002   -0.000002
                  -1     -0.000003   -0.000004  ...   -0.000002   -0.000002
PAKNECONOTHRXN_D  0     -0.000003   -0.000004  ...   -0.000002   -0.000002
PAKNECONOTHRXN_X  0     -0.000003   -0.000004  ...   -0.000002   -0.000002
PAKNYGDPGAP_       0     -2.257912   -2.395764  ...   -0.252723   -0.255448
PAKNEIMPGNFSXN    -1    -59.906153  -60.719917  ...  -12.792672  -12.367177
PAKNYGDPFCSTXN    -1    -32.760811  -33.775068  ...  -46.331679  -46.492474
Total              0     98.536566   98.579339  ...   99.897842   99.898918
Residual           0     -1.463434   -1.420661  ...   -0.102158   -0.101082

                           2093        2094        2095        2096  \
Variable          lag
PAKNECONOTHRXN    -1    101.141817  101.121018  101.101875  101.084286
PAKNYGDPFCSTXN     0     46.188267   46.338833   46.483184   46.621509
PAKNEIMPGNFSXN     0     11.434709   11.066966   10.714628   10.377171
PAKNECONOTHRXN_A  0     -0.000002   -0.000002   -0.000003   -0.000002
PAKGGREVGNFSXN     0     -0.000002   -0.000002   -0.000003   -0.000002
                  -1     -0.000002   -0.000002   -0.000003   -0.000002
PAKNECONOTHRXN_D  0     -0.000002   -0.000002   -0.000003   -0.000002
PAKNECONOTHRXN_X  0     -0.000002   -0.000002   -0.000003   -0.000002
PAKNYGDPGAP_       0     -0.258416   -0.261598   -0.264965   -0.268486
PAKNEIMPGNFSXN    -1    -11.959268  -11.568397  -11.194007  -10.835530
PAKNYGDPFCSTXN    -1    -46.647151  -46.795882  -46.938831  -47.076157
Total              0     99.899946   99.900928   99.901872   99.902782
Residual           0     -0.100054   -0.099072   -0.098128   -0.097218

                           2097        2098        2099        2100
Variable          lag
PAKNECONOTHRXN    -1    101.068162  101.053408  101.039937  101.027653
PAKNYGDPFCSTXN     0     46.753995   46.880832   47.002206   47.118299
PAKNEIMPGNFSXN     0     10.054064    9.744779    9.448788    9.165568
PAKNECONOTHRXN_A  0     -0.000002   -0.000002   -0.000002   -0.000002
PAKGGREVGNFSXN     0     -0.000002   -0.000002   -0.000002   -0.000002
                  -1     -0.000002   -0.000002   -0.000002   -0.000002
PAKNECONOTHRXN_D  0     -0.000002   -0.000002   -0.000002   -0.000002
PAKNECONOTHRXN_X  0     -0.000002   -0.000002   -0.000002   -0.000002
PAKNYGDPGAP_       0     -0.272134   -0.275885   -0.279718   -0.283601
PAKNEIMPGNFSXN    -1    -10.492400  -10.164046   -9.849902   -9.549410
PAKNYGDPFCSTXN    -1    -47.208018  -47.334571  -47.455971  -47.572369
Total              0     99.903658   99.904507   99.905328   99.906128
Residual           0     -0.096342   -0.095493   -0.094672   -0.093872

[13 rows x 81 columns]
```

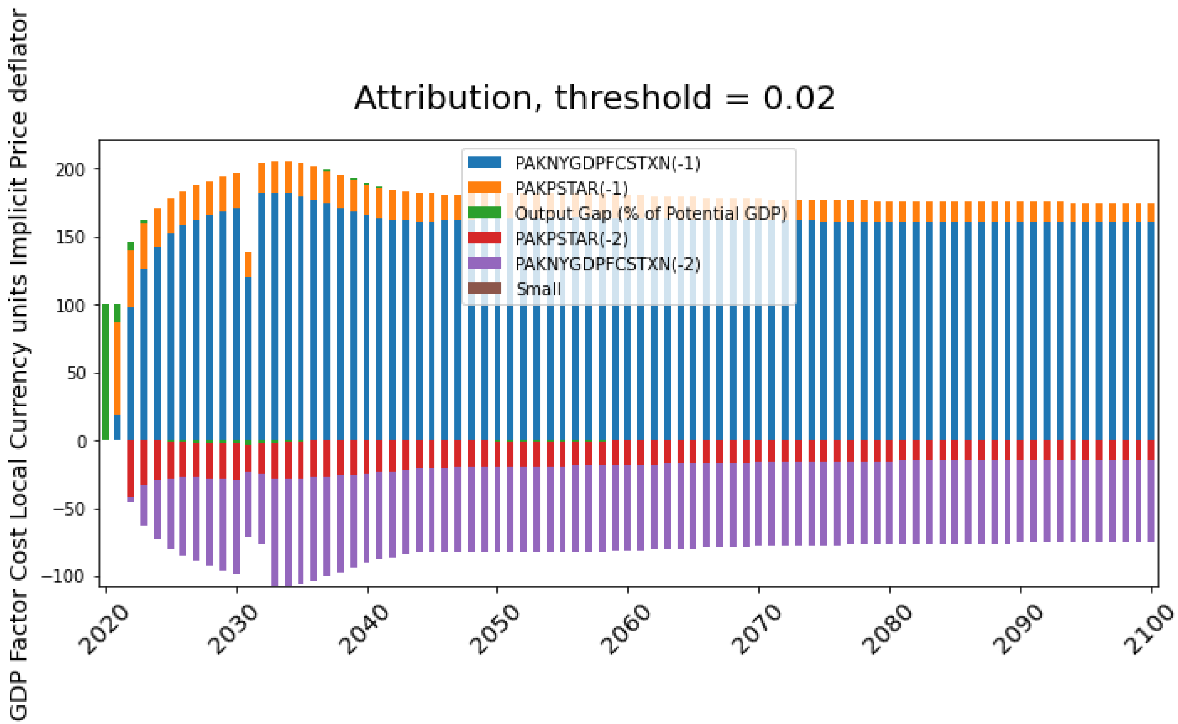
```python
fig=mpak.dekomp_plot('PAKNECONOTHRXN',pct=False,rename=True,threshold=.02); #decomp␣
↪of teh change in the level
```




```
fig=mpak.dekomp_plot('PAKNECONOTHRXN',pct=True,rename=True,threshold=.02);  #␣
↪expressed as a share (the conmtrinutions share)
```




```
fig=mpak.dekomp_plot('PAKNYGDPFCSTXN',pct=True,rename=True,threshold=.02);
```




```
help(mpak.dekomp_plot)
```

```
Help on method dekomp_plot in module modelclass:

dekomp_plot(varnavn, sort=True, pct=True, per='', top=0.9, threshold=0.0, lag=True,
↪ rename=True, nametrans=<function Dekomp_Mixin.<lambda> at 0x000001FED241A160>,␣
↪time_att=False) method of modelclass.model instance
    Returns  a chart with attribution for a variable over the smpl

    Parameters
    ----------
    varnavn : TYPE
        variable name.
    sort : TYPE, optional
        . The default is False.
    pct : TYPE, optional
        display pct contribution . The default is True.
    per : TYPE, optional
        DESCRIPTION. The default is ''.
    threshold : TYPE, optional
        cutoff. The default is 0.0.
    rename : TYPE, optional
        Use descriptions instead of variable names. The default is True.
    time_att : TYPE, optional
        Do time attribution . The default is False.
    lag : TYPE, optional
       separete by lags The default is True.
    top : TYPE, optional
      where to place the title
```

(continues on next page)

(continued from previous page)

```
    Returns
    -------
    a matplotlib figure instance .
```

```
mpak.dekomp_plot_per?
#mpak.dekomp_plot_per('PAKNYGDPFCSTXN',per=(2020,2030))  # gives a waterfall of␣
↪contributions
```

12.5 .totdif() presents the impacts at the model level

The method `.totdif()` returns an instance the totdif class, which provides a number of methods and properties to explore the attribution at the model level.

It works by solving the model numerous time, each time changing one of the right hand side variables and calculating the impact on the dependent variable. By default it uses the values from the `.lastdf` Dataframe as the shock values and the values in `.basedf` as the initial values.

For the purpose of this exercise lets look at a simulation where monetary policy tightens raising the policy interest rate by 100 basis points for 3 years, and then look at the impact on inflation.

For advanced users the RHS variables can be grouped into user defined blocks, which helps identify causal chains.

To begin we reload the model.

```
mpak,baseline = model.modelload('../models/pak.pcim',alfa=0.7,run=1,keep='Business as␣
↪Usual')
```

```
file read:  C:\mflow\modelflow-manual\papers\mfbook\content\models\pak.pcim
```

To determine the mnemonic for the monetary policy, search the dat dictionary for the word policy.

```
mpak['!*policy*'].des
```

```
PAKFMLBLPOLYXN        : Key Policy Interest Rate
PAKFMLBLPOLYXN_A      : Add factor:Key Policy Interest Rate
PAKFMLBLPOLYXN_D      : Fix dummy:Key Policy Interest Rate
PAKFMLBLPOLYXN_FITTED : Fitted  value:Key Policy Interest Rate
PAKFMLBLPOLYXN_X      : Fix value:Key Policy Interest Rate
PAKINTRDDIFF          : Domestic Interest Rate Spread Over Policy Rate
PAKINTRDDIFF_A        : Add factor:Domestic Interest Rate Spread Over Policy Rate
PAKINTRDDIFF_D        : Fix dummy:Domestic Interest Rate Spread Over Policy Rate
PAKINTRDDIFF_FITTED   : Fitted  value:Domestic Interest Rate Spread Over Policy␣
↪Rate
PAKINTRDDIFF_X        : Fix value:Domestic Interest Rate Spread Over Policy Rate
PAKINTREDIFF          : External Interest Rate Spread Over Policy Rate
PAKINTREDIFF_A        : Add factor:External Interest Rate Spread Over Policy Rate
PAKINTREDIFF_D        : Fix dummy:External Interest Rate Spread Over Policy Rate
PAKINTREDIFF_FITTED   : Fitted  value:External Interest Rate Spread Over Policy␣
↪Rate
PAKINTREDIFF_X        : Fix value:External Interest Rate Spread Over Policy Rate
```

The policy variable is PAKFMLBLPOLYXN, do a quick visualization to see its level (percentage points or perhaps ppt / 100?).

```
mpak['PAKFMLBLPOLYXN'].df
```

```
      PAKFMLBLPOLYXN
2016        5.734297
2017        5.754622
2018        5.805750
2019        6.187512
2020        6.670351
2021        7.056445
2022        7.287498
2023        7.376232
2024        7.362666
2025        7.285334
2026        7.171706
2027        7.039316
2028        6.899970
2029        6.763110
2030        6.637192
```

Create a new dataframe for the shocks scenario and increase the Policy rate by 1 percentage point during the period 2025-2027.

```
#MPShockdf=baseline.copy() #Create new df as copy of the baseline

import modelmf



MPShockdf=MPShockdf.upd('<2025 2027> WLDFCRUDE_PETRO + 30')


MPShockdf.loc[2023:2030,'WLDFCRUDE_PETRO']
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Input In [21], in <cell line: 7>()
      1 #MPShockdf=baseline.copy() #Create new df as copy of the baseline
      3 import modelmf
----> 7 MPShockdf=MPShockdf.upd('<2025 2027> WLDFCRUDE_PETRO + 30')
     10 MPShockdf.loc[2023:2030,'WLDFCRUDE_PETRO']

NameError: name 'MPShockdf' is not defined
```

```
#check to see if a variable is endogenous or endogenous
'PAKFMLBLPOLYXN' in mpak.endogene
```

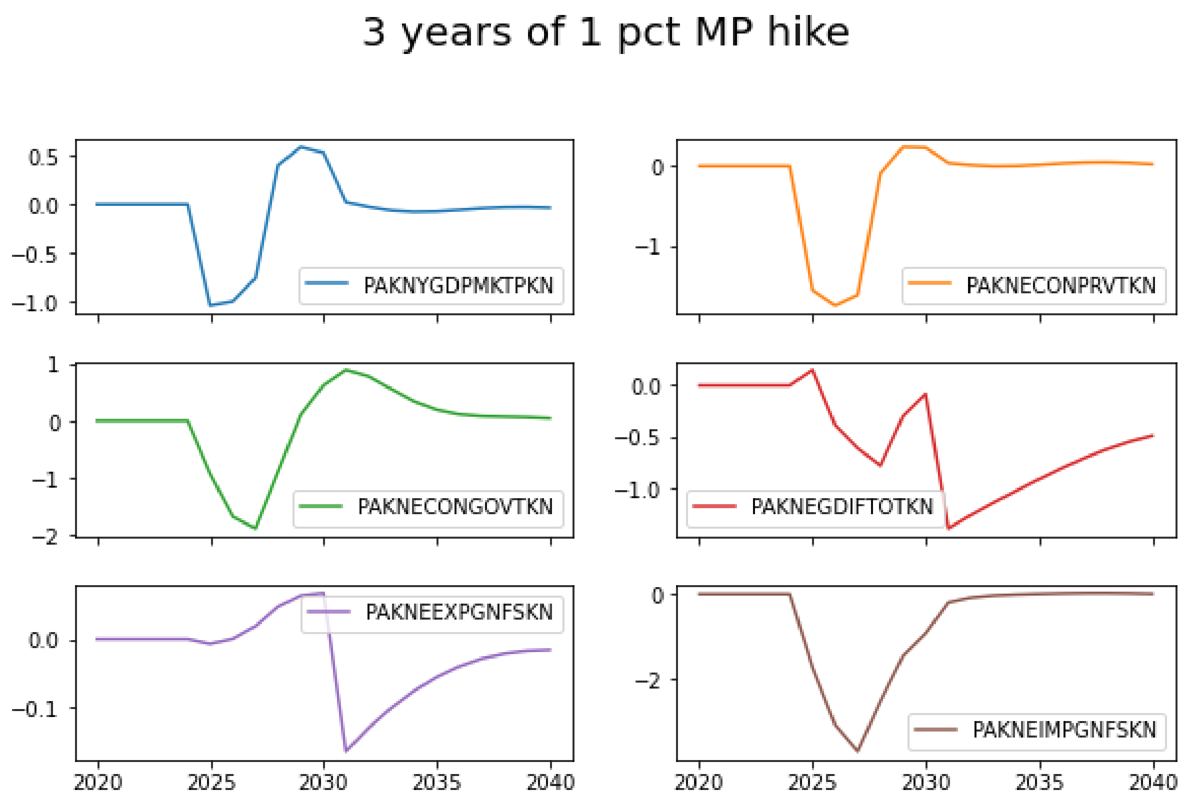
```
True
```

```
mpak['*PAN*'].des
```

```
MPres = mpak(MPShockdf,2020,2100,keep='Increase Policy rate 1 ppt 2025-27') #␣
↪simulates the model
```

```
with mpak.set_smpl(2020,2040):
    mpak['PAKNYGDPMKTPKN PAKNECONPRVTKN PAKNECONGOVTKN PAKNEGDIFTOTKN PAKNEEXPGNFSKN␣
↪PAKNEIMPGNFSKN'].difpctlevel.mul100.plot(title="3 years of 1 pct MP hike");
```

```
C:\Users\wb268970\.conda\envs\modelflow\lib\site-packages\IPython\core\pylabtools.
↪py:151: UserWarning: This figure was using constrained_layout, but that is␣
↪incompatible with subplots_adjust and/or tight_layout; disabling constrained_
↪layout.
  fig.canvas.print_figure(bytes_io, **kw)
```




```
junk=mpak.exodif()

junk.loc[2023:2035,:]
```

```
      PAKPANUSATLS_X  WLDFCRUDE_PETRO
2023      106.693704              0.0
2024      106.566459              0.0
2025      106.454266             30.0
2026      106.353322             30.0
2027      106.261138             30.0
2028      106.175825              0.0
2029      106.095356              0.0
```

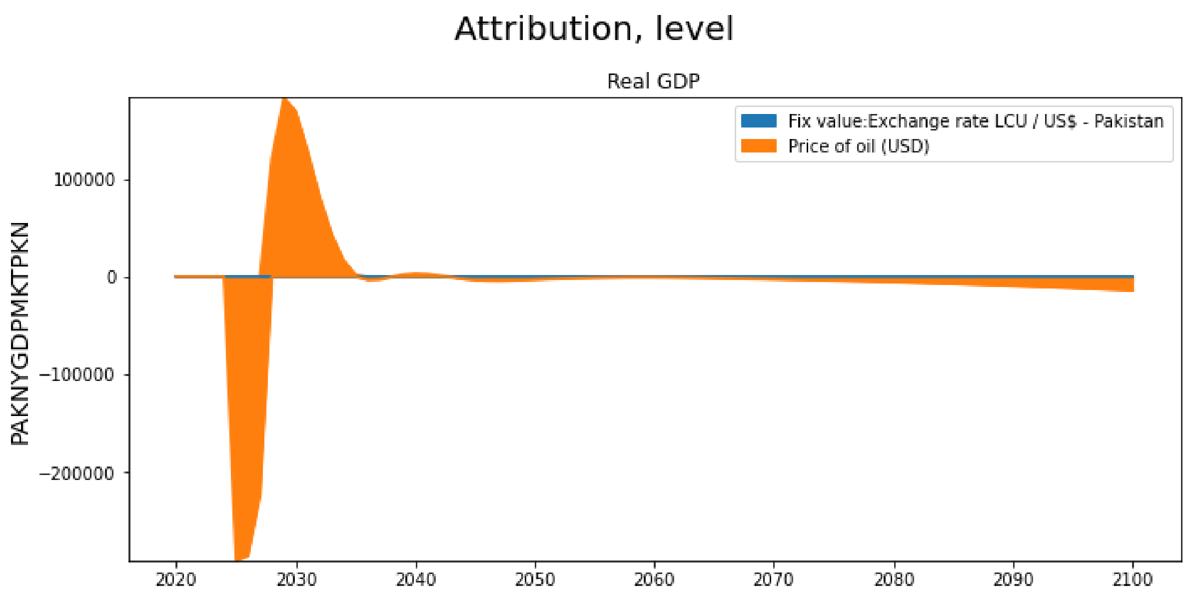
(continues on next page)

(continued from previous page)

```
2030      106.017313          0.0
2031      105.984507          0.0
2032      105.893285          0.0
2033      105.799864          0.0
2034      105.702380          0.0
2035      105.599506          0.0
```

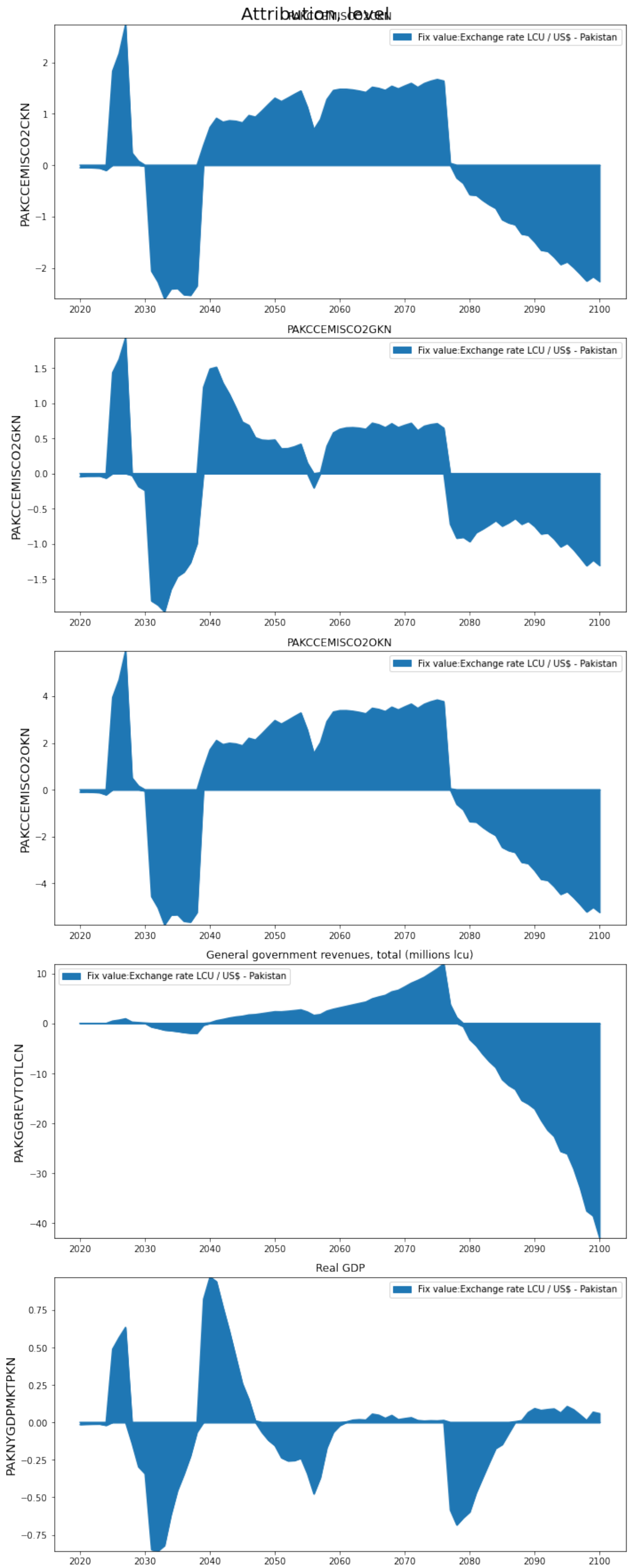
```
totdekomp = mpak.totdif() # Calculate the total derivatives of all equations in the␣
↪model.
showvar = 'PAKNYGDPMKTPKN'
totdekomp.explain_all(showvar,kind='area',stacked=True,top=0.20);
```

```
Total dekomp took       :         1.625 Seconds
```



12.5.1 .explain_all will visualize the results

```
showvar = 'PAKNYGDPMKTPKN PAKCCEMISCO2CKN PAKCCEMISCO2OKN PAKCCEMISCO2GKN␣
↪PAKGGREVTOTLCN'
totdekomp.explain_all(showvar,kind='area',stacked=True,top=0.95);
```


Attribution, level
PAKCCEMISCO2CKN
Fix value:Exchange rate LCU / US$ - Pakistan
PAKCCEMISCO2CKN
PAKCCEMISCO2GKN
Fix value:Exchange rate LCU / US$ - Pakistan
PAKCCEMISCO2GKN
PAKCCEMISCO2OKN
Fix value:Exchange rate LCU / US$ - Pakistan
PAKCCEMISCO2OKN
General government revenues, total (millions lcu)
Fix value:Exchange rate LCU / US$ - Pakistan
PAKGGREVTOTLCN
Real GDP
Fix value:Exchange rate LCU / US$ - Pakistan
PAKNYGDPMKTPKN
2020
2030
2040
2050
2060
2070
2080
2090
2100

12.5.2 Or we can use interactive widgets

This allows the user to select the specific variable of interest and what to display:

Note: If this is read in a manual the widget is not live.

In a notebook the selection widgets are live.

```
display(mpak.get_att_gui(var='PAKGGREVTOTLCN',ysize=7));
```

```
interactive(children=(Dropdown(description='Variable', index=108, options=(
↪'CHNEXR05', 'CHNPCEXN05', 'DEUEXR05…
```

```
None
```

When the results are displayed, they can be filtered, sliced and diced in a number of ways.

12.5.3 More advanced model attribution

For some models (like the EBA bank stress test model) the number of changed exogenous variables can be large. Using a dictionary to contain the experiments allows us to create experiments where all variables for each country are analyzed, or each macro variable for all countries are analyzed.

Also it is possible to use aggregated sums - useful for looking at impact on PD's. Or just the last time period - useful for looking at CET1 ratios.

If there are many experiments, data can be filtered in order to look only at the variables with an impact above a certain threshold.

The is also the possibility to anonymize the row and column names and to randomize the order of rows and/or columns - useful for bank names.

12.5.4 Single equation attribution chart

The results can be visualized in different ways.

```
help(mpak.dekomp_plot)
```

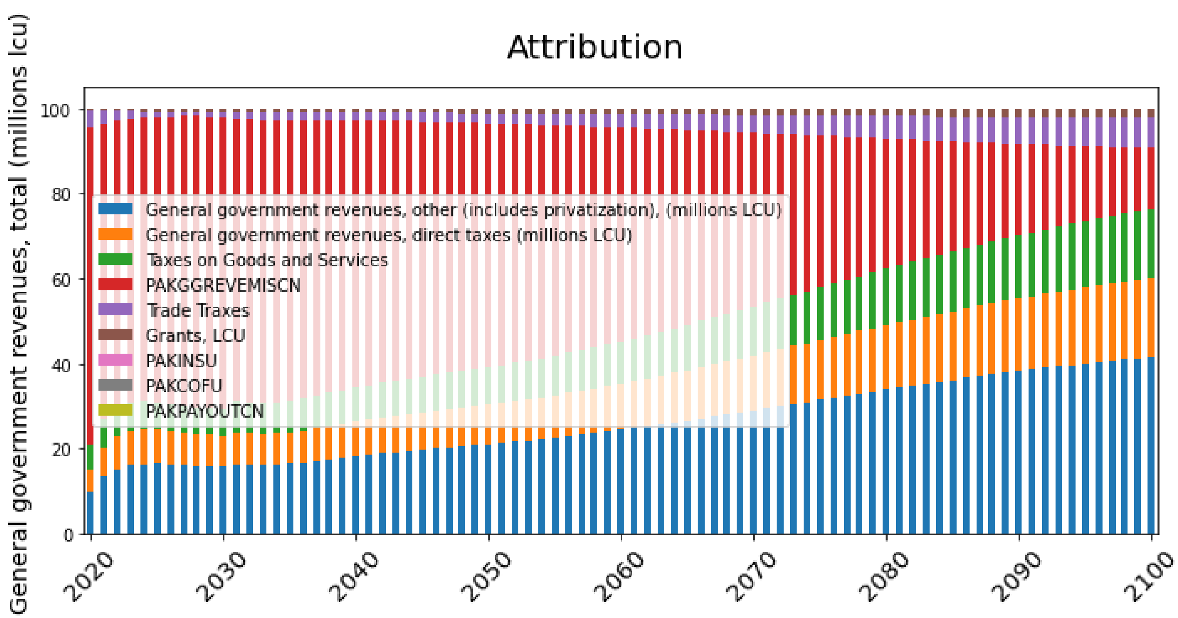
```
Help on method dekomp_plot in module modelclass:

dekomp_plot(varnavn, sort=True, pct=True, per='', top=0.9, threshold=0.0, lag=True,
↪ rename=True, nametrans=<function Dekomp_Mixin.<lambda> at 0x000002093C157280>,␣
↪time_att=False) method of modelclass.model instance
    Returns  a chart with attribution for a variable over the smpl

    Parameters
    ----------
    varnavn : TYPE
        variable name.
    sort : TYPE, optional
        . The default is False.
```

(continues on next page)

(continued from previous page)

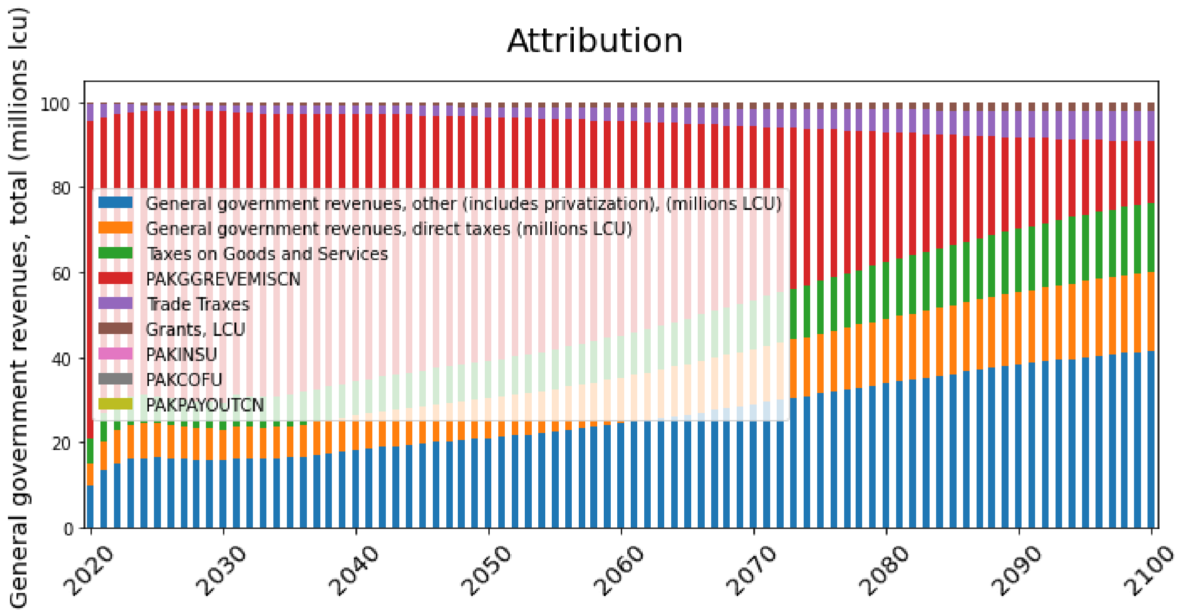
```
    pct : TYPE, optional
        display pct contribution . The default is True.
    per : TYPE, optional
        DESCRIPTION. The default is ''.
    threshold : TYPE, optional
        cutoff. The default is 0.0.
    rename : TYPE, optional
        Use descriptions instead of variable names. The default is True.
    time_att : TYPE, optional
        Do time attribution . The default is False.
    lag : TYPE, optional
       separete by lags The default is True.
    top : TYPE, optional
      where to place the title


    Returns
    -------
    a matplotlib figure instance .
```

```
fig=mpak.dekomp_plot('PAKGGREVTOTLCN',pct=1);
```



12.5.5 Chart in pct of the total

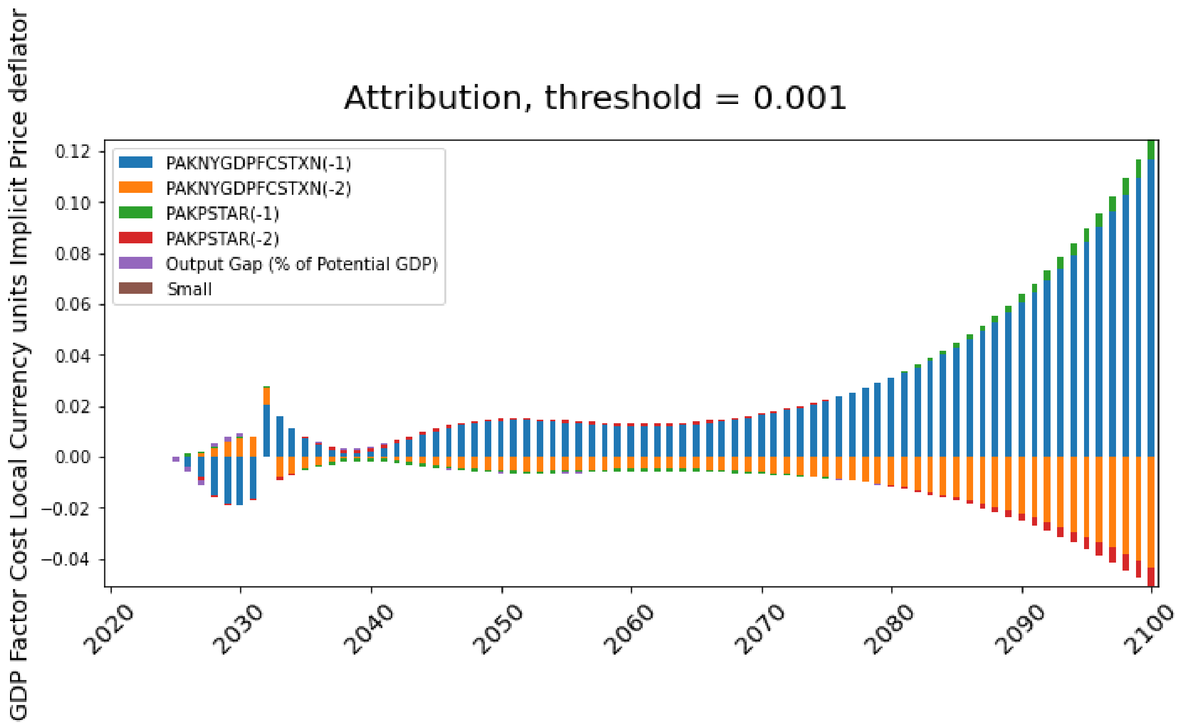
```
mpak.dekomp_plot('PAKGGREVTOTLCN',rename=1);
```



12.5.6 Chart for one year

The attribution for one year can be dispalyed in a waterfall chart.

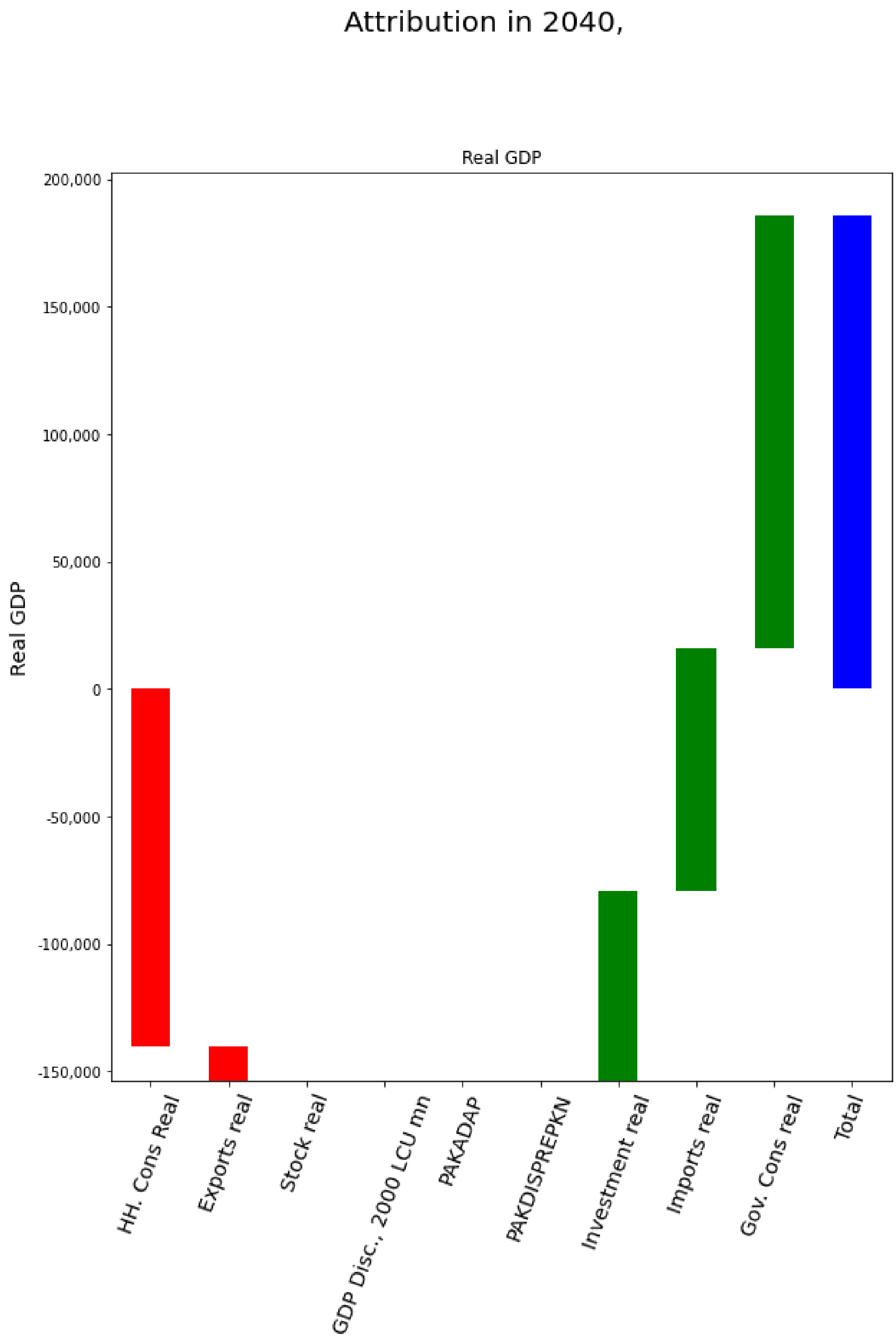
```
#mpak.dekomp_plot_per('PAKNYGDPMKTPXN',per=2040,rename=1,pct=0,ysize=12,threshold␣
↪=20);
mpak.dekomp_plot('PAKNYGDPFCSTXN',per=2040,rename=1,pct=0,threshold =0.001);
```



12.5.7 Sorting of attribution

```
mpak.dekomp_plot_per('PAKNYGDPMKTPKN',per=2040,pct=0,rename=1,sort=1,ysize=12);
```

```
C:\Users\wb268970\.conda\envs\modelflow\lib\site-packages\IPython\core\pylabtools.
↪py:151: UserWarning: This figure was using constrained_layout, but that is␣
↪incompatible with subplots_adjust and/or tight_layout; disabling constrained_
↪layout.
  fig.canvas.print_figure(bytes_io, **kw)
```

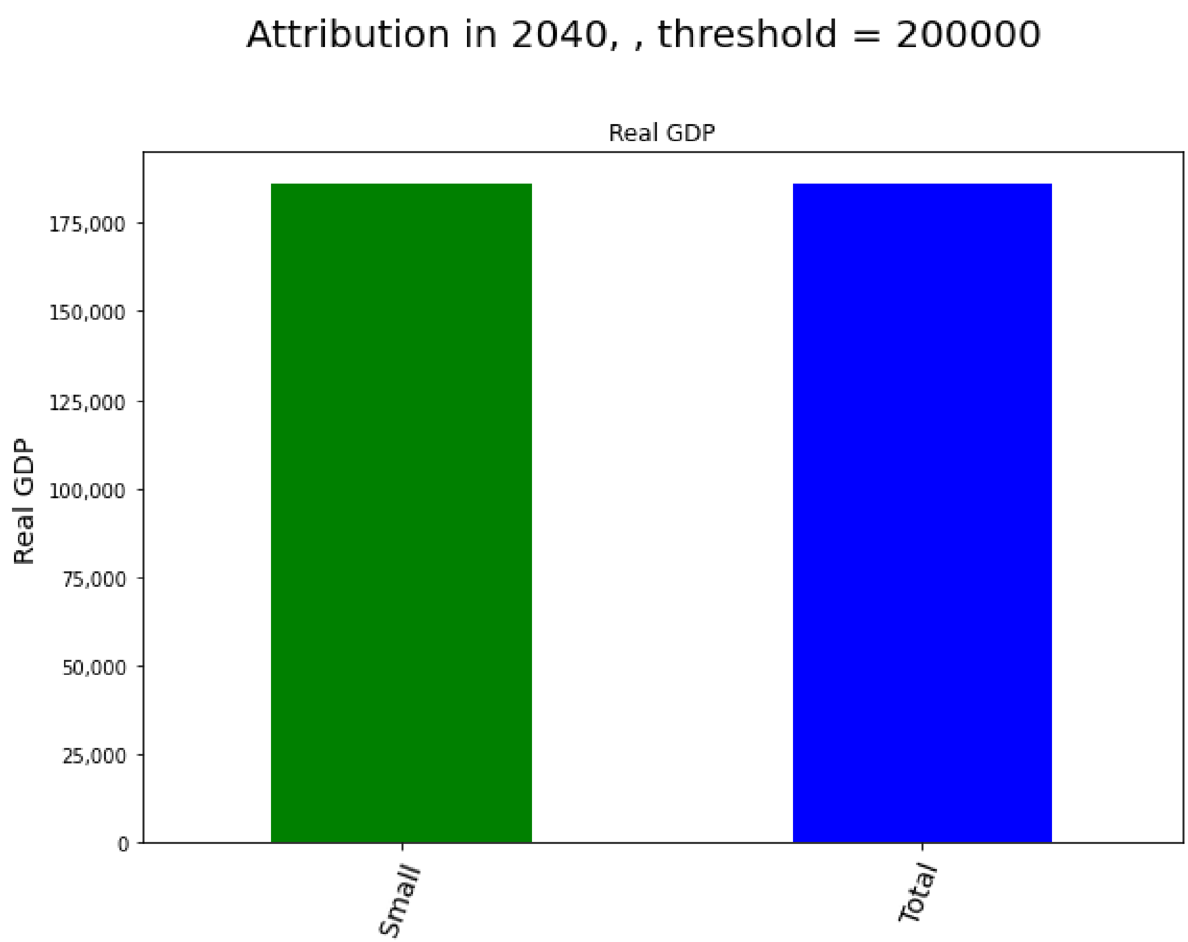

Attribution in 2040,
Real GDP
Real GDP
200,000
150,000
100,000
50,000
0
-50,000
-100,000
-150,000
HH. Cons Real
Exports real
Stock real
GDP Disc., 2000 LCU mn
PAKADAP
PAKDISPREPKN
Investment real
Imports real
Gov. Cons real
Total

12.5.8 Truncate attribution

Some equations have a lot of small contributions. These can be aggregated through the `threshold=<some number>` parameter. Variables for which all contributions are below the threshold will be lumped together in the **small** bin. Like below:

```
mpak.dekomp_plot_per('PAKNYGDPMKTPKN',per=2040,pct=0,rename=1,sort=1,threshold
↪=200000,ysize=7);
```

```
C:\Users\wb268970\.conda\envs\modelflow\lib\site-packages\IPython\core\pylabtools.
↪py:151: UserWarning: This figure was using constrained_layout, but that is
↪incompatible with subplots_adjust and/or tight_layout; disabling constrained_
↪layout.
  fig.canvas.print_figure(bytes_io, **kw)
```



12.5.9 Attribution when comparing time frames

In this case we seek to find out which variables explains the development from year to year. This is done only for the .lastdf dateframe.

```
with mpak.set_smpl(2020,2024):
    mpak['PAKNYGDPMKTPKN'].dekomp(time_att=True)
```

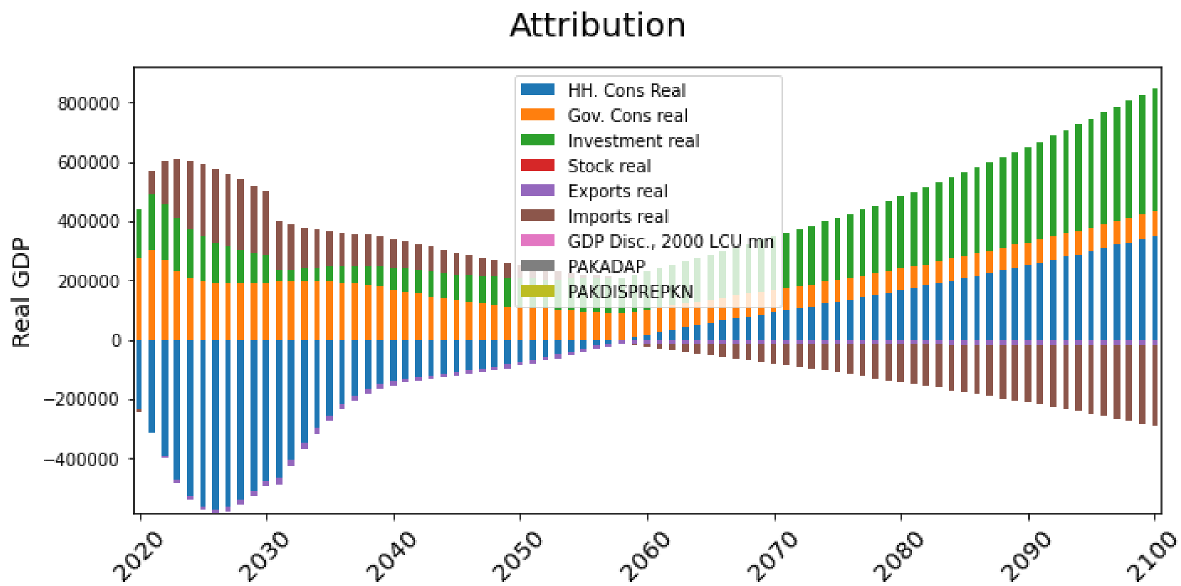
```
Formula        : FRML <IDENT> PAKNYGDPMKTPKN =
↪PAKNECONPRVTKN+PAKNECONGOVTKN+PAKNEGDIFTOTKN+PAKNEGDISTKBKN+PAKNEEXPGNFSKN-
↪PAKNEIMPGNFSKN+PAKNYGDPDISCKN+PAKADAP*PAKDISPREPKN $

                      2020        2021        2022        2023        2024
Variable   lag
t-1        0   25760579.27 26470022.65 26764926.87 26889649.52 27089036.50
t          0   26470022.65 26764926.87 26889649.52 27089036.50 27454422.35
Difference 0     709443.38   294904.22   124722.65   199386.98   365385.85
Percent    0          2.75        1.11        0.47        0.74        1.35

 Contributions to differende for  PAKNYGDPMKTPKN
                         2020       2021       2022       2023       2024
Variable       lag
PAKNECONPRVTKN 0    421908.71  220215.41  108899.48  183642.23  333465.16
PAKNECONGOVTKN 0    344368.08   56568.07   -7860.16   16274.44   61873.41
PAKNEGDIFTOTKN 0    222358.85   63224.20   21093.16    8397.32   10945.62
PAKNEGDISTKBKN 0      9896.74   10138.31   10385.77   10639.25   10898.93
PAKNEEXPGNFSKN 0     95604.61  108517.62  115961.28  120187.58  122631.24
PAKNEIMPGNFSKN 0   -385990.16 -165087.53 -125117.37 -141147.60 -175856.30
PAKNYGDPDISCKN 0      1296.36    1328.02    1360.44    1393.63    1427.65
PAKADAP        0        -0.03      -0.02      -0.01      -0.02      -0.02
PAKDISPREPKN   0        -0.03      -0.02      -0.01      -0.02      -0.02

 Share of contributions to differende for  PAKNYGDPMKTPKN
                          2020        2021        2022        2023        2024
Variable       lag
PAKNECONPRVTKN 0           59%         75%         87%         92%         91%
PAKNEEXPGNFSKN 0           13%         37%         93%         60%         34%
PAKNECONGOVTKN 0           49%         19%         -6%          8%         17%
PAKNEGDIFTOTKN 0           31%         21%         17%          4%          3%
PAKNEGDISTKBKN 0            1%          3%          8%          5%          3%
PAKNYGDPDISCKN 0            0%          0%          1%          1%          0%
PAKADAP        0           -0%         -0%         -0%         -0%         -0%
PAKDISPREPKN   0           -0%         -0%         -0%         -0%         -0%
PAKNEIMPGNFSKN 0          -54%        -56%       -100%        -71%        -48%
Total          0          100%        100%        100%        100%        100%
Residual       0           -0%         -0%         -0%         -0%         -0%

 Contribution to growth rate PAKNYGDPMKTPKN
                          2020        2021        2022        2023        2024
Variable       lag
PAKNECONPRVTKN 0          0.0%        0.0%        0.0%        0.0%        0.0%
PAKNECONGOVTKN 0          0.0%        0.0%       -0.0%        0.0%        0.0%
PAKNEGDIFTOTKN 0          0.0%        0.0%        0.0%        0.0%        0.0%
PAKNEGDISTKBKN 0          0.0%        0.0%        0.0%        0.0%        0.0%
PAKNEEXPGNFSKN 0          0.0%        0.0%        0.0%        0.0%        0.0%
PAKNEIMPGNFSKN 0         -0.0%       -0.0%       -0.0%       -0.0%       -0.0%
PAKNYGDPDISCKN 0          0.0%        0.0%        0.0%        0.0%        0.0%
```

(continues on next page)

(continued from previous page)

```
PAKADAP         0        -0.0%      -0.0%      -0.0%      -0.0%      -0.0%
PAKDISPREPKN    0        -0.0%      -0.0%      -0.0%      -0.0%      -0.0%
```

```
mpak.dekomp_plot('PAKNYGDPMKTPKN',pct=0,rename=1,sort=1,threshold =0,time_att = True);
```



12.5.10 Visualizing attribution in dependency graphs

The logical graph of the model can be used to show the upstream and downstream variable for a specific variable. More on this here When drawing the logical graph for a variable the model attribution will be used to guide the thickness of edges between nodes (variables). This enables a visual impression of which variables drives the impact.

Note: If png == 0 the graph below will be rendered in SVG format. This enables tooltips with additional information when the mouse hovers over an edge or an node.

Unfortunately svg can't be displayed in the manual, so png has to be True for the manual. In a live jupyter notebook set latex=0. This will enable svg format.

```
#mpak['PAKNYGDPMKTPKN PAKNECONPRVTKN'].draw(up=3,down=0,png=latex,filter=20) # For␣
↪book
mpak['PAKNYGDPMKTPKN'].draw(up=3,down=0,png=False,filter=400,svg=True,size=(8,40))#3␣
↪for interactice
```

```
<IPython.core.display.SVG object>
```

```
mpak['PAKNYGDPMKTPKN PAKNECONPRVTKN'].draw(up=1,down=1,png=latex)   # diagram all␣
↪direct dependencies
```

```
<IPython.core.display.SVG object>
```

```
<IPython.core.display.SVG object>
```

12.5.11 The attribution can be filtered and more levels can be displayed.

```
mpak['PAKNYGDPMKTPKN'].draw(up=2,down=1,png=latex,filter=20)
```

```
<IPython.core.display.SVG object>
```

12.5.12 Or it can be used in a dashboard (not avaiable in the offline manual)

```
try:
    mpak.modeldash('PAKNYGDPMKTPKN',jupyter=1,inline=False)
except:
    print('No Dashboard installed')
```

```
No Dash
No Dash, name 'DashInteractiveGraphviz' is not defined
```

```
!pip install
```

Part V

More modelflow features

The preceding chapters have demonstrated how to install `modelflow`, download a World Bank macro model, perform a variety of simulations with the model and report the results using some of the core functionalities of the `modelflow` library. In this section demonstrates some additional functionality that can be helpful.

CHAPTER

THIRTEEN

THE MODEL WIDGET CLASS

The model widget class provides some additional display routines that might be useful.

To access it in addition to initializing you python session with 1`modelflow` one also needs to import the `modelwidget` class, which is installed as part of the `modelflow` library.

```
import pandas as pd
from modelclass import model
import modelwidget as mw  # import the modelwidget class into the python session
model.widescreen()
model.scroll_off()

#load a pre-existing model
mpak,result = model.modelload('../../models/pak.pcim',run=1,silent=1)
```

```
<IPython.core.display.HTML object>
```

```
Open file from URL:  https://raw.githubusercontent.com/IbHansen/modelflow-manual/
↪main/model_repo/pak.pcim
```

13.1 The .html_widget_df() method

The `.html_widget_df()` method takes a dataframe and displays it as a nicely formatted html table. Below we create a `DataFarme` comprised of the growth rates of all the national expenditure variables in the `mpak` model.

```
dispdf=mpak['PAKNE*KN'].pct.mul100.df #return a dataframe with all ofthe variables␣
↪matching
                         #the wildcard specification.  In this case national␣
↪expenditure variables.

mw.htmlwidget_df(mpak,dispdf.loc[2025:2030]).show
```

```
HTML(value='<style type="text/css">\n#T_0ff04 thead tr:nth-child(1) th {\n ␣
↪position: sticky;\n  background-co…
```

```
#mw.htmlwidget_fig(mpak,dispdf).show

help(mw)
```

```
Help on module modelwidget:

NAME
    modelwidget - Created on Mon Aug  9 14:46:11 2021

DESCRIPTION
    To define Jupyter widgets  show variables.
    @author: Ib

CLASSES
    builtins.object
        basewidget
        htmlwidget_df
        htmlwidget_fig
        htmlwidget_label
        sheetwidget
        slidewidget
        sumslidewidget
        tabwidget
        updatewidget
        visshow

    class basewidget(builtins.object)
     |  basewidget(datachildren: list = <factory>) -> None
     |
     |  basis for widget updating in jupyter
     |
     |  Methods defined here:
     |
     |  __eq__(self, other)
     |
     |  __init__(self, datachildren: list = <factory>) -> None
     |
     |  __repr__(self)
     |
     |  update_df(self, df, current_per)
     |      will update container widgets
     |
     |  ----------------------------------------------------------------------
     |  Data descriptors defined here:
     |
     |  __dict__
     |      dictionary for instance variables (if defined)
     |
     |  __weakref__
     |      list of weak references to the object (if defined)
     |
     |  ----------------------------------------------------------------------
     |  Data and other attributes defined here:
     |
     |  __annotations__ = {'datachildren': <class 'list'>}
     |
     |  __dataclass_fields__ = {'datachildren': Field(name='datachildren',type...
     |
     |  __dataclass_params__ = _DataclassParams(init=True,repr=True,eq=True,or...
     |
     |  __hash__ = None
```

(continues on next page)

(continued from previous page)

```
    class htmlwidget_df(builtins.object)
     |  htmlwidget_df(mmodel: <built-in function any>, df_var: <built-in function␣
↪any> = Empty DataFrame
     |  Columns: []
     |  Index: [], trans: <built-in function any> = <function htmlwidget_df.
↪<lambda> at 0x000001BFEF6A1E50>, transpose: bool = False, expname: str = '',␣
↪percent: bool = False) -> None
     |
     |  class displays a dataframe in a html widget
     |
     |  Methods defined here:
     |
     |  __eq__(self, other)
     |
     |  __init__(self, mmodel: <built-in function any>, df_var: <built-in function␣
↪any> = Empty DataFrame
     |  Columns: []
     |  Index: [], trans: <built-in function any> = <function htmlwidget_df.
↪<lambda> at 0x000001BFEF6A1E50>, transpose: bool = False, expname: str = '',␣
↪percent: bool = False) -> None
     |
     |  __post_init__(self)
     |
     |  __repr__(self)
     |
     |  trans lambda x
     |
     |  ----------------------------------------------------------------------
     |  Readonly properties defined here:
     |
     |  show
     |
     |  ----------------------------------------------------------------------
     |  Data descriptors defined here:
     |
     |  __dict__
     |      dictionary for instance variables (if defined)
     |
     |  __weakref__
     |      list of weak references to the object (if defined)
     |
     |  ----------------------------------------------------------------------
     |  Data and other attributes defined here:
     |
     |  __annotations__ = {'df_var': <built-in function any>, 'expname': <clas...
     |
     |  __dataclass_fields__ = {'df_var': Field(name='df_var',type=<built-in f...
     |
     |  __dataclass_params__ = _DataclassParams(init=True,repr=True,eq=True,or...
     |
     |  __hash__ = None
     |
     |  df_var = Empty DataFrame
     |  Columns: []
     |  Index: []
```

(continues on next page)

(continued from previous page)

```
     |
     |  expname = ''
     |
     |  percent = False
     |
     |  transpose = False

    class htmlwidget_fig(builtins.object)
     |  htmlwidget_fig(figs: <built-in function any>, expname: str = '', format:␣
↪str = 'svg') -> None
     |
     |  class displays a dataframe in a html widget
     |
     |  Methods defined here:
     |
     |  __eq__(self, other)
     |
     |  __init__(self, figs: <built-in function any>, expname: str = '', format:␣
↪str = 'svg') -> None
     |
     |  __post_init__(self)
     |
     |  __repr__(self)
     |
     |  ----------------------------------------------------------------------
     |  Data descriptors defined here:
     |
     |  __dict__
     |      dictionary for instance variables (if defined)
     |
     |  __weakref__
     |      list of weak references to the object (if defined)
     |
     |  ----------------------------------------------------------------------
     |  Data and other attributes defined here:
     |
     |  __annotations__ = {'expname': <class 'str'>, 'figs': <built-in functio...
     |
     |  __dataclass_fields__ = {'expname': Field(name='expname',type=<class 's...
     |
     |  __dataclass_params__ = _DataclassParams(init=True,repr=True,eq=True,or...
     |
     |  __hash__ = None
     |
     |  expname = ''
     |
     |  format = 'svg'

    class htmlwidget_label(builtins.object)
     |  htmlwidget_label(expname: str = '', format: str = 'svg') -> None
     |
     |  class displays a dataframe in a html widget
     |
     |  Methods defined here:
     |
     |  __eq__(self, other)
```

(continues on next page)

(continued from previous page)

```
     |
     |  __init__(self, expname: str = '', format: str = 'svg') -> None
     |
     |  __post_init__(self)
     |
     |  __repr__(self)
     |
     |  ----------------------------------------------------------------------
     |  Data descriptors defined here:
     |
     |  __dict__
     |      dictionary for instance variables (if defined)
     |
     |  __weakref__
     |      list of weak references to the object (if defined)
     |
     |  ----------------------------------------------------------------------
     |  Data and other attributes defined here:
     |
     |  __annotations__ = {'expname': <class 'str'>, 'format': <class 'str'>}
     |
     |  __dataclass_fields__ = {'expname': Field(name='expname',type=<class 's...
     |
     |  __dataclass_params__ = _DataclassParams(init=True,repr=True,eq=True,or...
     |
     |  __hash__ = None
     |
     |  expname = ''
     |
     |  format = 'svg'

    class sheetwidget(builtins.object)
     |  sheetwidget(df_var: <built-in function any> = Empty DataFrame
     |  Columns: []
     |  Index: [], trans: <built-in function any> = <function sheetwidget.<lambda>␣
↪at 0x000001BFEF70D9D0>, transpose: bool = False, expname: str = 'Carbon tax rate,
↪ US$ per tonn ') -> None
     |
     |  class defining a widget which updates from a sheet
     |
     |  Methods defined here:
     |
     |  __eq__(self, other)
     |
     |  __init__(self, df_var: <built-in function any> = Empty DataFrame
     |  Columns: []
     |  Index: [], trans: <built-in function any> = <function sheetwidget.<lambda>␣
↪at 0x000001BFEF70D9D0>, transpose: bool = False, expname: str = 'Carbon tax rate,
↪ US$ per tonn ') -> None
     |
     |  __post_init__(self)
     |
     |  __repr__(self)
     |
     |  reset(self, g)
     |
```

(continues on next page)

(continued from previous page)

```
     |  trans lambda x
     |
     |  update_df(self, df, current_per=None)
     |
     |  ----------------------------------------------------------------------
     |  Data descriptors defined here:
     |
     |  __dict__
     |      dictionary for instance variables (if defined)
     |
     |  __weakref__
     |      list of weak references to the object (if defined)
     |
     |  ----------------------------------------------------------------------
     |  Data and other attributes defined here:
     |
     |  __annotations__ = {'df_var': <built-in function any>, 'expname': <clas...
     |
     |  __dataclass_fields__ = {'df_var': Field(name='df_var',type=<built-in f...
     |
     |  __dataclass_params__ = _DataclassParams(init=True,repr=True,eq=True,or...
     |
     |  __hash__ = None
     |
     |  df_var = Empty DataFrame
     |  Columns: []
     |  Index: []
     |
     |  expname = 'Carbon tax rate, US$ per tonn '
     |
     |  transpose = False

    class slidewidget(builtins.object)
     |  slidewidget(slidedef: dict, altname: str = 'Alternative', basename: str =
↪'Baseline', expname: str = 'Carbon tax rate, US$ per tonn ') -> None
     |
     |  class defefining a widget with lines of slides
     |
     |  Methods defined here:
     |
     |  __eq__(self, other)
     |
     |  __init__(self, slidedef: dict, altname: str = 'Alternative', basename: str␣
↪= 'Baseline', expname: str = 'Carbon tax rate, US$ per tonn ') -> None
     |
     |  __post_init__(self)
     |
     |  __repr__(self)
     |
     |  reset(self, g)
     |
     |  set_slide_value(self, g)
     |      updates the new values to the self.current_vlues
     |      will be used in update_df
     |
     |  update_df(self, df, current_per)
```

(continues on next page)

(continued from previous page)

```
     |      updates a dataframe with the values from the widget
     |
     |  ----------------------------------------------------------------------
     |  Data descriptors defined here:
     |
     |  __dict__
     |      dictionary for instance variables (if defined)
     |
     |  __weakref__
     |      list of weak references to the object (if defined)
     |
     |  ----------------------------------------------------------------------
     |  Data and other attributes defined here:
     |
     |  __annotations__ = {'altname': <class 'str'>, 'basename': <class 'str'>...
     |
     |  __dataclass_fields__ = {'altname': Field(name='altname',type=<class 's...
     |
     |  __dataclass_params__ = _DataclassParams(init=True,repr=True,eq=True,or...
     |
     |  __hash__ = None
     |
     |  altname = 'Alternative'
     |
     |  basename = 'Baseline'
     |
     |  expname = 'Carbon tax rate, US$ per tonn '

    class sumslidewidget(builtins.object)
     |  sumslidewidget(slidedef: dict, maxsum: <built-in function any> = None,␣
↪altname: str = 'Alternative', basename: str = 'Baseline', expname: str = 'Carbon␣
↪tax rate, US$ per tonn ') -> None
     |
     |  class defefining a widget with lines of slides
     |
     |  Methods defined here:
     |
     |  __eq__(self, other)
     |
     |  __init__(self, slidedef: dict, maxsum: <built-in function any> = None,␣
↪altname: str = 'Alternative', basename: str = 'Baseline', expname: str = 'Carbon␣
↪tax rate, US$ per tonn ') -> None
     |
     |  __post_init__(self)
     |
     |  __repr__(self)
     |
     |  reset(self, g)
     |
     |  set_slide_value(self, g)
     |      updates the new values to the self.current_vlues
     |      will be used in update_df
     |
     |  update_df(self, df, current_per)
     |      updates a dataframe with the values from the widget
     |
```

(continues on next page)

(continued from previous page)

```
     |  ----------------------------------------------------------------------
     |  Data descriptors defined here:
     |
     |  __dict__
     |      dictionary for instance variables (if defined)
     |
     |  __weakref__
     |      list of weak references to the object (if defined)
     |
     |  ----------------------------------------------------------------------
     |  Data and other attributes defined here:
     |
     |  __annotations__ = {'altname': <class 'str'>, 'basename': <class 'str'>...
     |
     |  __dataclass_fields__ = {'altname': Field(name='altname',type=<class 's...
     |
     |  __dataclass_params__ = _DataclassParams(init=True,repr=True,eq=True,or...
     |
     |  __hash__ = None
     |
     |  altname = 'Alternative'
     |
     |  basename = 'Baseline'
     |
     |  expname = 'Carbon tax rate, US$ per tonn '
     |
     |  maxsum = None

    class tabwidget(builtins.object)
     |  tabwidget(tabdefdict: dict, tab: bool = True, selected_index: <built-in␣
↪function any> = None) -> None
     |
     |  A widget to create tab or acordium contaners
     |
     |  Methods defined here:
     |
     |  __eq__(self, other)
     |
     |  __init__(self, tabdefdict: dict, tab: bool = True, selected_index: <built-
↪in function any> = None) -> None
     |
     |  __post_init__(self)
     |
     |  __repr__(self)
     |
     |  reset(self, g)
     |      will reset  container widgets
     |
     |  update_df(self, df, current_per)
     |      will update container widgets
     |
     |  ----------------------------------------------------------------------
     |  Data descriptors defined here:
     |
     |  __dict__
     |      dictionary for instance variables (if defined)
```

(continues on next page)

(continued from previous page)

```
     |
     |  __weakref__
     |      list of weak references to the object (if defined)
     |
     |  ----------------------------------------------------------------------
     |  Data and other attributes defined here:
     |
     |  __annotations__ = {'selected_index': <built-in function any>, 'tab': <...
     |
     |  __dataclass_fields__ = {'selected_index': Field(name='selected_index',...
     |
     |  __dataclass_params__ = _DataclassParams(init=True,repr=True,eq=True,or...
     |
     |  __hash__ = None
     |
     |  selected_index = None
     |
     |  tab = True

    class updatewidget(builtins.object)
     |  updatewidget(mmodel: <built-in function any>, a_datawidget: <built-in␣
↪function any>, basename: str = 'Business as usual', keeppat: str = '*', varpat:␣
↪str = '*', showvarpat: bool = True, exodif: <built-in function any> = Empty␣
↪DataFrame
     |  Columns: []
     |  Index: [], lwrun: bool = True, lwupdate: bool = False, lwreset: bool =␣
↪True, lwsetbas: bool = True, lwshow: bool = True, outputwidget: str = 'jupviz',␣
↪prefix_dict: dict = <factory>, display_first: <built-in function any> = None,␣
↪vline: list = <factory>, relativ_start: int = 0, short: bool = False, legend:␣
↪bool = False) -> None
     |
     |  class to input and run a model
     |
     |  Methods defined here:
     |
     |  __eq__(self, other)
     |
     |  __init__(self, mmodel: <built-in function any>, a_datawidget: <built-in␣
↪function any>, basename: str = 'Business as usual', keeppat: str = '*', varpat:␣
↪str = '*', showvarpat: bool = True, exodif: <built-in function any> = Empty␣
↪DataFrame
     |  Columns: []
     |  Index: [], lwrun: bool = True, lwupdate: bool = False, lwreset: bool =␣
↪True, lwsetbas: bool = True, lwshow: bool = True, outputwidget: str = 'jupviz',␣
↪prefix_dict: dict = <factory>, display_first: <built-in function any> = None,␣
↪vline: list = <factory>, relativ_start: int = 0, short: bool = False, legend:␣
↪bool = False) -> None
     |
     |  __post_init__(self)
     |
     |  __repr__(self)
     |
     |  reset(self, g)
     |
     |  run(self, g)
     |
```

(continues on next page)

(continued from previous page)

```
 |  setbasis(self, g)
 |
 |  show(self, g=None)
 |
 |  update(self, g)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  __annotations__ = {'a_datawidget': <built-in function any>, 'basename'...
 |
 |  __dataclass_fields__ = {'a_datawidget': Field(name='a_datawidget',type...
 |
 |  __dataclass_params__ = _DataclassParams(init=True,repr=True,eq=True,or...
 |
 |  __hash__ = None
 |
 |  basename = 'Business as usual'
 |
 |  display_first = None
 |
 |  exodif = Empty DataFrame
 |  Columns: []
 |  Index: []
 |
 |  keeppat = '*'
 |
 |  legend = False
 |
 |  lwreset = True
 |
 |  lwrun = True
 |
 |  lwsetbas = True
 |
 |  lwshow = True
 |
 |  lwupdate = False
 |
 |  outputwidget = 'jupviz'
 |
 |  relativ_start = 0
 |
 |  short = False
 |
 |  showvarpat = True
 |
```

(continues on next page)

(continued from previous page)

```
     |  varpat = '*'

    class visshow(builtins.object)
     |  visshow(mmodel: <built-in function any>, varpat: str = '*', showvarpat:␣
↪bool = True, show_on: bool = True) -> None
     |
     |  visshow(mmodel: <built-in function any>, varpat: str = '*', showvarpat:␣
↪bool = True, show_on: bool = True)
     |
     |  Methods defined here:
     |
     |  __eq__(self, other)
     |
     |  __init__(self, mmodel: <built-in function any>, varpat: str = '*',␣
↪showvarpat: bool = True, show_on: bool = True) -> None
     |
     |  __post_init__(self)
     |
     |  __repr__(self)
     |      Return repr(self).
     |
     |  ----------------------------------------------------------------------
     |  Readonly properties defined here:
     |
     |  show
     |
     |  ----------------------------------------------------------------------
     |  Data descriptors defined here:
     |
     |  __dict__
     |      dictionary for instance variables (if defined)
     |
     |  __weakref__
     |      list of weak references to the object (if defined)
     |
     |  ----------------------------------------------------------------------
     |  Data and other attributes defined here:
     |
     |  __annotations__ = {'mmodel': <built-in function any>, 'show_on': <clas...
     |
     |  __dataclass_fields__ = {'mmodel': Field(name='mmodel',type=<built-in f...
     |
     |  __dataclass_params__ = _DataclassParams(init=True,repr=True,eq=True,or...
     |
     |  __hash__ = None
     |
     |  show_on = True
     |
     |  showvarpat = True
     |
     |  varpat = '*'

FUNCTIONS
    fig_to_image(figs, format='svg')

FILE
```

(continues on next page)

(continued from previous page)

```
   c:\users\wb268970\.conda\envs\modelflow\lib\site-packages\modelflow-1.0.8-py3.
↪9.egg\modelwidget.py
```

Note: Dataframes with strings and zip

Pandas dataframes are very versatile. Here the dataframe is filled not with scalars but with strings.

Also the zip function is used to combine lists. More here

13.1.1 List of fixed variables

```
mpak.split_calc_add_factor
```

```
True
```

13.1.2 Show the equations for the .calc_add_factor_model

Here only 3 are displayed. Delete the [:3] and all equations will be displayed.

```
mpak.calc_add_factor_model.equations.split('$')[:3]
```

```
['FRML <CALC> PAKBMFSTOTHRCD_A = -PAKBMFSTOTHRCD/PAKNYGDPMKTPCD+ ((-0.
↪0106244247103773)) ',
 ' FRML <CALC> PAKBMFSTREMTCD_A = -PAKBMFSTREMTCD/PAKNYGDPMKTPCD+ ((5.
↪83179728399106E-05)) ',
 ' FRML <CALC> PAKBMGSRGNFSCD_A = -100*PAKBMGSRGNFSCD/PAKBMGSRGNFSCD(-1)+ (((100 *␣
↪( (PAKNEIMPGNFSCD) / (PAKNEIMPGNFSCD(-1)) -1)) +0.16331992292838*DUMH)) +100']
```

Part VI

References

BIBLIOGRAPHY

[1] Doug Addison. *The World Bank revised minimum standard model (RMSM) : concepts and issues*. Number WPS231 in Policy Research Working Papers. World Bank, Washington DC., 1989. URL: https://documents.worldbank.org/en/publication/documents-reports/documentdetail/997721468765042532/the-world-bank-revised-minimum-standard-model-rmsm-concepts-and-issues.

[2] Olivier Blanchard. On the future of Macroeconomic models. *Oxford Review of Economic Policy*, 34(1-2):43–54, 2018. URL: https://academic.oup.com/oxrep/article/34/1-2/43/4781808, doi:https://doi.org/10.1093/oxrep/grx045.

[3] Andrew Burns, Benoit Campagne, Charl Jooste, David Stephan, and Thi Thanh Bui. *The World Bank Macro-Fiscal Model Technical Description*. Number 8965 in Policy Research Working Papers. World Bank, Washington DC., 2019. URL: https://openknowledge.worldbank.org/handle/10986/32217.

[4] Andrew Burns, Charl Jooste, and Gregor Schwerhoff. *Climate Modeling for Macroeconomic Policy : A Case Study for Pakistan*. Number 9780 in Policy Research Working Papers. World Bank, Washington, DC, 2021. URL: https://openknowledge.worldbank.org/bitstream/handle/10986/36307/Climate-Modeling-for-Macroeconomic-Policy-A-Case-Study-for-Pakistan.pdf?sequence=1&isAllowed=y.

[5] Andrew Burns, Charl Jooste, and Gregor Schwerhoff. *Macroeconomic Modeling of Managing Hurricane Damage in the Caribbean: The Case of Jamaica*. Volume 9505 of Policy Research Working Paper. World Bank, Washington DC., 2021. URL: https://documents1.worldbank.org/curated/en/593351609776234361/pdf/Macroeconomic-Modeling-of-Managing-Hurricane-Damage-in-the-Caribbean-The-Case-of-Jamaica.pdf.

[6] Hollis Chenery. *Studies in Development Planning.* Harvard University Press,, Cambridge, MA., 1971.

[7] M. R. Wickens and T. S. Breusch. Dynamic Specification, the Long-Run and the Estimation of Transformed Regression Models. *The Economic Journal*, 98:189–205, April 1988.