The World Bank's MFMod Framework in Python with Modelflow

Andrew Burns and Ib Hansen

Apr 25, 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:

$$y_t^1 = f^1(y_{t+u}^1 ..., y_{t+u}^n ..., y_t^2 ..., y_t^n ... y_{t-r}^1 ..., y_{t-r}^n, x_t^1 ... x_t^k, ... x_{t-s}^1 ..., x_{t-s}^k) \tag{1.1}$$

$$y_t^2 = f^2(y_{t+u}^1 ..., y_{t+u}^n ..., y_t^1 ..., y_t^n ... y_{t-r}^1 ..., y_{t-r}^n, x_t^1 ... x_t^k, ... x_{t-s}^1 ..., x_{t-s}^k) \tag{1.2}$$

$$\vdots \tag{1.3}$$

$$y_t^n = f^n(y_{t+u}^1 ..., y_{t+u}^n ..., y_t^1 ..., y_t^{n-1} ... y_{t-r}^1 ..., y_{t-r}^n, x_t^1 ... x_t^r, x ..._{t-s}^1 ..., x_{t-s}^k) \tag{1.4}$$

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:

$$Y_t = C_t + I_t + G + t + (X_t - M_t) \tag{1.5}$$
$$C_t = c_t(C_{t-1}, C_{t-2}, I_t, G_t, X_t, M_t, P_t) \tag{1.6}$$
$$I_t = c_t(I_{t-1}, I_{t-2}, C_t, G_t, X_t, M_t, P_t) \tag{1.7}$$
$$G_t = c_t(G_{t-1}, G_{t-2}, C_t, I_t, X_t, M_t, P_t) \tag{1.8}$$
$$X_t = c_t(X_{t-1}, X_{t-2}, C_t, I_t, G_t, M_t, P_t, P_t^f) \tag{1.9}$$
$$M_t = c_t(M_{t-1}, M_{t-2}, C_t, I_t, G_t, X_t, P_t, P_t^f) \tag{1.10}$$

and where P_t, P_t^f (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.

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.

2.1 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.

2.1.1 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.

2.2 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

2.2.1 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.

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

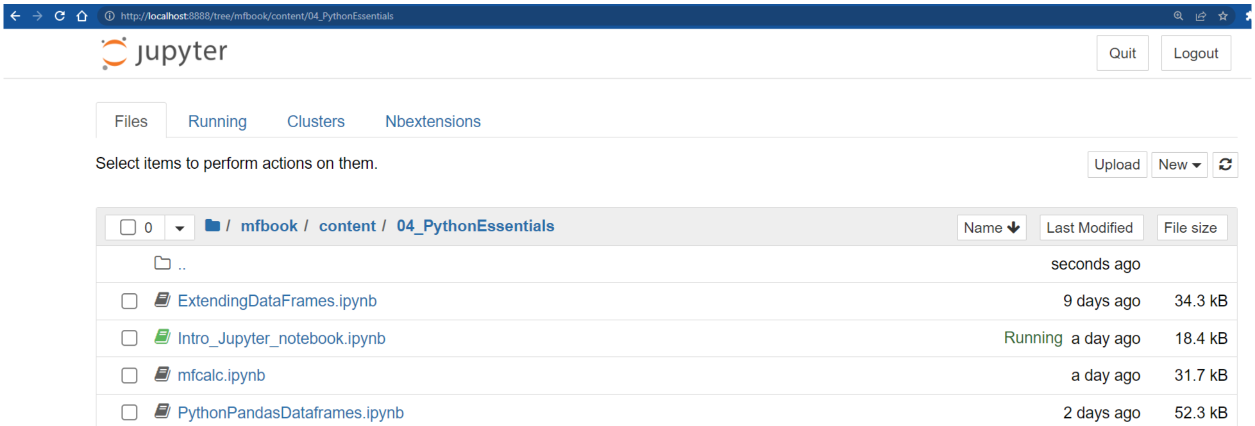
```
conda activate modelflow
```

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

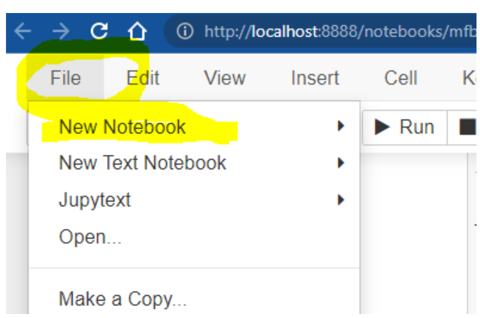


Fig. 3.1: A newly created Jupyter Notebook session

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

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

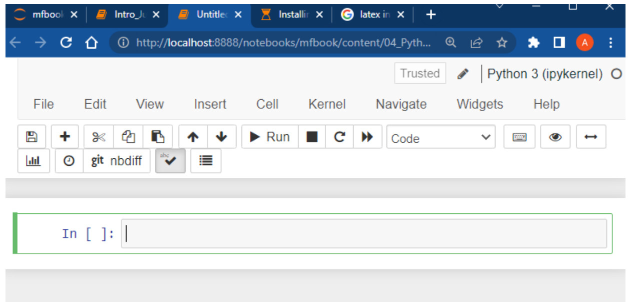


Fig. 3.2: A newly created Jupyter Notebook

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.

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

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)

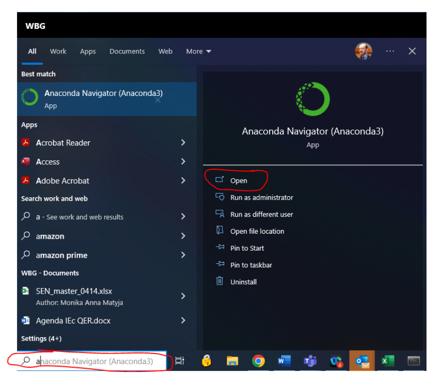


Fig. 4.1: A newly created Jupyter Notebook session

3. 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

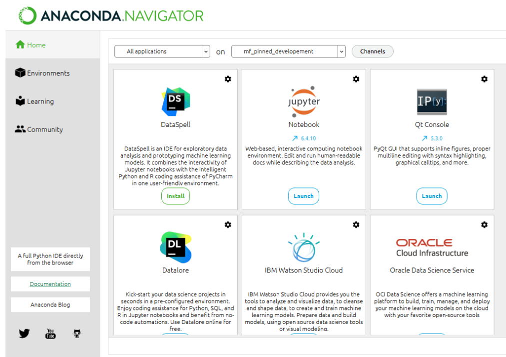


Fig. 4.2: A newly created Jupyter Notebook session

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 of a (instantiates) `DataFrame` object – i.e. a variable that is 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 can be interpreted as a time-series of an economic variable. So in this dataframe, A, B and C woudl normally be interpreted as economic time series.

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 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[]`, 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.

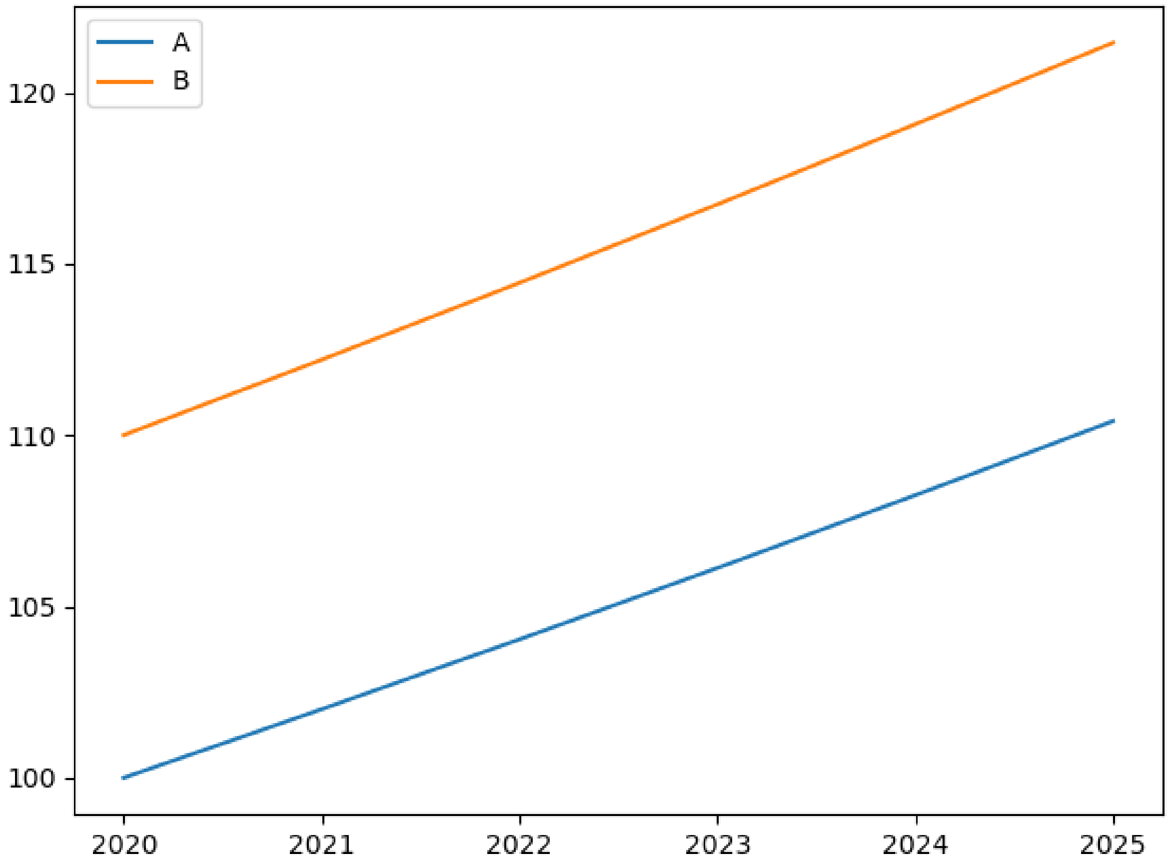
```
df = pd.DataFrame(100,
       index=[2020+v for v in range(number_of_rows)], # create row index
       # equivalent to index=[2020,2021,2022,2023,2024,2025]
       columns=['A','B'])

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()
```

```
<Axes: >
```

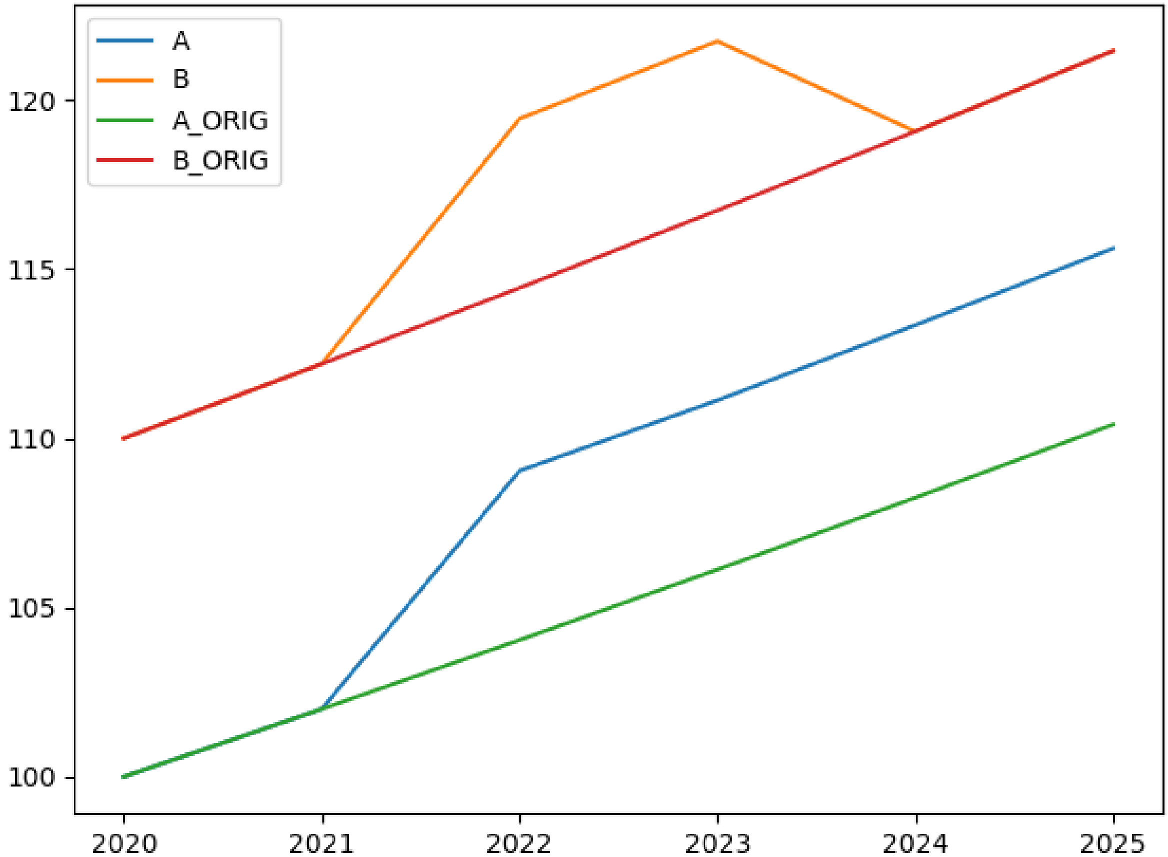


Now lets modify each by adding 5 to the level in 2022 and 2023. For B we will do setting the keep_growth option as False and for 'B' keep_growth positive. While the keep_growth is a global variable it can be set at the line level also using the –kg option (`keep_growth=True`) and –nkg option (`keep_growth=False`).

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

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

```
<Axes: >
```



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.

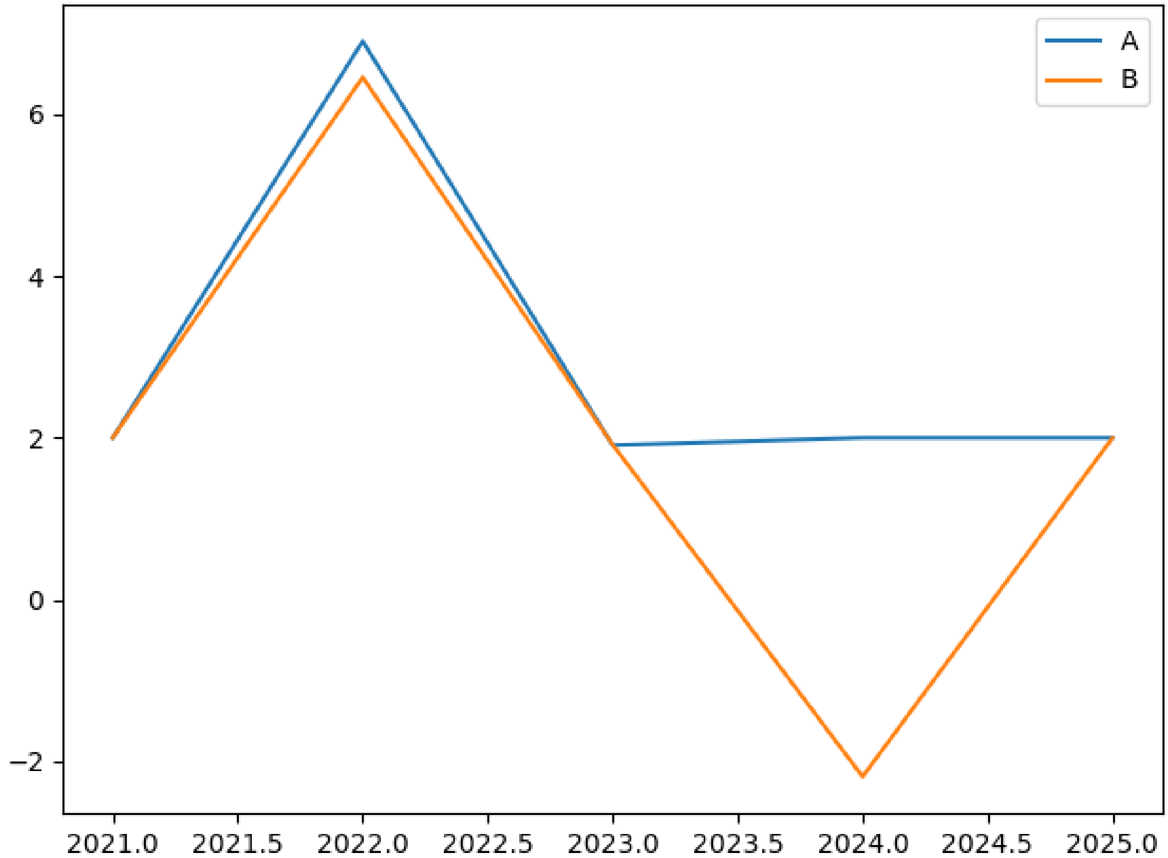
For the 'B' variable the same level change was input but because of the `--nkg` (equivalent to `keep_growth=False`) the periods after the change were unaffected and retained their old values.

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()
```

```
<Axes: >
```



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
```

(continues on next page)

(continued from previous page)

```
Growth:
        A
2020  NaN
2021  1.0
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
```

(continues on next page)

(continued from previous page)

```
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
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
```

(continues on next page)

(continued from previous page)

```
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
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
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.

```python
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
```

6.3 `.mfcalc()` in action

6.3.1 `.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
```

(continues on next page)

(continued from previous page)

```
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
```

6.3.2 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
```

6.3.3 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
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
```

6.3.4 `.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:

$$dlog(a) = 0.02 \tag{6.1}$$

$$log(a) - log(a_{t-1}) = 0.02 \tag{6.2}$$

$$log(a) = log(a_{t-1}) + 0.02 \tag{6.3}$$

$$a = e^{log(a_{t-1})+0.02} \tag{6.4}$$

$$a = a_{t-1} * e^{0.02} \tag{6.5}$$

(6.6)

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

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

6.3.5 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
```

6.3.6 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.

6.3.7 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
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()
```

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

Ib : next text has been split out and hidden in JB in the metadata

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

Ib : some changes

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 may 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`.

Ib : As it is now mpak will first look in the location specified, then it will look in the Global model repo.

```
#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')
```

```
Open file from URL:  https://raw.githubusercontent.com/IbHansen/modelflow-manual/
↪main/model_repo/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

```
mpak.PAKNECONPRVTKN
```

```
Endogeneous: PAKNECONPRVTKN: Household Consumption
Formular: FRML <Z,EXO> PAKNECONPRVTKN = (PAKNECONPRVTKN(-1)*EXP(-PAKNECONPRVTKN_A+
 ↪(-0.2*(LOG(PAKNECONPRVTKN(-1))-LOG((PAKNYYWBTOTLCN(-1)*(1-PAKGGREVDRCTXN(-1)/
 ↪100))/PAKNECONPRVTXN(-1)))+1*((LOG((PAKNYYWBTOTLCN*(1-PAKGGREVDRCTXN/100))/
 ↪PAKNECONPRVTXN))-(LOG((PAKNYYWBTOTLCN(-1)*(1-PAKGGREVDRCTXN(-1)/100))/
 ↪PAKNECONPRVTXN(-1))))+0.0303228629698929+0.0163839011059956*DURING_2010-0.
 ↪3*(PAKFMLBLPOLYXN/100-((LOG(PAKNECONPRVTXN))-(LOG(PAKNECONPRVTXN(-1)))))  ))  *
 ↪(1-PAKNECONPRVTKN_D)+ PAKNECONPRVTKN_X*PAKNECONPRVTKN_D   $

PAKNECONPRVTKN   : Household Consumption
DURING_2010      :
PAKFMLBLPOLYXN   : Policy Rate
PAKGGREVDRCTXN   : Effective tax rates
PAKNECONPRVTKN_A: Add factor:Household Consumption
PAKNECONPRVTKN_D: Exo dummy:Household Consumption
PAKNECONPRVTKN_X: Exo value:Household Consumption
PAKNECONPRVTXN   : Household demand
PAKNYYWBTOTLCN   : Economy-wide wage bill

Values :
```

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

```
Input last run:
```

```
<pandas.io.formats.style.Styler at 0x1e8de244460>
```

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 variable

Method	Example	Information returned
`.<variable name>`	`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`.

Ib : Changed XN to KN below so match with heading

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

```
['PAKNECONENGYKN', 'PAKNECONGOVTKN', 'PAKNECONOTHRKN', 'PAKNECONPRVTKN']
```

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 : Household demand
PAKNECONPRVTKN : Household Consumption
PAKNECONPRVTXN : Household demand
```

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

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

```
PAKNEHURAVERKN_        : Hurricane damage, percent of GDP
PAKNYGDPFCSTXN         : GDP at factor cost
PAKNYGDPFCSTXN_A       : Add factor:GDP at factor cost
PAKNYGDPFCSTXN_D       : Exo dummy:GDP at factor cost
PAKNYGDPFCSTXN_FITTED  : Fitted  value:GDP at factor cost
PAKNYGDPFCSTXN_X       : Exo value:GDP at factor cost
PAKNYGDPGAP_           : Output Gap (% of Potential GDP)
PAKNYGDPMKTPCD         : Nominal GDP Market Prices (USD)
PAKNYGDPMKTPCN         : GDP, LCU
PAKNYGDPMKTPKN         : GDP Expenditure side at market prices
PAKNYGDPMKTPXN         : GDP deflator
PAKNYGDPPOTLKN         : Potential GDP
```

Ib : deleted it does not return a dict

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

Ib : Error in code reflection @lru_cache(maxsize=2048), removed and works in new version

Ib : deleted #Why does this not work? mylist=['PAKNECONPRVTKN','PAKNECONGOVTKN','PAKNEGDIFTOTKN','PAKNEEXPGI
mpak[mylist].des

Ib : It is a property

The property `.var_description` is a dictionary which contains the descriptor of the associated variable. As a dictionary you can not use wildcards. If there is no description to the variable the dictionary will return the variable name it was presented with.

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

```
'GDP Expenditure side at market prices'
```

```
mpak.var_description['A_VARIABLE']
```

```
'A_VARIABLE'
```

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: Household Consumption
Formular: FRML <Z,EXO> PAKNECONPRVTKN = (PAKNECONPRVTKN(-1)*EXP(-PAKNECONPRVTKN_A+
↪(-0.2*(LOG(PAKNECONPRVTKN(-1))-LOG((PAKNYYWBTOTLCN(-1)*(1-PAKGGREVDRCTXN(-1)/
↪100))/PAKNECONPRVTXN(-1)))+1*((LOG((PAKNYYWBTOTLCN*(1-PAKGGREVDRCTXN/100))/
↪PAKNECONPRVTXN))-(LOG((PAKNYYWBTOTLCN(-1)*(1-PAKGGREVDRCTXN(-1)/100))/
↪PAKNECONPRVTXN(-1))))+0.0303228629698929+0.0163839011059956*DURING_2010-0.
↪3*(PAKFMLBLPOLYXN/100-((LOG(PAKNECONPRVTXN))-(LOG(PAKNECONPRVTXN(-1)))))  ))  *
↪(1-PAKNECONPRVTKN_D)+ PAKNECONPRVTKN_X*PAKNECONPRVTKN_D   $

PAKNECONPRVTKN  : Household Consumption
DURING_2010     :
PAKFMLBLPOLYXN  : Policy Rate
PAKGGREVDRCTXN  : Effective tax rates
PAKNECONPRVTKN_A: Add factor:Household Consumption
PAKNECONPRVTKN_D: Exo dummy:Household Consumption
PAKNECONPRVTKN_X: Exo value:Household Consumption
PAKNECONPRVTXN  : Household demand
PAKNYYWBTOTLCN  : Economy-wide wage bill
```

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
```

```
Not avaiable
```

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 $λisgreaterthanzerobutlessthanone(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 we prepare a new `ModelFlow` session, initializing a pandas session and importing and solving a saved WBG model (NB: these are precisely the same commands they we used to start the previous chapter).

```
# 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')
```

```
&lt;IPython.core.display.HTML object&gt;
```

```
Open file from URL:  https://raw.githubusercontent.com/IbHansen/modelflow-manual/
↪main/model_repo/pak.pcim
```

As noted when, as was the case of the load, the model is solved without changing any inputs the model should return (reproduce) exactly the same data as before[^fn2]. To test this for `mpak` we can compare the results from the simulation 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 in 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 is zero.

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

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[2], line 3
      1 # Need statement to change the default format
      2 mpak.smpl(2020,2030)
----> 3 mpak['PAKNYGDPMKTPKN PAKNECONPRVTKN'].difpctlevel.mul100.df

File C:\modelflow2\modelflow\modelvis.py:183, in vis.difpctlevel(self)
    181 ''' Returns the differens between the basedf and lastdf'''
    182 difdf = (self.thisdf-self.model.basedf.loc[:,self.names])/ self.model.
 ↪basedf.loc[:,self.names]
--> 183 return vis(model=self.model,df=difdf,pat=self.__pat__)

AttributeError: 'vis' object has no attribute '__pat__'
```

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 so we exclude this possibility.

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 we change the oil price increasing it by $25 for the three years between 2025 and 2027 inclusive.

To do this we first create a new input dataframe with the revised data.

Then we use the `mfcalc` method to change the value for the three years in question.

Finally we do a but of pandas math to display the initial value, the changed value and the difference between the two, confirming that the mfcalc statement did what we hoped.

```
#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 we can execute the simulation. In the command below, the simulation is run from 2020 to 2040, using the oilshockdf as the input dataframe. The results of the simulation will be put into a new dataframe `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'*.

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

Using the `modelflow` visualization tools we can see the impacts of the shock; as a print out; as charts and within Jupyter notebook as an interactive widget.

Results

The display below confirms that the shock we wanted to introduce was executed. 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):` clauses 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.

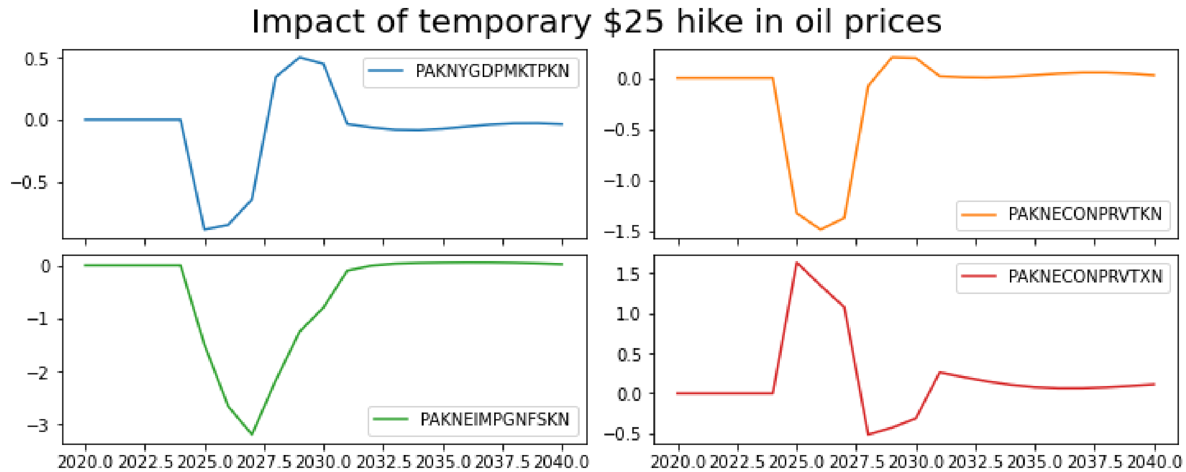
```
with mpak.set_smpl(2020,2030):
    print(mpak['WLDFCRUDE_PETRO'].dif.df);
```

```
      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 we look at the impact of this change on a few variables, expressed as a percent deviation of the variable from its pre-shock level.

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")
```



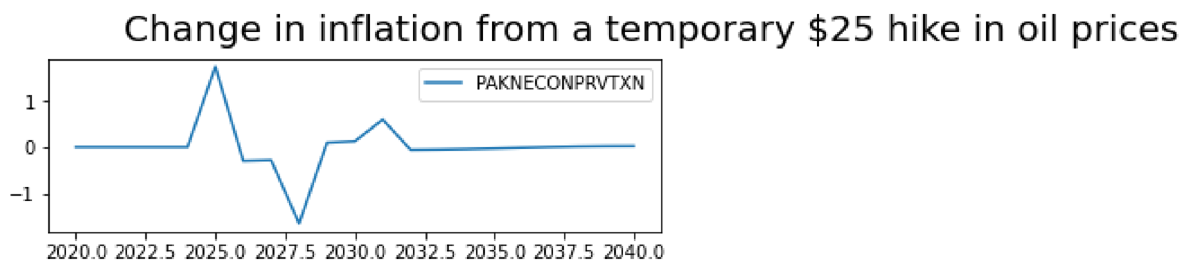

```
print(round(mpak['PAKNYGDPMKTPKN PAKNECONPRVTKN PAKNEIMPGNFSKN PAKNECONPRVTXN'].difpctlevel.mul100.df,2))
```

	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 negative impact would on household consumption would be stronger but follow a similar pattern. The reason that the GDP impact is smaller, is partly because of the impact on imports which decline strongly. Because imports enter into the GDP identity with a negative sign, lower imports actually increase aggregate GDP.

Finally as could be expected prices rise sharply initially with higher oil prices, but 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) requires a separate graph using `difpct.mul100`, which shows the change in the rate of growth of variables where the growth rate is expressed as a per cent.

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



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

This view, gives a more nuanced result. Inflation spikes initially by about 1.2 percent, but falls below its pre-shock level as the influence of the slowdown weighs on the lagged effect of higher oil prices. In 2028 when oil prices drop back to their previous level this adds to the dis-inflationary forces in the economy at first, but the boost to demand from lower prices soon translates into an acceleration in growth and higher inflation.

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 we exogenize consumption for the entire simulation period.

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

Such a shock can be specified either manually or use the method `.fix()`. 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` is created as a slightly modified version of `mpak.basedf`.

`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 – effectively transforming the equation to `PAKNECONPRVTKN=PAKNECONPRVTKN_X`. Then it 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.

```
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.

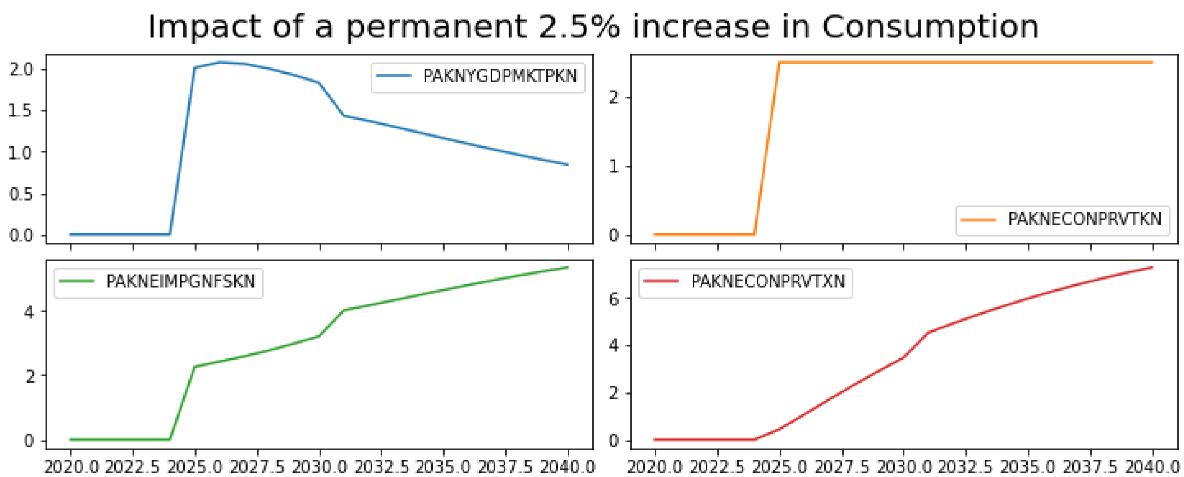
The `.upd()` method can be used to implement the assumption that Real consumption (`PAKNECONPRVTYKN`) would be 2.5% stronger.

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

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

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

```
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")
```




```
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            2.01            2.50            2.27            0.44
2026            2.07            2.50            2.43            1.06
2027            2.05            2.50            2.59            1.69
2028            1.99            2.50            2.78            2.31
2029            1.92            2.50            2.99            2.90
2030            1.83            2.50            3.22            3.47
2031            1.43            2.50            4.03            4.53
2032            1.37            2.50            4.18            4.92
2033            1.30            2.50            4.34            5.29
2034            1.23            2.50            4.50            5.64
2035            1.16            2.50            4.66            5.97
2036            1.09            2.50            4.81            6.28
2037            1.03            2.50            4.96            6.56
2038            0.96            2.50            5.10            6.82
2039            0.90            2.50            5.24            7.06
2040            0.84            2.50            5.36            7.28
```

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, while 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.

To fully explore the differences in the approaches, three scenarios are executed.

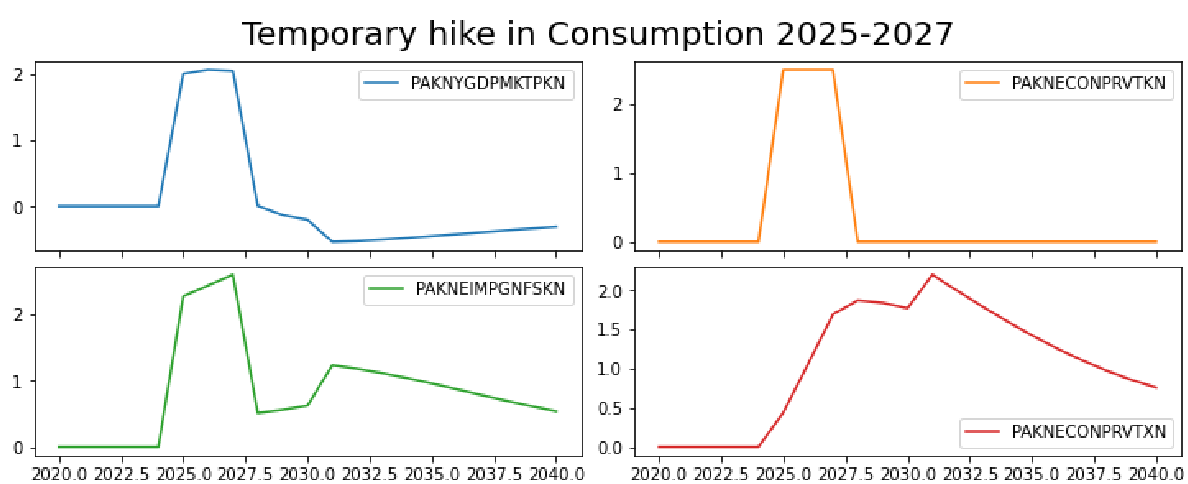
1. Exogenizes the variable for the whole period, but shock it for three years (2025-2027). Afterwards, the level of consumption falls to (and is frozen at) its pre-shock levels.
2. Exogenizes the variable for the whole period, but shock it for three years (2025-2027). On this occasion the –kg option is used to hold **the growth rates** of the exogenized variable the same in the post-shock period.
3. Exogenizes the variable only for the period during period that the dependent variable is shocked (2025-2027). Afterwards the consumption equation is activated and determines the path of post-shock consumption.

Temporary shock exogenized for the whole period

Here the set up is basically the same as before.

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

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

#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")
```



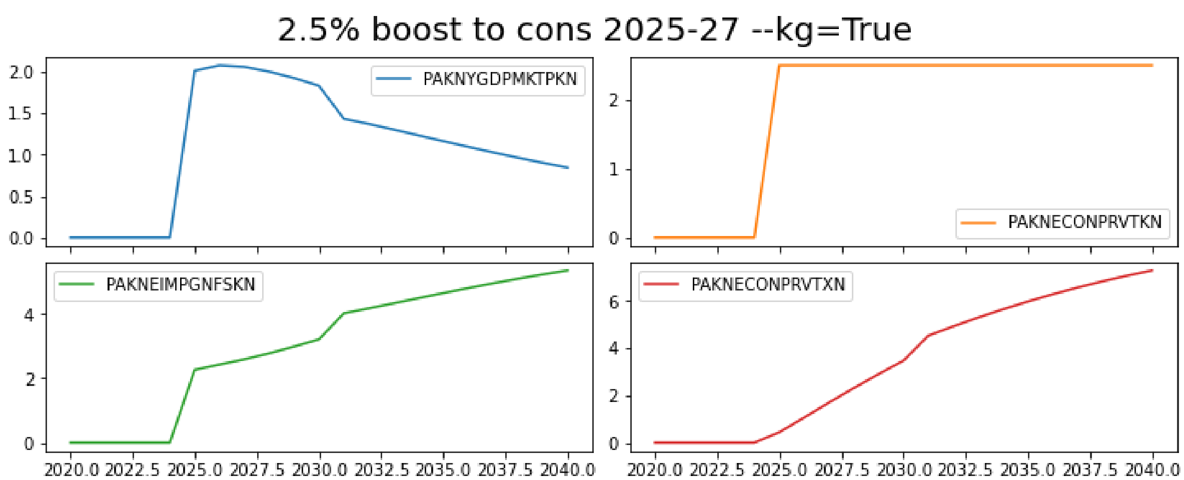
The results are quite different. GDP is boosted initially as before but when consumption drops back to its pre-shock level, GDP and imports decline sharply.

Prices (and inflation) are higher initially but when the economy starts to slow after 2025 prices start to fall (disinflation).

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 --keep_growth=True') # simulates the model
mpak['PAKNYGDPMKTPKN PAKNECONPRVTKN PAKNEIMPGNFSKN PAKNECONPRVTXN'].difpctlevel.
↪mul100.plot(title="2.5% boost to cons 2025-27 --kg=True")
```



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

This is version of our scenario introduces a subtle but import difference. Here we will exogenize the variable, again using the fix syntax. But this time we will exogonize it only for the period where the variable is shocked.

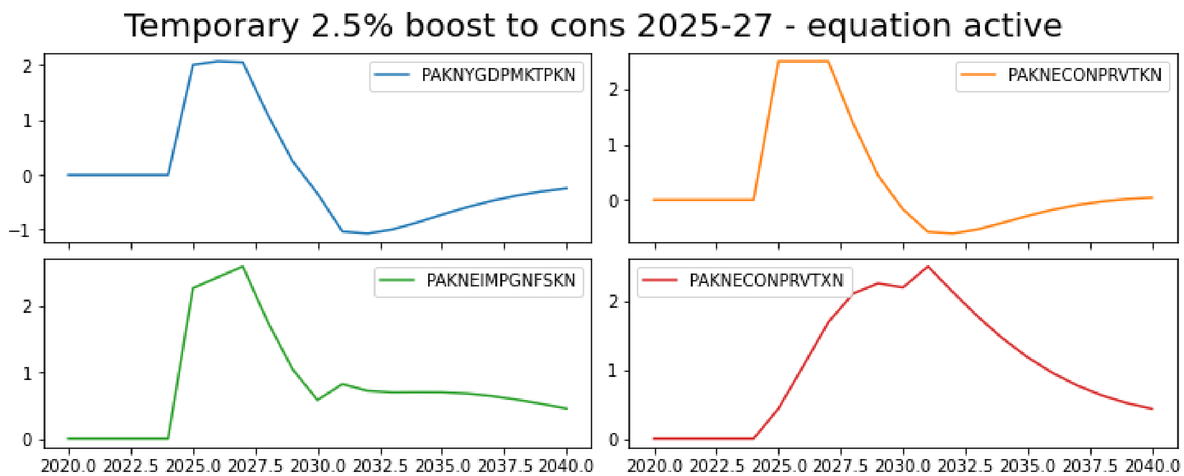
What this means is that the consumption function will be de-actvated for only three years (instead of the whole period as in the previous examples). As a result, the values consumption take in 2028, 2029, … 2040 will depend on the model, not the level it was set to when exogenized (which was the case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

```
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
```

(continues on next page)

(continued from previous page)

```
#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")
```



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, actually overshoots (falls below its earlier level) and then gradually returns 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.

```
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: 27,616,949.
Percent difference: 5.36

Real values in 2040
Base value:  38,676,995.        Shocked value: 38,693,167.
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. As such, they are often referred to as **endogenous** shocks (versus an exogenous shock).

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.
  - 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 of 1 percent of GDP per year, beginning in 2025, being financed through foreign direct investment. The add-factor approach is chosen because the additional investment is likely to increase demand for the products of other firms and have important knock on effects for investment as well as other components of demand.

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['*FB*'].des
```

```
WLDFBANANA_US              : TEMP
WLDFBANANA_US_VALUE_2010 : WLDFBANANA_US_VALUE_2010
WLDFBEEF                   : TEMP
WLDFBEEF_VALUE_2010       : WLDFBEEF_VALUE_2010
```

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.

```
mpak['PAKNEGDIFPRVKN'].frml
```

```
PAKNEGDIFPRVKN : FRML <Z,EXO> 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 $
```

Calculate the size of the required add factor shock

The shock to be executed is 1 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 a growth rate equation, therefore the add-factor needs to be shocked by adding 1 percent of GDP to private investment in 2028 divided by private investment in 2027

```
AFShock=bline

0.01*(AFShock.loc[2020:2030,'PAKNYGDPMKTPKN']/AFShock.loc[2019:2029,'PAKNEGDIFPRVKN'])


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

```
Pre shock levels
```

```
2025   -0.000458
2026   -0.000389
2027   -0.000331
2028   -0.000281
2029   -0.000239
2030   -0.000203
Name: PAKNEGDIFPRVKN_A, dtype: float64
```

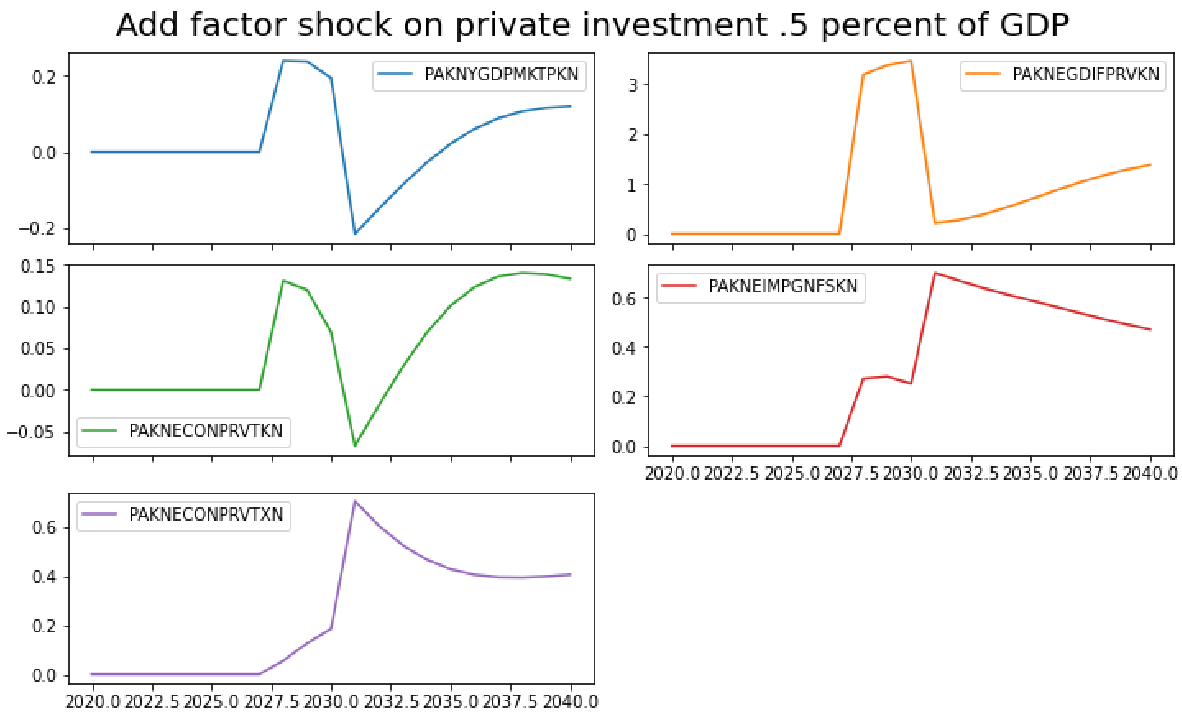
```
AFShock=AFShock.mfcalc("<2028 2028> PAKNEGDIFPRVKN_A = PAKNEGDIFPRVKN_A + .
↪00005*(PAKNYGDPMKTPKN/PAKNEGDIFPRVKN(-1))");

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

```
Post shock levels
```

```
2025   -0.000458
2026   -0.000389
2027   -0.000331
2028    0.000681
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 PAKNECONPRVTXN'].
↪difpctlevel.mul100.plot(title="Add factor shock on private investment .5 percent of␣
↪GDP")
```



8.2 The keep option

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

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.

Modelflfow has several special routines that allow results from kept scenarios to be displayed and compared.

8.2.1 The `.keep_plot()` method

For example the keep_plot method can be used to plot the value, growth rate or percent change in levels of different values across from each of the kept solutions.

Differences of growth rates

For example below we have graphs of the growth rates of GDP, Consumption and Imports from the four scenarios that we have run.

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

```
{'PAKNYGDPMKTPCN': <Figure size 720x432 with 1 Axes>,
 'PAKNECONPRVTKN': <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 PAKNEIMPGNFSKN', diffpct=1,showtype=
↪'level', legend=True);
```

8.2.2 Some variations on keep_plot(

The **variables** we want to be displayed is 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 is 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

`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.

```
mpak.fix_dummy_fixed
```

```
[]
```

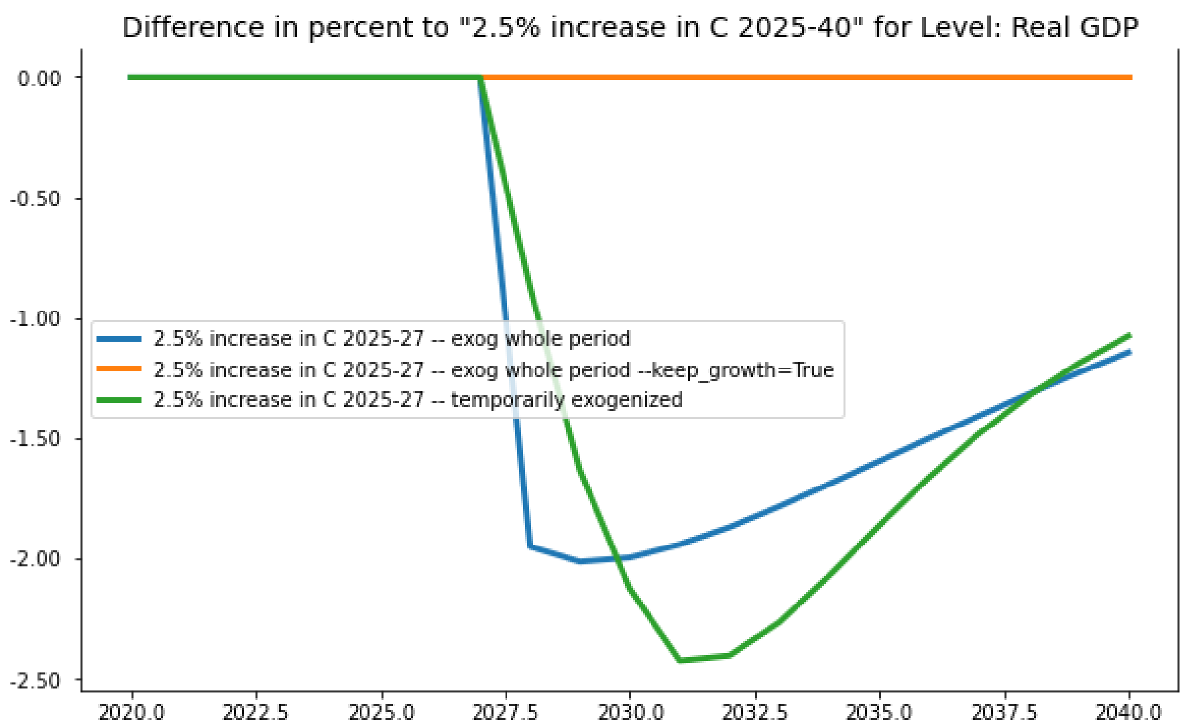
```
mpak['PAKNYGDPMKTPCN PAKNYGDPMKTPKN PAKGGEXPTOTLCN PAKGGREVTOTLCN PAKNECONGOVTKN']
```

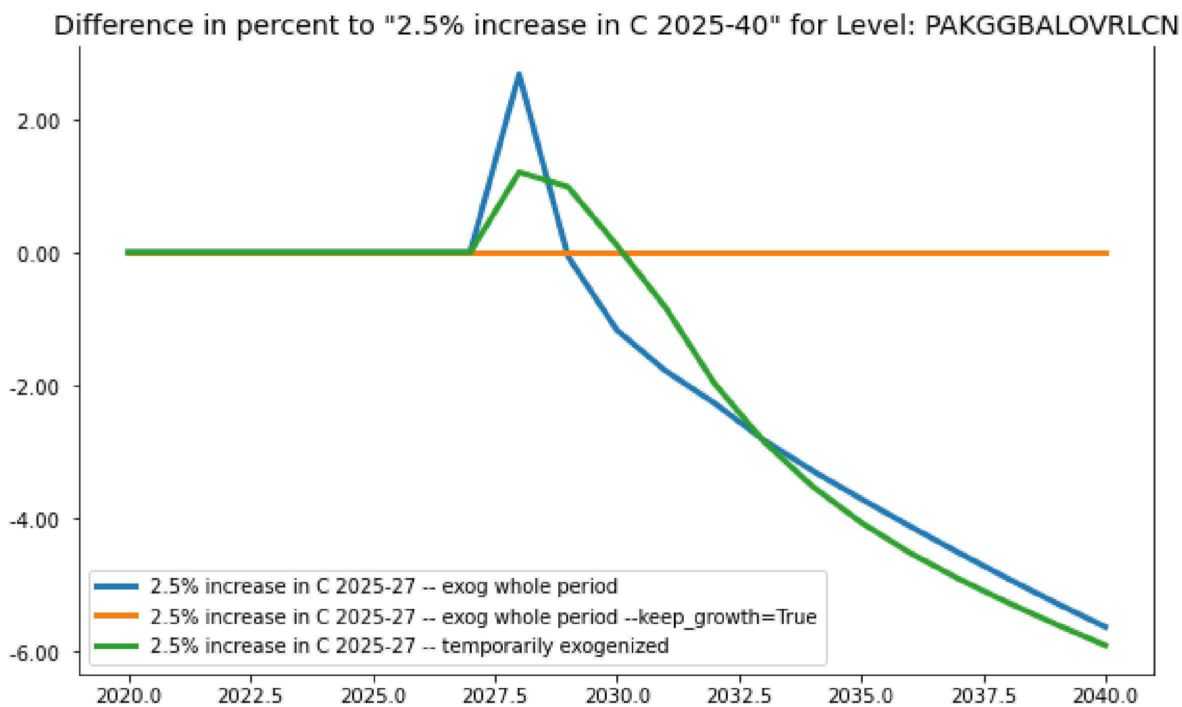
```
Tab(children=(Tab(children=(HTML(value='<?xml version="1.0" encoding="utf-8"␣
↪standalone="no"?>\n<!DOCTYPE svg …
```

```
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 --keep_growth=True', '2.5% increase in C 2025-27 --␣
↪temporarily exogenized', '1 % of GDP increase in FDI and private investment (AF␣
↪shock)'])
```

```
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 --keep_
↪growth=True|2.5% increase in C 2025-27 -- temporarily exogenized'):
    mpak.keep_plot('PAKNYGDPMKTPKN PAKGGBALOVRLCN PAKGGDEBTTOTLCN',diffpct=1,showtype=
↪'level',legend=True);
```




Difference in percent to "2.5% increase in C 2025-40" for Level: PAKGGBALOVRLCN
2.00
0.00
-2.00
-4.00
-6.00
2020.0
2022.5
2025.0
2027.5
2030.0
2032.5
2035.0
2037.5
2040.0
2.5% increase in C 2025-27 -- exog whole period
2.5% increase in C 2025-27 -- exog whole period --keep_growth=True
2.5% increase in C 2025-27 -- temporarily exogenized

Part IV

References

BIBLIOGRAPHY

[Add89] 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.

[Bla18] 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.

[BCJ+19] 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.

[BJS21a] 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.

[BJS21b] 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.

[Che71] Hollis Chenery. *Studies in Development Planning.* Harvard University Press,, Cambridge, MA., 1971.

[WB88] 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.