The World Bank's MFMod Framework in Python with Modelflow

Andrew Burns and Ib Hansen

Apr 01, 2023

CONTENTS

Andrew Burns and Ib hansen

Part I

The World Bank's MFMod Framework and Modelflow

CHAPTER ONE

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 II

Getting started

CHAPTER TWO

INTRODUCTION

> **Warning:** This Jupyter Book is work in progress.

This paper describes the implementation of the World Bank's MacroFiscalModel (MFMod) **:cite:pt:`burns_world_2019`** in 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.

2.1 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 ({cite}p:`addison_world_2019`), 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` is Burns *et al.* and .

[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 [], 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].

2.1.1 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 undeprin the economic analysis contained in many of the World Bank's Country Climate Development Reports.

2.2 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 and their maintenance, modification and revision.

2.3 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.* and here Burns *et al.* [2021], Burns *et al.* [2021]). Nor is it documentation for the specific models described and worked with below.

CHAPTER THREE

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 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]. In these articles he 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.

3.1 A system of equations

Macro-structural models are a system of equations comprised of two kinds of equations and three kinds of variables.

- `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 C = 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, would often be set as an exogenous variable because the level of activity of the economy itself is unlikely to affect the world price of oil.

In a fully general form it can be written as:

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

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

Rewritten for our GDP identity and substituting the variable mnemonics Y,C,I,G,X,M we could write a simple model as a system of 6 equations in 6 unknowns:

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

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

3.2 Behavioural equations

Behavioural equations in a macrostructural equation are typically estimated. In MFMod they are often expressed in Error Correction form. In this approach the behaviour of the dependent variable (say Consumption) is assumed to be the joint product of a long-term economic relationship – usually drawn from economic theory, and various short-run influences which can be more ad hoc in nature. The ECM formulation has the advantage of tieing down the long run behavior of the economy to economic theory, while allowing its short-run dynamics (where short-run can in some cases be 5 or more years) to reflect the way the economy actually operates (not how textbooks say it should behave).

For the consumption equation, utility maximization subject to a budget constraint might lead us to define a long run relationship like this economic theory might lead us to something like this:

$$C_t = \alpha + \beta_1 \frac{rK_t + WB_t + GTR_t(1 - \tau^{Direct})}{PC_t} - \beta_3(r_t - \dot{p}_t) + \eta_t$$

Where in the long run consumption (C_t) for a given interest rate is a stable share of real disposable income ($\frac{rK_t + WB_t + GTR_t}{PC_t}$), implying a constant savings rate. And where real disposable income is given by interest earned on capital (rK_t) plus earnings from labour (WB_t) + Government transfers to households (GTR_t) multiplied by 1 less the direct rate (τ^{Direct}). The final term reflects the influence of real interest rates on final consumption, such that as real interest rates rise consumption as a share of disposable income declines (the savings rate rises).

Replacing the expression following β with Y^{disp}_t, the above simplifies and can be rewritten as:

$$C_t = (\alpha + \beta_1 Y^{disp}_t - \beta_3(r_t - \dot{p}_t))$$

and dividing both sides by Y^{disp}_t gives:

$$\frac{C_t}{Y^{disp}_t} = \beta_1 - \beta_3 \frac{r_t - \dot{p}_t}{Y^{disp}_t}$$

or in logarithms

$$c_{t-1} - y^{disp}_{t-1} - ln(\beta_1) + \beta_3 ln(r_{t-1} - \dot{p}_{t-1} - y^{disp}_{t-1}) = 0$$

we can then write our ECM equation as

$$\Delta c_t = -\lambda(\eta_{t-1}) + SR_t$$

Substituting the LR expression for the error term in t-1 we get

$$\Delta c_t = -\lambda(c_{t-1} - y^{disp}_{t-1} - ln(\beta_1) + \beta_3 ln(r_{t-1} - \dot{p}_{t-1} - y^{disp}_{t-1})) + \beta_{SR1} y^{disp}_t - \beta_{SR2} ln(r_t - \dot{p}_t - y^{disp}_t)$$

where β_{SR1} is the short run elasticity of consumption to disposable income; β_{SR2} is the short run real interest rate elasticity of consumption and λ is the speed of adjustment (the rate at which past errors are corrected in each period).

Burns *et al.* [2019] provides more complete derivations of the functional forms for most of the behavioural equations in MFmod.

3.3 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.

3.3.1 A brief history of ModelFlow

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

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

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

CHAPTER

FOUR

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.

4.1 Installation of Python

Python is an extremely powerful and 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, it is strongly suggested that you 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`.

4.1.1 Installation of Anaconda under Windows

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

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" will substabitally increase the complexity of maintaining python on your computer.

[1] Wikipedia article on python

4.1.2 Installation of Python under macOS

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

4.1.3 Installation of Python under Linux

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

Once Anaconda is fully installed, you can then go to the following instruction on how to install `modelflow` and the packages on which it depends.

4.2 Installation of Modelflow

`Modelflow` is a python package that defines the modelflow class `model` among others. `Modelflow1` 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 chapter, please do so **Now**.

4.2.1 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 ModelFlow, and this will have been activated.

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

Note: NB: The next time you want to work with modelflow, you 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 folling command

```
conda activate modelflow
```

CHAPTER

FIVE

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.

5.1 The idea of the 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 encourage practices that will ensure that if followed exactly by others, that they will be able to generate the same results.

5.2 Jupter Notebook cells

A Jupyter Notebook does all of that (and perhaps a bit more). It is divided into '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 headers, or instruct JN 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.

The notebook has associated with it a "Kernel", which is an instance of the computing environment in which code will be executed. For JN taht work with modelflow this will be a Python Kernel.

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.

5.3 Execution of cells

Every cell in a JN 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.

5.3.1 Execution of code cells

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[#].

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

```
45
```

the semi-colon ";" supresses 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
```

5.3.2 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)

5.4 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.

5.4.1 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

5.4.2 Tables in markdown

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

Below is the markdown code that generated the above table:

```
| symbol            | Effect          |
|:--|:----------------|
| \#                | Header        |
| \#\#              | second level |
| \*\*Bold text\*\* | **Bold text**   |
| \*Italics text\* | *Italics text*   |
|
| 1\. text   | 1. Numbered bullets   |
```

5.4.3 Displaying code

To display a (unexecutable) block of code within a markdown cell, encapsulate it (sourround it) with three ` at the beginning and end. The below code entered in a markdown cell,

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

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

5.4.4 Rendering mathematics in markdown

Jupyter Notebook's inmplementation of Markdown supports $\LaTeX$è mathematical notation.

Inline enclose the $\LaTeX$è 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.

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

If you want the math to stand alone (not be in-line, then use two `$` signs)

The below block renders as below, with the `&` symbol telling $\LaTeX$è to align the different lines (separates by `\\`) on the character immediately after the `&`. Here it is used to align the texts on the space preceding the equals sign in each line.

```
\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*}
```
```

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

5.4.5 links to more info on markdown

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

5.5 How to add, delete and move cells

Jupyter Notebook cells can be added, deleted and moved.

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
- **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

5.6 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

CHAPTER

SIX

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.

6.1 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 member variables (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 classes "constructor".

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.

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.

6.2 Importing packages, libraries, modules and classes

Some libraries, packages, 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 on your system before importing them into your sessions.

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.

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.

Below, some insight into the structure and content of packages and different ways to import them into a program or Typically a python program will start with the importation of the libraries, classes and modules that will be used. Because a Jupyter Notebook.

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
```

6.2.1 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
```

6.2.2 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
```

6.2.3 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

SEVEN

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

CHAPTER

EIGHT

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 libarary contains many classes and methods. Here we are going to focus on a **Series** and **DataFrames**, each of which are very useful for time-series data.

Unlike other statistical packages neither `series` nor `dataframes` are inherently or exclusively time-series in nature. In `modelflow` and macroeconomists use them in this way, but the classes themselves are not dated in anyway out-of-the-box.

8.1 Pandas series

A pandas series is an object that holds a two dimensional array comprised of values and index.

The constructor for a pandas.Series is `pandas.Series()`. The content inside the parentheses will determine the nature of the series. 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.

8.1.1 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.

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

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

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

```
values=[2,3,4,5,-15]
weird=["Some text","Some more Text",2,3]

# Here the constructor is passed a numeric list
```

(continues on next page)

(continued from previous page)

```
Simplest=pd.Series([2,3,4,5,-15])
Simplest
```

```
0     2
1     3
2     4
3     5
4   -15
dtype: int64
```

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

```
0     2
1     3
2     4
3     5
4   -15
dtype: int64
```

```
# Here the constructor is passed a string 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
```

Constructed in this way each of these Series are automatically assigned a zero-based index.

8.1.2 Series declared using a specific index

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

```
# In this example the constructor is given both the values
# and specific values for the index
Simplest=pd.Series([2,3,4,5,-15],index=[1966,1967,1996,1999,2000])
Simplest
```

```
1966     2
1967     3
1996     4
1999     5
2000   -15
dtype: int64
```

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

```
1966     2
1967     3
1996     4
1999     5
2000   -15
dtype: int64
```

Now the Series look more like time series data!

8.1.3 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" 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:-15}
simple2=pd.Series(mydict2)
simple2
```

```
1966     2
1967     3
1968     4
1969     5
2000   -15
dtype: int64
```

8.2 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.

8.2.1 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.

8.2.2 Creating or instantiating a dataframe

Like any object we an create a dataframe by calling the dataframe constructor of the pandas class. Each class has many constructors, so there are very many ways to create a dataframe.

The code example below creates a dataframe of three columns A,B,C and indexed between 2019 and 2021. Macroeconomists may interpret the index as dates, but for pandas they are just numbers. The .DataFrame() method is constructor for the dataframe class. It takes several forms (as with series), but always returns an instance of a (instantiates) dataframe – i.e. a variable that is a dataframe.

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

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

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

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

Although less frequent, `modelflow` and pandas can also contain timeseries of matrices or vectors.

8.2.3 Adding a column to a dataframe

If we assign a value to a column that does not exist, then pandas will add a column with that name and the values of the calculation.

```
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
```

8.2.4 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. you must provide exactly as many values as there are rows). Alternatively if you provide just one, then that value will replace all of the values in the specified 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
```

8.3 Column names in Modelflow

Modelflow variable names

Modelflow places more restrictions on columnnames 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

8.4 .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
```

8.4.1 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 `modelflow` 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)` That will be the value in the next row.

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

8.4.2 .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')
```

8.4.3 .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.

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

.eval is a native dataframe method, which allows us to do calculations on a dataframe. 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 appended to dataframe that allows such calculations to be performed.

8.4.5 .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

8.5 Modelflows 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.

8.5.1 .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 us 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

`.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.

8.5.2 `.upd()` some examples

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
```

(continues on next page)

(continued from previous page)

```
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
```

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.

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
<-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
```

(continues on next page)

(continued from previous page)

```
2024   46  100    0.0  33.0
2025  100  100    0.0  33.0
```

```
**Time scope of .upd()**



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.
```
```

### 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 &lt;&gt; 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
```

(continues on next page)
```

(continued from previous page)

```
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
```

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
```

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
```

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, 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
```

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
```

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 3 4 5 6 # Add to the existing growth rate these numbers
''')
print(f'Dataframe:\n{res}\n\nGrowth:\n{res.pct_change()*100}\n')
```

```
** Error, There should be 1 values. There is: 5
** Update = A Data= [2.0, 3.0, 4.0, 5.0, 6.0] 2021 2021
```

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
Input In [11], in <cell line: 1>()
----> 1 res =df.upd('''
      2 <2021 > A =GROWTH  1  # All selected years set to the same growth rate
      3 a +growth   2 3 4 5 6 # Add to the existing growth rate these numbers
      4 ''')
      5 print(f'Dataframe:\n{res}\n\nGrowth:\n{res.pct_change()*100}\n')

File ~\.conda\envs\mf_pinned_developement_test\lib\site-packages\ModelFlow-1.0.8-
↪py3.9.egg\modelclass.py:6907, in upd.__call__(self, updates, lprint, scale,␣
↪create, keep_growth)
   6904 def __call__(self,updates, lprint=False,scale = 1.0,create=True,keep_
↪growth=False,):
   6906     indf = self._obj
-> 6907     result =   model.update(indf, updates=updates, lprint=lprint,scale =␣
↪scale,create=create,keep_growth=keep_growth,)
   6908     return result

File ~\.conda\envs\mf_pinned_developement_test\lib\site-packages\ModelFlow-1.0.8-
↪py3.9.egg\modelclass.py:1627, in Model_help_Mixin.update(indf, updates, lprint,␣
↪scale, create, keep_growth)
   1624          multiplier = list(accumulate([(1+i) for i in growth_rate],
↪operator.mul))
   1626 # print(varname,op,value,arg,sep='|')
-> 1627  update_var(df, varname.upper(), op, value,time1,time2 ,
   1628              create=create, lprint=lprint,scale = scale)
   1630  if update_growth:
   1631      lastvalue = df.loc[time2,varname]

File ~\.conda\envs\mf_pinned_developement_test\lib\site-packages\ModelFlow-1.0.8-
↪py3.9.egg\modelhelp.py:76, in update_var(databank, xvar, operator, inputval,␣
↪start, slut, create, lprint, scale)
     74     print('** Error, There should be',antalper,'values. There is:',
↪len(inputdata))
     75     print('** Update =',var,'Data=',inputdata,start,slut)
---> 76     raise Exception('wrong number of datapoints')
     77 else:
     78     inputserie=pd.Series(inputdata,current_per)*scale
```

(continues on next page)

(continued from previous page)

```
Exception: wrong number of datapoints
```

Set $\Delta = var_t - var_{t-1}$ to specified values (=diff operator)

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
```

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).

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

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

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

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

The default keep_growth behaviour

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

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

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

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

Note: At the line level:

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

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

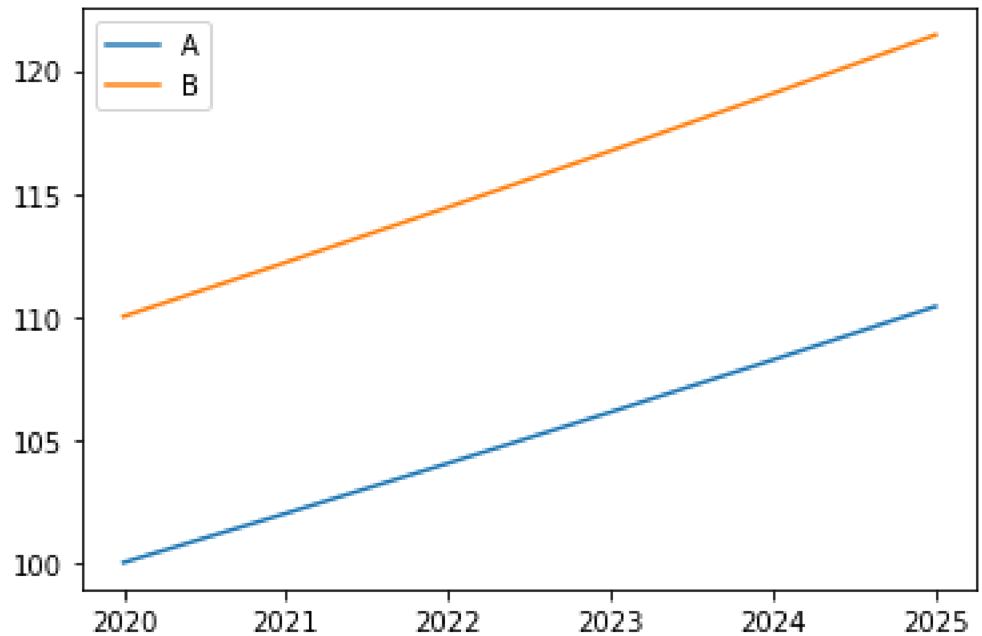
```
df = pd.DataFrame(100,
       index=[2020+v for v in range(number_of_rows)], # create row index
       # equivalent to index=[2020,2021,2022,2023,2024,2025]
       columns=['A','B'])

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

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

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

```
<AxesSubplot:>
```

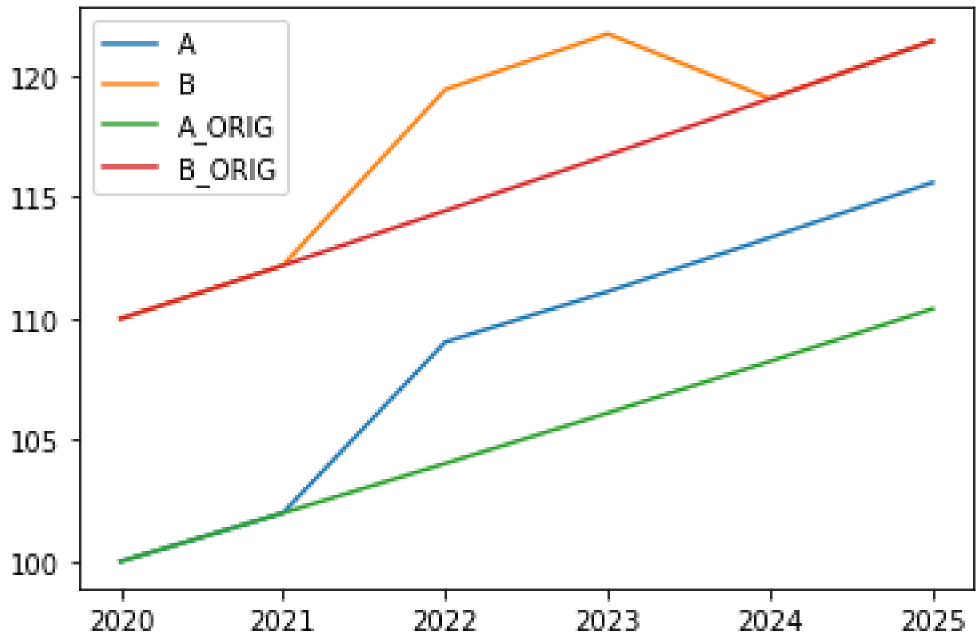


Now lets modify each by adding 5 to the level in 2022 and 2023. For B we will do setting the keep_growth option as False and for 'B' keep_growth positive. While the keep_growth is a global variable it can be set at the line level also using the –kg option (`keep_growth=True`) and –nkg option ((`keep_growth=False`).

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

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

```
<AxesSubplot:>
```



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

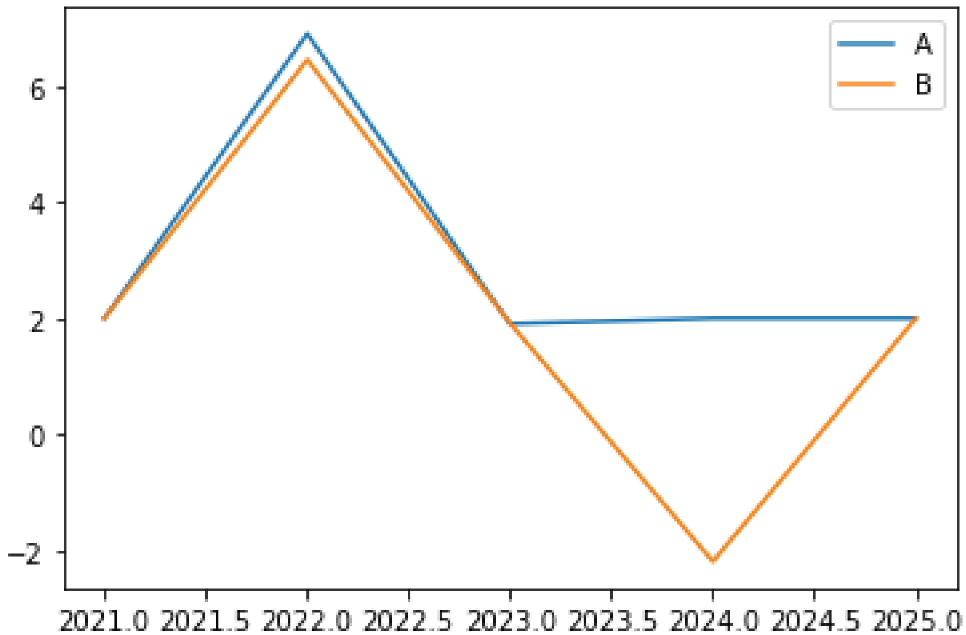
For the 'B' variable the same level change was input but because of the `--nkg` (equivalent to `keep_growth=False`) the periods after the change were affected retained their old values.

Below we plot the growth rates of the two transformed series.

Here series both series accelerate growth 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 teh growth rate of A returns to 2 percent, while the growth rate of B is actually negative because the level (see earlier graph) has fallen back to its original level.

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

```
<AxesSubplot:>
```



Note: Python constructs

print(f'Dataframe:\n{res}\n\nGrowth:\n{res.pct_change()*100}\n')

Uses

Python construct	Explanation	Links
'\n'	A line break	
dataframe.pct_change	Percentage change between the current and a prior element.	Description
f'{varname} = ….'	A f-string, {expression} is replaced by the value of expression	Search

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

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

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

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

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

Growth:
        A
2020  NaN
2021  1.0
2022  2.0
2023  3.0
2024  4.0
2025  5.0
```

now update A in 2021 to 2023 to a new value

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

In the second example we use the `-kg` line option, telling modelview 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)


print(f'growth No kg:\n{nokg.pct_change()*100}\ngrowth kg:\n{kg.pct_change()*100}\n\n\
↪nLevel No kg:\n{nokg}\nLevel kg:\n{kg}\n')
```

```
growth No kg:
               A
2020         NaN
2021   20.00000
2022    0.00000
2023    0.00000
2024   -8.03748
2025    5.00000
growth kg:
           A
2020     NaN
2021    20.0
2022     0.0
2023     0.0
```

(continues on next page)

(continued from previous page)

```
2024   4.0
2025   5.0


Level No kg:
               A
2020  100.000000
2021  120.000000
2022  120.000000
2023  120.000000
2024  110.355024
2025  115.872775
Level kg:
           A
2020  100.00
2021  120.00
2022  120.00
2023  120.00
2024  124.80
2025  131.04
```

Note: In the first 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

.upd() with the option keep_growth set globally

Above we used the line level option –keep_growth or –kg to keep the growth rate for a given operation.

This works because by default the option Keep_growth is set to false, so in effect we are temporarily setting it to true for the specific lines above.

The keep_growth variable 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
```

```
dfres = dftest.upd('''
<2022 2023> c = 200
<2022 2023> d = 300
<2022 2023> e = 400  --no_keep_growth
''',keep_growth=True)  # <===  Here we have set the 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
```

Some more advanced examples

These examples continue to use update, but with some examples of how to embed Python loops into commands.

First create a string with update lines

In this example we create a string dynamically is comprised of a variety of update statements. The loop repeats the two lines above replacing the {varname} expression with c d e and f as the loop is executed.

```
lines = '\n'.join(
 [f'''<2020     > {varname} = 100
      <2021 2025> {varname} =growth 1 2 3 4 5'''
     for varname in 'c d e f'.split()])
print(lines)
```

```
<2020     > c = 100
      <2021 2025> c =growth 1 2 3 4 5
<2020     > d = 100
      <2021 2025> d =growth 1 2 3 4 5
<2020     > e = 100
      <2021 2025> e =growth 1 2 3 4 5
<2020     > f = 100
      <2021 2025> f =growth 1 2 3 4 5
```

Note: ***Python constructs***

The creation of update lines involves a number of useful python constructs. A short description:

Python construct	explanation	Google
'a b'.split()	splits a string by `blanks`into a list	Search
'\n'.join()	Creates a string from a list of string separated by \n (linebreak)	Search
f'{varname} = ….'	A f-string, {varname} is replaced by the value of varname	Search
[varname for varname in a_list]	List comprehension which creates an implicit loop	Search

Use the update lines to update a dataframe

Here we pass the variable lines to the `upd` method.

```
dfnew = df.upd(lines)
dfnew
```

```
        A      B           C           D           E           F
2020  100  100.0  100.000000  100.000000  100.000000  100.000000
2021  100  100.0  101.000000  101.000000  101.000000  101.000000
2022  100  100.0  103.020000  103.020000  103.020000  103.020000
2023  100  100.0  106.110600  106.110600  106.110600  106.110600
2024  100  100.0  110.355024  110.355024  110.355024  110.355024
2025  100  100.0  115.872775  115.872775  115.872775  115.872775
```

8.5.4 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     H     I     J     K            P            Q  \
2020  100  100.0   0.0   0.0   0.0   0.0  1000.000000  1000.000000
2021  100  100.0   0.0   0.0   0.0   0.0  1020.000000  1020.000000
2022  100  100.0  40.0  40.0  40.0  40.0  1040.400000  1040.400000
2023  100  100.0  40.0  40.0  40.0  40.0  1061.208000  1061.208000
2024  100  100.0  40.0  40.0  40.0  40.0  1082.432160  1082.432160
2025  100  100.0   0.0   0.0   0.0   0.0  1104.080803  1104.080803

                R            S
2020  1000.000000  1000.000000
2021  1020.000000  1020.000000
2022  1040.400000  1040.400000
2023  1061.208000  1061.208000
```

(continues on next page)

(continued from previous page)

```
2024  1082.432160  1082.432160
2025  1104.080803  1104.080803
```

8.5.5 .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
2020  100  100.0
2021  100  100.0
2022  100  100.0
2023  100  100.0
2024  100  100.0
2025  100  100.0


severity=0
Dataframe:
        A      B
2020  100  100.0
2021  100  100.0
2022  100  100.0
2023  100  100.0
2024  100  100.0
2025  100  100.0
```

(continues on next page)

(continued from previous page)

```
Growth:
        A    B
2020  NaN  NaN
2021  0.0  0.0
2022  0.0  0.0
2023  0.0  0.0
2024  0.0  0.0
2025  0.0  0.0


severity=0.5
Dataframe:
               A      B
2020  100.000000  100.0
2021  100.500000  105.0
2022  101.505000  105.0
2023  103.027575  105.0
2024  105.088126  105.0
2025  107.715330  105.0

Growth:
        A    B
2020  NaN  NaN
2021  0.5  5.0
2022  1.0  0.0
2023  1.5  0.0
2024  2.0  0.0
2025  2.5  0.0


severity=1
Dataframe:
               A      B
2020  100.000000  100.0
2021  101.000000  110.0
2022  103.020000  110.0
2023  106.110600  110.0
2024  110.355024  110.0
2025  115.872775  110.0

Growth:
        A     B
2020  NaN   NaN
2021  1.0  10.0
2022  2.0   0.0
2023  3.0   0.0
2024  4.0   0.0
2025  5.0   0.0
```

8.5.6 .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               100.0000            4200.0000            4100.0000
2022               100.0000            4400.0000            4300.0000
```

```
         A      B
2020   100  100.0
2021  4200  100.0
2022  4400  100.0
2023   100  100.0
2024   100  100.0
2025   100  100.0
```

8.5.7 .upd(„create=True) Requires the variable to exist

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

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

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

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

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

8.5.8 The call

def upd(indf, updates, lprint=False,scale = 1.0,create=True,keep_growth=False,start=",end=")

```
Args:
        indf (DataFrame): input dataframe.
        basis (string): string specifying line-by-line the updates to be done
        lprint (bool, optional): if True each update is printed  Defaults to False.
        scale (float, optional): A multiplier used on all update input . Defaults to␣
↪1.0.
        create (bool, optional): Creates a variables if not in the dataframe .␣
↪Defaults to **True**.
        keep_growth(bool, optional): Keep the growth rate after the update time frame.
↪ Defaults to False.

    Returns:
        df (TYPE): the updated dataframe .

    A line in updates looks like this:

    "<"[[start] end]">" <var....> <=|+|*|%|=growth|+growth|=diff> <value>...  [--keep_
↪growth_rate|--no_keep_growth_rate]
```

8.6 `.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.

8.6.1 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
```

8.6.2 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
```

8.7 `.mfcalc()` in action

8.7.1 `.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')
```

```
          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
```

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.

Combining these two behaviours generates the result where the command `df.mfcalc('x = x(-1) + a')` results in a zero in 2020 for X (because there was no X variable defined for 2019 (indeed no such row exists), but then the subsequent rows add the contemporaneous value of A to the preceding value of x.

Once again, the result of the .mfcalc is displayed. However, because the results from `df.mfcalc()` call was not assigned to a variable (no equals sign to the left of the call), the output of the command is displayed but not stored.

Note: In the above example a dataframe with the result is created and displayed, but the df dataframe did not change. To have it change we would have had to assign it the result of the initial operation, as below.

```
df
```

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

8.7.2 Stopring the result of an `.mfcalc()` call

In this instance, the results of the `.mfcalc()` call is assigned to the variable df2 and therefore stored.

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

```
          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
```

8.7.3 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
```

```
               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
```

8.7.4 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);
```

```
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:

$$dlog(a) = 0.02 < br > log(a) - log(a_{t-1}) = .02 < br > log(a) = log(a_{t-1}) + .02 < br > a = e^{log(a_{t-1})+0.02} < br > a = a_{t-1} * e^{0.02}$$

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

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

```
df
```

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

8.7.5 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)
```

```
FRML <> A=A(-1)+(2)$
```

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

8.7.6 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
```

```
          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
```

Here the diff(a) is not defined for 2020 because there is no value for a in 2019.

As a result `modelflow` generates a result only for the periods 2021 through 2025 and it is this result that is passed to the second equation, which adds 42 to this number. 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 not undertaken in two separate calls to `.mfcalc()`.

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

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

```
          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 is 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 you would see if you ran the two commands sequentially.

8.7.7 Setting a time frame with mfcalc.

It can useful in some circumstances to limit the time frame for which the calculations are performed. By specifying a start date and end date enclosed in <> in a line we can restrict the time period over which calculation is performed.

Below, as in the example above we have zeroes 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

The World Bank's MFMod Framework and Modelflow

CHAPTER NINE

A SIMPLE MACROSTRUCTURAL MODEL IN MODELFLOW

`Modelflow` is a sophisticated tool that can deal with extremely large and complicated models, including the Federal Reserve's FRB/US model and the World Bank's climate-aware macrostructural models. In this chapter we illustrate some of the main features of `modelflow` using a very simple macrostructural model.

In the following chapter we use `modelflow` with a full-blown macro-structural model, and examine some of the more advanced features of the `modelflow` class.

9.1 Setting up the environment

As always, the python environment needs to be set up by importing the classes and modules upon which the following program(s) will depend.

```
%matplotlib Notebook
from modelclass import model
from modelgrabwf2 import GrabWfModel
import modelpattern as pt #Allows pattern a selections from model structures
import re
import pandas as pd
model.widescreen()
model.scroll_off()
%load_ext autoreload
%autoreload 2
```

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

9.2 A simple model

In this simple example we will load a simple real-side only macroeconomic model that was created in EViews. The model structure is simple. Its i comprised of two identities:

$$Y_t = CPV_t + I_t + G_t + (X_t - M_t) + Y_t^{statdisc}$$

$$GDE_t = CPV_t + I_t + G_t + X_t$$

and four behavioural equations variables for private consumption (CPV), Investment (I), for Government spending (G) and Imports (M).

$$CPV_t = C'(\chi_t) + \eta_t^C$$

$$I_t = I'(\chi_t) + \eta_t^I$$
$$G_t = G'(\chi_t) + \eta_t^G$$
$$M_t = X'(\chi_t) + \eta_t^M$$

and two exogenous variables (X for exports and $Y^{statdisc}$ for the statistical discrepancy.

Each of the behaviourals is a simple error correction equation written as :

$$\Delta var_t = -\gamma * (var_{t-1} - base_{t-1} - \beta_2)2) + \Delta base_t$$

where for each $var \in (CPV, I, G)$ the base is Y, while for M it is GDE.

9.2.1 Load the method `.modelload()`

The modelflow method `.modellow` opens a pre-existing modelflow model, and assigns the variable `msimple` with the model object created by model.load. The variable `init` is assigned the value of the `dataframe` associated with the

---

Note: The variable names `msimple` and `init` are completely arbitrary and could be any legal python name.

```
msim,init = model.modelload(r'../models/simple.pcim',run=1,silent=1)
```

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

Below, we solve the model over the period 2016 to 2030, initializing it with the initial data loaded above.

The options:

- **silent=1** limits reporting as the model is solved, which ensures faster operation;
- **alfa= .5** influences the step-size when the model is solved. alfa= 1 implies larger step sizes and faster solution, but may prevent the model from finding a solution, smaller step sizes are more computationally expensive but increase the likelihood that solutions will be found.
- **ldumpvars** controls whether the model should store intermediate results as it iterates towards the final solution. ldumpvar=1 retains these intermeidate results, which may be useful in determining which equation if any is causing trouble in model solution.

```
res = msim(init,2016,2030,silent=1,alfa=.5,ldumpvar=0)#ldumpvar saves iterations 0 =>␣
↪don't;
                                                        #alfa <1 reduces step size␣
↪when iterating
```

9.3 Extract information about the model

A macrostructural model is a system of equations comprised of identities (accounting rules that are always true), estimated behavioural equations and exogenous variables.

For our simple model, the identities are Y=C+I+G+X-M+STatDisc, and the behavioural equations (or stochastic equations) are CPV,I, G, M, with X and StatDisc being exogenous variables.

We can use the msim.identity(); msim.stoch() and msim.exogenous() functiosn to extract lists of the variables of each of these types in the model.

As a class `model` has methods and properties. Methods perform actions on the data of the class, and properties are effectively the data associated with an instance of a class (msim in our case).

When we created the model we included in it both identities, behavioural equations and implicitly exogenous variables.

Both identities and behaviourals are endogenous variables (model determined), while exogenous variables are provided by the modeller and condition the model forecast.

The following methods returns lists of variable mnemonics from the *economic* model based on their economic role in the model as: identities, behavoural equations or exogenous variables.

`model` property	Explanation
`.model_identity`	Returns a python list of the mnemonics of all identities in the model
`.model_stochastic`	Returns a python list of the mnemonics of all behavioural (or stochastic) equations in the model
`.model_endogene`	Returns a python list of the mnemonics of all endogenous variables in the model (Identities and Behavioral)
`.model_exogene`	Returns a python list of the mnemonics of all exogenous variables in the model

The mathematical model includes some additional "helper" variables that are mathematically either endogenous or exogenous in the model. Mathematically there is no real difference between an identity equation and a behavioural equation. The "helper" variables allow us to treat behavioral equations differently than identities in a way that make sense economically. The following methods return lists that include both the "economic" variables listed above and these helper variables that form part of the mathematical model.

`model` property	Explanation
`.endogene`	Lists all endogenous variables in the model (Identities and behaviourals)
`.exogene()`	Lists all exogenous variables in the model

These will have to be updated with the embodied calls when available

9.3.1 List all identities in the model

```
ident = {v for v in msim.endogene if pt.kw_frml_name(msim.allvar[v]['frmlname'], '') ␣
↪}
#ident=msim.model_identity()
ident
```

```
{'GDE', 'Y'}
```

9.3.2 List all behavioural equations in the model

```
stoc = {v for v in msim.endogene if pt.kw_frml_name(msim.allvar[v]['frmlname'], 'Z') ␣
↪}
#stoch=msim.model_stochastic()
stoc
```

```
{'CPV', 'G', 'I', 'M'}
```

9.3.3 List all exogenous variables in the model

```
#exog=msim.model_exogene()
exog = {v for v in msim.exogene if not '_' in v  }
exog
```

```
{'X', 'YDISC'}
```

9.4 Equations in a `modelflow` model

As noted earlier, a macrostructural model is comprised of identities, behavioural equations and exogenous variables.

9.4.1 Identities

Identities are accounting rules that are always true. GDP is an identity because GDP is identically equal to C+I+G+(X-M)+ YDISC. The Fiscal balance (Deficit when negative) is an identity $Fisc_t^{Balance} = Fisc_t^{Revenues} - Fisc_t^{Expenditure}$ etc.

9.4.2 Behavioural equations: Fitted Values and Add Factors

In World Bank models, behavioural equations are split into two parts. The fitted value of the equation and an add factor. This split derives naturally from the econometrics of behavioural equations.

Below is a standard regression equation for a linear equation.

$$y_t = \alpha + \beta X_t + \eta_t$$

Let $\hat{\alpha}$ and $\hat{\beta}$ represent the econometrically estimated values of α and β above, then we can define the fitted value for y_t ($\hat{y}_t$) as:

$$\hat{y}_t \equiv \hat{\alpha} + \hat{\beta} X_t$$

We can then define the add factor for the behavioural variable y as (y_t^{AF}) as

$$y_t^{AF} \equiv \hat{y}_t - (\hat{\alpha} + \hat{\beta} X_t)$$

Over the historical period, *Add Factors* are assigned values that ensure that the sum of the fitted value and its add-factor exactly equals the observed historical value. **In the historical period the Add Factor Equals the regression error term.**

Over the forecast period, the regression error term η_t does not exist (as there is no data with which to calculate it). By retaining the *Add Factor*, the model has a mechanism that allows the modeller to cause the forecast to deviate from the pathway that would be dictated by the fitted values of the equation.

In World Bank models add-factors for behavioural equations (they are only defined for behavioural equations) are indicated by adding _A to the variable name.

Important: Reproducibility

Over the historical period, the Add-Factor ensures that the model *reproduces* history.

Reproducibility is an essential quality for a macro model.

In forecast mode, the Add-Factor allows the forecast to deviate from the fitted value of the behavioral equations of the model – reflecting the judgment of the analyst.

In simulations, the Add-Factor allows for the path of endogenous behavioural variables to be shocked by specific amounts and over specific time periods. Shocked in this way the equation for the model remains active and can react endogenously through the simulation period to the influence of the shock.

9.4.3 Using Add factors when forecasting

When building a baseline forecast, a modeller can use the **Add Factor** to add his own judgment to the forecast value for a variable.

For example, suppose the fitted value for Consumption was 100 in 2023. Effectively this says the conditional forecast of the model for Consumption (CPV) given (conditioned upon) the level of all the other variables is 100.

$$CPV\_FITTED_{2023} \equiv C\hat{P}V_{2023} = \hat{\alpha} + \hat{\beta}X_2023$$

If a forecaster had information that the model did not, say the onset of Covid earlier that year (or of a major storm), s/he good add to this conditional forecast their judgement that consumption is expected to be to be 20 units lower than the 100 expected by the model.

The fully formed equation then becomes

$$CPV_{2023} = CPV\_FITTED_{2023} + CPV\_A_{2023}$$

or \$$CPV_{2023} = 80 = 100 + (-20)$\$

Note: In addition to the _A (Add Factor) variable, modelflow also generates an _FITTED variable that holds the conditional forecast of the model for that variable at any given time. The forecast is conditional because it is conditioned on the state of the other variables (the X_t in the regression equation).

NB: The _FITTED variable is calculated by solving the behavioural equation with all add factors set to zero. :::

9.4.4 Extracting information about equations

`Modelflow` contains two methods to display equations from the model. The first `.frml` displays the formula for selected variables as it has been translated into the business logic language of `modelflow`.

The `.frml` method

When equations are displayed using the `.frml` method in the Business logic language of `modelflow`. in `business logic`, all equations are normalized, such that the normalized equation solves for the level of the dependent variable.

`.frml` output of a simple identity

For simple identities like GDP, the Y variable in the simple model `msim`, the normalized version of the model equation is the same as the input equation because it was originally normalized.

```
msim['Y'].frml
```

```
Y : FRML <> Y = CPV+I+G+X-M+YDISC $
```

In the output, the initial field (before the :) shows the dependent variable that the equation determines, the part following that is the actual FRML equation with the text between <> indicating the features of the particular equation, in this case the blank space indicates it is an Identity.

`.frml` output of a behavioural

For a more complex equation, such as say the ECM equation of our simple consumption equation, the normalized output will differ from the original specification.

Thus for an original (simple) ECM style equation that might have looked like this:

$$\Delta ln(C_t) = \beta_2(ln(C_{t-1}) - ln(Y_{t-1}) + \beta_1) + \beta_{10}\Delta ln(Y_t)$$

The normalized version would look like

$$ln(C_t) = ln(C_{t-1}) + \beta_2(ln(C_{t-1}) - ln(Y_{t-1}) + \beta_1) + \beta_{10}\Delta ln(Y_t) + AF_t$$

$$C_t = C_{t-1} * e^{(\beta_2(ln(C_{t-1}) - ln(Y_{t-1}) + \beta_1) + \beta_{10}\Delta ln(Y_t) + AF_t)}$$

The normalized version of the consumption equation in `msim` is given below:

```
msim['CPV'].frml
```

```
CPV : FRML <Z,EXO> CPV = (CPV(-1)*EXP(CPV_A+ (-.3*(LOG(CPV(-1))-LOG(Y(-1))-LOG(0.
↪866239851149167))+0.0237316411085375*((LOG(Y))-(LOG(Y(-1))))) )) * (1-CPV_D)+␣
↪CPV_X*CPV_D $
```

As before, the first part of the `.frml` output indicates the mnemonic of the behavioural variable that the formula determines (in this case CPV). This is followed by a `FRML` statement (the actual Business Logic formulation generated by `modelflow`). The `FRML` is the normalized version of the actual equation submitted – in this case a logarithmic growth equation, normalized to solve for the level of the dependent variable.).

The above `FRML` statement indicates that this is a behavioural equation (the Z between the <>, that can be exogenized (EXO). Where exogenized means that the equation can be turned off and the value of the behavioural equation set to a specific value determined by the modeller.

Note: Behavioural equations can be exogenized. Ecogenizing, effectively de-activates the equation, allowing the modeller to impose a value on the dependent variable of the equation that is different from that which the equation would return.

Equations can be exogenized either to impose the judgement of the analyst in forecasting mode, or to perform what if scenarios.

Automatically generated variables associated with behavioural equations

Behavioural equations like CPV above include three automatically generated variables that form part of the mathematical model that is actually solved by `modelflow`, but are not part of the "economic model". These three variables are formed by by adding _A _X _D to the dependent variables of the dependent variable for each behavioural equation in the model.

The first of these _A is the add factor discussed above. The second (_D) is a dummy variable which when it has the value zero indicates that the estimated equation will be used to determine the value of the dependent variable in a behaviorual equation. When the (_D) variable has the value of 1, then the equation is said to be exogenized or de-activated and the dependent variable will be set equal to the _X variable.

In addition, `modelflow` also generates one reporting variable _FITTED (discussed above) which contains the value of teh conditional forecast of the behavioural equation for the dependent variable.

Suffix	Name	Role
_A	Add Factor	Used to impose (add) judgement to the fitted value of a behavioural equation (see following section)
_D	Exog Switch	A special dummy variable that determines whether a behavioural equation is turned
_X	Exog Value	Value taken by an exogenized variable (if _D=1)
_FIT-TED	Fitted Value	The result of the behavioural equation when solved for X_t but with add factors equal to zero.

Function of the _X _D variables in the model

The `.frml` method returns the normalized version of the initial equation **– multiplied by the (1-varame_D)** + plus varname_X*varname_D).

This expression effectively defines two equations for the dependent variable. In the first instance (when varname_D=0) the varname will follow the normalized equation. But when varame_D=1. The first expression resolved to zero, and the second expression varname_D*varname_X determines the level of the dependent variable setting it to the value of varname_X.

Setting varname_D=1 effectively turns the equation off and makes the equation a simple identity where varname=varname_X.

The normalized equation with the extra variables that allow it to be exogenized.

$$C_t = \left(C_{t-1} * e^{\beta_2(ln(C_{t-1})-ln(Y_{t-1})+\beta_1)+\beta_{10}\Delta ln(Y_t)+AF_t}\right) * (1 - CPV\_D_t) + CPV\_D_t * CPV\_X_t$$

When $CPV\_D_t = 0$ this simplifies to

$$C_t = \left(C_{t-1} * e^{\beta_2(ln(C_{t-1})-ln(Y_{t-1})+\beta_1)+\beta_{10}\Delta ln(Y_t)+AF_t}\right)$$

When $CPV\_D_t = 1$ this simplifies to:

$$C_t = CPV\_X_t$$

Important: Setting the _D variable equal to one effectively turns the equation off. It **exogenizes** the endogenous variable, setting its value to the value of the _X variable. This can be done for the whole period or just a sub period.

Passing multiple variables to `.frml`

In addition to extracting only one variable you can extract the formulae of many variables by just widening the selection criteria.

Thus `msim['Y CPV']` returns the formulae for both GDP and consumption.

```
msim['Y CPV'].frml
```

```
Y   : FRML <> Y = CPV+I+G+X-M+YDISC $
CPV : FRML <Z,EXO> CPV = (CPV(-1)*EXP(CPV_A+ (-.3*(LOG(CPV(-1))-LOG(Y(-1))-LOG(0.
↪866239851149167))+0.0237316411085375*((LOG(Y))-(LOG(Y(-1)))) )) * (1-CPV_D)+␣
↪CPV_X*CPV_D $
```

Note that the formula for Y is an identity, as such there is no _A _X _D (or _FITTED) variables. Moreover, the <> expression contains nothing **This will have to be changed when new version of modelflow released.** because it cannot because identities cannot be exogenized.

9.4.5 The mathemetically endogenous and exogenous variables of the model

Because in `modelflow` the *economic* model is augmented with the above variables _A, _D, _X, _FITTED the set of mathematically exogenous and endogenous variables is larger. These sets can be retrieved with the methods: `.endogene` and `exogene`.

#####The mathematically exogenous variables of our simple model.

```
msim.exogene
```

```
{'CPV_A',
 'CPV_D',
 'CPV_X',
 'G_A',
 'G_D',
 'G_X',
 'I_A',
 'I_D',
 'I_X',
 'M_A',
 'M_D',
 'M_X',
 'X',
 'YDISC'}
```

The mathematically endogenous variables in our model.

Note this includes both identities and behavioural equations, because mathematically each is an endogenous equation – the distinction identity vs behavioural is important economically but has no meaning mathematically. Each equation determines the value of a variable in the system of equations that constitute the model.

Note the reporting variables _FITTED are mathematically endogenous. They form part of the model even if they do not interact with any other variables in the model.

```
msim.endogene
```

```
{'CPV',
 'CPV_FITTED',
 'G',
 'GDE',
 'G_FITTED',
 'I',
 'I_FITTED',
 'M',
 'M_FITTED',
 'Y'}
```

9.5 Data storage in modelflow

Modelflow uses the pandas dataframe system to store data. Every model instance will have at least two dataframes `.lastdf` and `.basedf`. The first contains the results of the most recent simulation, and the second contains the initial or baseline values of the data prior to the running of any simulations.

Following our load and test solving of our simple model, we can inspect the values for each of these dataframes.

Below we are using standard pandas functions and python constructs to

1. set the display format we want to use the `with pd.option_context('display.float_format', '{:,.6f}'.format):` line
2. Indicate what we want to display – here the results of a manipulation of the data, which in this case calculates the difference between the value for GDP (Y) in the two dataframes, expressed as a percent of the `basedf` dataframe. The formula used is equivalent to $\left(\frac{y^{lastdf} - y^{basedf}}{y^{basedf}}\right) * 100$

```
with pd.option_context('display.float_format', '{:,.8f}'.format):
    display((msim.lastdf['Y']/msim.basedf['Y']-1)*100)
```

```
2000   0.00000000
2001   0.00000000
2002   0.00000000
2003   0.00000000
2004   0.00000000
2005   0.00000000
2006   0.00000000
2007   0.00000000
2008   0.00000000
2009   0.00000000
2010   0.00000000
2011   0.00000000
2012   0.00000000
2013   0.00000000
2014   0.00000000
2015   0.00000000
2016   0.00000000
2017   0.00000000
2018   0.00000000
```

(continues on next page)

(continued from previous page)

```
2019    0.00000000
2020    0.00000000
2021    0.00000000
2022    0.00000000
2023    0.00000000
2024    0.00000000
2025    0.00000000
2026    0.00000000
2027    0.00000000
2028    0.00000000
2029    0.00000000
2030    0.00000000
Name: Y, dtype: float64
```

Important: The model has returned the same values as we input. This is very important because it implies the model passed the test that it reproduces history and in this case the forecast result when no changes are made to the model.

As we run more meaningful simulations below we can explore some of the data visualizations built into modelflow, which includes the mathplotlib and pandas functions as well as `modelflow` specific extensions to them.

9.6 Simulating the model

To perform a simulation we must change one of the variables in the model. As seen above, and in compliance with basic mathematics, if we change none of the model inputs and solve its system of equations it will always return the same result.

There are several ways that a model can be shocked.

- Shock an exogenous variable
- Exogenize a behavioural equation and shock it
- Shock the Add-factor of a behavioural equation

Below we will do each in turn, using the simple model. The objective here is to understand the mechanisms at play, and the steps necessary to perform each kind of simulation.

9.6.1 Shock an exogenous variable

In the model we have only two exogenous variables X (Exports) and YDISC (the statistical discrepancy).

To illustrate how to perform a simulation, lets assume that Demand for our countries exports increase by 10% between 2024 and 2026 and the return to their earlier level.

To do this we will need to change the values of exports and solve the model with the new values.

A simple way to do this would be to revise the value of X for the years 2024, 2025, 2026 by 10 percent. Pandas offers many ways to change the values of cells in a dataframe, we will do it in a `modelflow` way using the method `.mfcalc()` which allows us among other things to revise a the value of a variable. In this case we multiply the existing value of X in the initial dataframe by 1.1 or increasing it by 10%.

```
XShockdf=init.mfcalc("<2024 2026> X = X*1.1")

print((XShockdf['X']/init['X']-1)*100)
```

```
2000     0.0
2001     0.0
2002     0.0
2003     0.0
2004     0.0
2005     0.0
2006     0.0
2007     0.0
2008     0.0
2009     0.0
2010     0.0
2011     0.0
2012     0.0
2013     0.0
2014     0.0
2015     0.0
2016     0.0
2017     0.0
2018     0.0
2019     0.0
2020     0.0
2021     0.0
2022     0.0
2023     0.0
2024    10.0
2025    10.0
2026    10.0
2027     0.0
2028     0.0
2029     0.0
2030     0.0
Name: X, dtype: float64
```

To simulate the model using this new input, we can just submit this new revised dataframe in the same way we did the initial simulation.

Note: The results of a simulation are stored in the variable to the left of the call to the simulation, but are also automatically stored in an internal variable `.lastdf`, along with `.basedf` which contains the initial pre-shock dataframe.

Each time a simulation is run the value of `lastdf` gets overwritten with the results of the new simulation.

```
XShock_result = msim(XShockdf,2016,2030,silent=1,alfa=.5,ldumpvar=0)#ldumpvar saves␣
↪iterations 0 => don't;
                                                          #alfa <1 reduces step size␣
↪when iterating

# Use straight up pandas to display the results
with pd.option_context('display.float_format', '{:,.2f}'.format):
    display((msim.lastdf['Y']/msim.basedf['Y']-1)*100)
```

```
2000   0.00
2001   0.00
2002   0.00
2003   0.00
```

(continues on next page)

(continued from previous page)

```
2004    0.00
2005    0.00
2006    0.00
2007    0.00
2008    0.00
2009    0.00
2010    0.00
2011    0.00
2012    0.00
2013    0.00
2014    0.00
2015    0.00
2016    0.00
2017    0.00
2018    0.00
2019    0.00
2020    0.00
2021    0.00
2022    0.00
2023    0.00
2024    0.96
2025    1.05
2026    1.14
2027    0.29
2028    0.33
2029    0.36
2030    0.37
Name: Y, dtype: float64
```

In addition to the standard pandas features we have used to visualize data and simulation results, `modelflow` also has some built in methods for displaying results.

9.7 Text-based modelflow methods for displaying simulation results

Below are some `modelflow` specific methods for displaying results.

Method	Example	Short Name	Explanation
.dif	msim['Y'].dif.df	Shock-control (level)	The difference in the levels between `.lastdf` and `.basedf` $X^{lastdf} - X^{basedf}$
.difpct	msim['Y'].difpct.df	Change in growth rates	Difference between the growth rate of selected variables in the `.lastdf` dataframe vs the `.basedf` dataframe $(\dot{X}^{lastdf} - \dot{X}^{basedf})$
.mul100	msim['Y'].difpct.mul100.df	Multiplies result by 100	
.difpct.mul100	msim['Y'].difpct.mul100.df	Change in growth rates * 100	Difference between the growth rate (multiplied by 100) of selected variables in the `.lastdf` dataframe vs the `.basedf` dataframe $(\dot{X}^{lastdf} - \dot{X}^{basedf}) * 100$
.pctdiflevel	msim['Y'].pctdiflevel.mul100.df	Shock-control (% of baseline)	The change in the level of the variable divided by the level in the `.basedf` multiplied by 100 $\left(\frac{X^{lastdf} - X^{basedf}}{X^{basedf}}\right) * 100$

Note: The `msim.smpl(2020,2030)` restricts the period over which following modelflow commands operate. Here it limits the display of data to the period 2020 through 2030.

9.7.1 .dif The difference in levels between solutions

The `.dif` method shows the difference in the levels between two simulations $X^{lastdf} - X^{basedf}$.

```
msim.smpl(2020,2030)
msim['Y CPV'].dif.df

##This is equivalent to standard pandas (except here we have restricted the display␣
↪period to 2020 2030)
#print((msim.lastdf['Y']-msim.basedf['Y'])*100)
```

	Y	CPV
2020	0.000181	0.000060
2021	0.000197	0.000084
2022	0.000246	0.000106
2023	0.000297	0.000131
2024	318402.160251	5025.431813
2025	365062.261995	71302.371920
2026	419555.064658	130418.333366
2027	113193.218871	180026.429613
2028	134016.502259	157209.057757
2029	153479.554069	144823.692626
2030	168433.334917	139800.653296

9.7.2 .difpct the difference between the growth rates from the pre-shock and post-shock database

In this case `msim['Y'].difpct.df` prints the growth rate from the lastdf dataframe less the growth rate from the basedf dataframe.

$$(\dot{X}^{lastdf} - \dot{X}^{basedf})$$

Adding the function `.mul100` multiplies the result by 100.

Thus `msim['Y'].difpct.mul100.df` returns

$$(\dot{X}^{lastdf} - \dot{X}^{basedf}) * 100$$

This is precisely equivalent to the this pure pandas command `print((msim.lastdf['Y'].pct_change()-msim.basedf['Y'].pct_change())*100)`.

Because `msim['Y'].difpct.mul100.df` is a `modelflow` extension to pandas it will respect the sample period set by any earlier `.smpl(Begin,Year)` statement, whereas teh pure pandas version would display all of the data.

```
msim['Y'].difpct.mul100.df
```

```
                  Y
2020  1.245448e-10
2021  1.230127e-11
2022  1.350253e-10
2023  1.272538e-10
2024  1.013413e+00
2025  8.912041e-02
2026  9.972999e-02
2027 -8.856609e-01
2028  3.820072e-02
2029  3.020152e-02
2030  1.576184e-02
```

9.7.3 .difpctlevel - the percent change in the level of the variable.

In this case `msim['Y'].difpct.mul100.df` returns the percent change in the level of the variable Y.

Mathematically it is $\left(\frac{X^{lastdf}}{X^{basedf}} - 1\right) * 100$

Or as modelers often call it the impulse response function following a shock.

```
msim['Y CPBV'].difpctlevel.mul100.df

#print((msim.lastdf['Y']/msim.basedf['Y']-1)*100)
```

```
                  Y
2020  6.834967e-10
2021  6.950156e-10
2022  8.232579e-10
2023  9.441284e-10
2024  9.624021e-01
2025  1.047847e+00
2026  1.143541e+00
2027  2.929525e-01
2028  3.293305e-01
2029  3.581004e-01
2030  3.731188e-01
```

9.8 Graphics-based modelflow visualization methods

Instead of adding .df at the end of a comparison command, one can add plot to send the results to a graph. The results of the calculation and the impact of the sample period commands are the same.

Thus to view a graph of the level difference

IB Why are these not rendering in the book?

```
pd.options.display.float_format = '{:.1f}'.format # set the decimal points of the axis


msim['Y'].dif.plot(kind='line',title='Real GDP -10 %  hike in exports',colrow=1,top=0.
↪5)
```

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

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

9.8.1 Change in the growth rates

```
msim['Y'].difpct.mul100.plot(kind='line',title='Real GDP - pct change in growth rate',
↪colrow=1,top=0.8)
```

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

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

```
msim['Y'].difpctlevel.mul100.plot(kind='line',title='Real GDP (Pct change from␣
↪baseline)',colrow=1,top=0.8)
```

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

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

This time with multiple charts drawn from a single command

```
msim['Y CPV'].difpctlevel.mul100.plot(kind='line',title='GDP and Consumption (pct␣
↪deviation from baseline)',colrow=1,top=0.8)
```

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

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

9.9 Interactive comparisons of results

When working in jupyter books any of the above commands absent the .df or .plot will generate a widget that displays all of these results both as tables and graphs in different tabs.

```
msim['Y CPV'].difpct
```

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

Part IV

Using modelflow with World Bank models

CHAPTER

TEN

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

CHAPTER ELEVEN

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 **:cite:pt:`burns_world_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 **:cite:pt:`burns_climate_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.

CHAPTER

TWELVE

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
import pickle        # Used to store and retrieve model information


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

%load_ext autoreload
%autoreload 2
```

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

CHAPTER

THIRTEEN

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

13.1 Load a pre-existing model, data and descriptions

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

The command below

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

instantiates (creates an instance of) a 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 `baseline`. The model is solved with the parameter alfa set to 0.7. The $alfa \in (0, 1)$ parameter determines the step size of the solution engine. The larger alfa the larger the step size. Larger step sizes solve faster, but may have trouble finding a unique solution. Smaller step sizes take longer to solve but are more likely to find a unique solution. Values of alfa=.7 work well for World Bank models.

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

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

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

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.

13.2 Variables in World Bank models

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).

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, i.e. NY means National Income Accounts (see below for others)
MMM	The three-letter major sub-category of the data - i.e. GDP, EXP - expenditure
NNNN	The 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 Accounting systems include

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 from the goods and services accounts in current USD

13.3 Extract a list of variables

To extract a list of all variables matching a pattern, we can use the names function. Below we ask for a list of all variables for **PAK**istan **N**ational **E**xpenditure accounts **CON**sumption **X**price deflators **N** in local currency.

Note: **Wildcards** The `*` in the command `mpak['PAKNECON*XN'].names` is a `wildcard` character and the expression 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.

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

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

13.3.1 Additional variable information functions

Variable information methods and syntax

Command	Returns
model-name['varname'].des	Dictionary of mnemonic and variable description
model-name['varname'].desc	List of variable description alone
Wildcards	Search on mnemonics * matches multiple characters ? matches just one character
model-name.['*partialname*'].des	Returns Dictionary of all mnemonic and variable descriptions whose mnemonic matches
model-name.['*partialname'].desc	Returns list of variable descriptions whose mnemonic matches
model-name.['*partialname'].des	Returns Dictionary of all mnemonic and variable descriptions whose mnemonic matches
model-name.['*partialname'].names	Returns list of variable mnemonics that match
!Operator	**!** Search on description – Exclamation mark causes `modelflow` to search on the description instead of the mnemonic
model-name.['!*GDP*'].des	Returns Dictionary of all mnemonic and variable descriptions whose description contains the string GDP
model-name.['!*GDP*'].desc	Returns list of variable descriptions whose description contains the string GDP
model-name.['!*GDP*'].des	Returns Dictionary of all mnemonic and variable descriptions contains the string GDP
model-name.['!*GDP*'].names	Returns list of variable mnemonics whose description contains the string GDP
#Operator	Returns variable info from list
model-name.['#MyList'].des	Returns Dictionary of all mnemonic and variable descriptions of the variables contained in the list MyList
model-name.['#MyList'].desc	Returns list of variable descriptions whose description of the variables contained in the list MyList
model-name.['#MyList"].des	Returns Dictionary of all mnemonic and variable descriptions of the variables contained in the list MyList
model-name.['#MyList'].names	Returns list of variable mnemonics whose description of the variables contained in the list MyList

```python
import fnmatch

def match_desc(model,s2Match):
    reverse_des  = {v:k for k,v in model.var_description.items()}
    list_des = fnmatch.filter(reverse_des.keys(),s2Match)
    list_var = [reverse_des[v] for v in list_des]
    Results = {}
    for key, val in zip(list_var,list_des):
        Results.setdefault(key,val)
    return Results

def match_mnem(model,s2Match):
    model.var_description.items()
    list_des = fnmatch.filter(model.var_description.keys(),s2Match)
    list_var = [model.var_description[v] for v in list_des]
    Results = {}
    for key, val in zip(list_var,list_des):
        Results.setdefault(key,val)
    return Results
```

(continues on next page)

(continued from previous page)

```
desc=match_desc(mpak,"*GDP*")
mnems=match_mnem(mpak,"PAKNYGDPMKTP*N")

#mpak['!*GDP*'].des #returns the des of vars whose description match the string
#mpak['#listname'].des#the descriptions of teh variables in th list
#mpak['PAKNYGDPMKTP*N'].des  #the descriptions of the mnemonics that matchg

#mpak['!*GDP*'].desc#returns the des of vars whose description match the string
```

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

```
['PAKNECONPRVTCN', 'PAKNECONPRVTKN', 'PAKNECONPRVTXN']
```

CHAPTER

FOURTEEN

BEHAVIOURAL EQUATIONS IN THE MFMOD FRAMEWORK

Recall a behavioural equation determine 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.

14.1 The ECM specification

The ECM approach addresses the above challenge by modelling both the long run relationship and the short run short run behaviour and bringing 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.

14.1.1 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.

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

14.1.2 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:

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

14.2 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

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

14.3 Putting it together

From before we have the short run equation:

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

Inserting our 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 we are 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 we are away from equilibrium).

14.3.1 Lamda, the speed of adjustment

We can then interpret the parameter \$$\lambda asthespeedofadjustment.Aslongasλisgreaterthanzeroandlessorequaltooneifthere$

To be convergent λ must be between 0 and 1, if its is negative or greater than one, then the long run portion of the equation will cause the disequilibrium to grow each period ($\lambda > 1) notdiminishoroscillatefrompositivetonegative($\lambda <0\$).

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

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

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

Equation information methods

There are several functions to extract the equations from a model. The two most interesting are:

Command	Effect
`mpak['PAKNECONPRVTKN'].frml`	Returns the **normalized** version of the equation (the one actually used in modelflow)
`mpak['PAKNECONPRVTKN'].eviews`	In World Bank models imported from Eviews, reports the original eviews specification
`mpak['PAKNECONPRVTKN'].original`	Returns an intermediate version of the unnormalized equation, that replaces EViews syntax with Business Logic syntax

If we look at the equation for consumption in `mpak` we see that it follows something very close to this formulation.

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

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

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

```
'Real GDP'
```

Remember the `.frml` method presents the economic equation in a normalized form.

(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+0.0163839011059956DURING_2010-0.3*(PAKFMLBLPOLYXN/100-((LOG(PAKNECONPRVTXN))-(LOG(PAKNECONPRVTXN(-1)))))))) * (1-PAKNECONPRVTKN_D)+ PAKNECONPRVTKN_X*PAKNECONPRVTKN_D $

Taking logarithms of both sides of the the first expression (excluding the *(1-PAKNECONPRVTKN_D) term) and collecting the PAKNECONPRVTKNs onm teh left-hand side , we can recover the originally estimated ECM expression, where we simplify the mnemonics to ease reading of the equation using:

Model Mnemonic	Simplified	Meaning
PAKNECONPRVTKN	CON_t^{KN}	Household Consumption
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

$$\Delta log(CON_t^{KN}) = -0.2 * \Bigg[LOG(CON_{t-1}^{KN}) - LOG\Bigg(\frac{WAGEBILL_{t-1}^{CN} * (1 - DirectTxR_{t-1}/100)}{CON_{t-1}^{XN}}\Bigg)\Bigg] + 1.0 * \Delta log\Bigg(WAGEB$$

$$+0.030 + 0.016 * D_t^{2010} - 0.3 * \Bigg(r_t^{policy}/100 - \Delta log(CON_t^{XN})\Bigg) - CON_t^{KN_AF}$$

Where in this instance the short-run elasticity of consumption to disposable income has been constrained to 1, and teh short run elasticitya of consumption to the real interest rate is 0.3.

The `mpak['PAKNECONPRVTKN'].eviews` command returns the equations in a slightly more legible form, where the $\Delta ln()$ expressions are written using eviews syntax as dlog().

```
## This command not yet released, un comment when available.
##mpak.PAKNECONPRVTKN.eviews
```

CHAPTER

FIFTEEN

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. As noted when, as was the case of the load, the model is solved without changing any inputs we would expect that the model will return exactly the same data as before. To test this for `mpak` we can compare the results from the simulation using the `basedf` and `lastdf` dataframes.

Below we are gratified to see that the percent difference between the variables in the two dataframes following a simulation where the inputs were not changes is zero.

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

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

15.1 Different kinds of simulations

The `modelflow` package allows us to do 4 different kinds of simulations:

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.

15.1.1 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
```

```
      WLDFCRUDE_PETRO      LASTDF   Dif
2000        28.229719   28.229719   0.0
2001        24.351825   24.351825   0.0
2002        24.927748   24.927748   0.0
2003        28.898903   28.898903   0.0
2004        37.733388   37.733388   0.0
2005        53.391025   53.391025   0.0
2006        64.288259   64.288259   0.0
2007        71.116559   71.116559   0.0
2008        96.990454   96.990454   0.0
2009        61.756922   61.756922   0.0
2010        79.040772   79.040772   0.0
2011       104.009398  104.009398   0.0
2012       105.009629  105.009629   0.0
2013       104.077497  104.077497   0.0
2014        96.235000   96.235000   0.0
2015        50.752778   50.752778   0.0
2016        42.811667   42.811667   0.0
2017        52.805000   52.805000   0.0
2018        56.070279   56.070279   0.0
2019        59.537471   59.537471   0.0
2020        63.219063   63.219063   0.0
2021        67.128311   67.128311   0.0
2022        71.279294   71.279294   0.0
2023        75.686960   75.686960   0.0
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
```

15.1.2 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

Here we confirm that the shock we wanted to introduce was executed. The dif.df method returns the difference between the selected variable(s) as a dataframe, the `smpl method` restructs the time period of over which subsequent commands are effectuated.

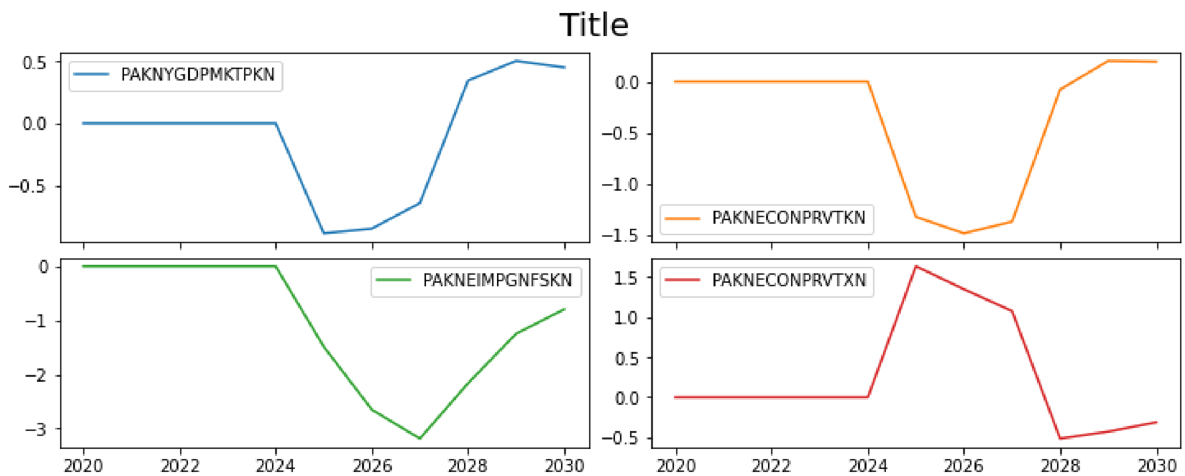
```
mpak.smpl(2020,2030)
mpak['WLDFCRUDE_PETRO'].dif.df
```

```
      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`.

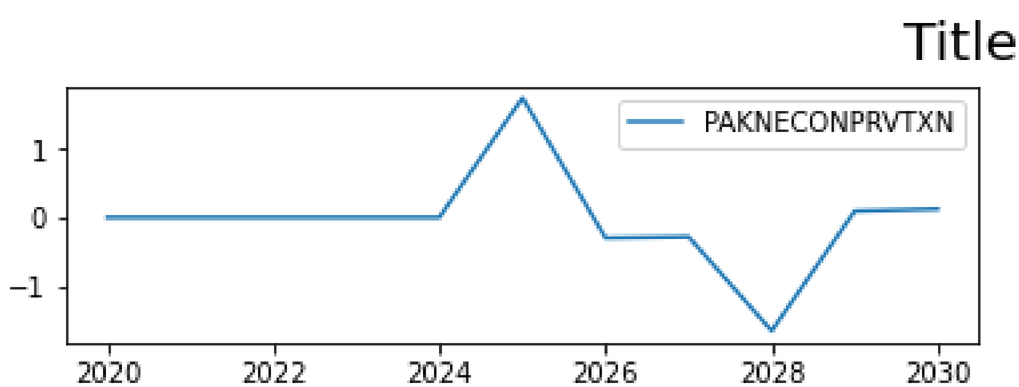
```
mpak['PAKNYGDPMKTPKN PAKNECONPRVTKN PAKNEIMPGNFSKN PAKNECONPRVTXN'].difpctlevel.
↪mul100.plot(Title="Impact of temporary $25 hike in oil prices")
```



The graphs show the change in the level as a percent of the previous level. The graphs suggest that a temporary $25 oil price hike would reduce GDP in the first year by about 1.5 percent, that the impact would diminish in the second year to about -.25 percent and that the impact would 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 they reduce the overall impact on 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 above shows what is happening to the **price level**. To see the impact on inflation (the rate of growth of prices) we will have to do a separate graph using `difpct.mul100`, which shows teh 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")
```



This view, gives a more nuanced result. Inflation spikes initially by about 1.2 percent, but falls below 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 fro lower prices soon translates into an acceleration in growth and higher inflation.

15.2 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 we assume that a change in weather patterns has increased the number of sunny days by 10 percent which has increased households happiness and therefore causes them to permanently increase their spending by 2.5% beginning in 2025.

We can do so either by manually or use the method `.fix()`. For simplicity we will use `.fix()` and we will explain the manual steps that `.fix()` does for us.

To exogenize `PAKNECONPRVTKN` for the entire simulation period we will first create a new dataframe as a slightly modified version of our basedf.

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

This does two things, that we could have 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 behavioural equation is effctively de-activated or exogenized.

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

Our assumption was that Real consumption would be 2.5% stronger.

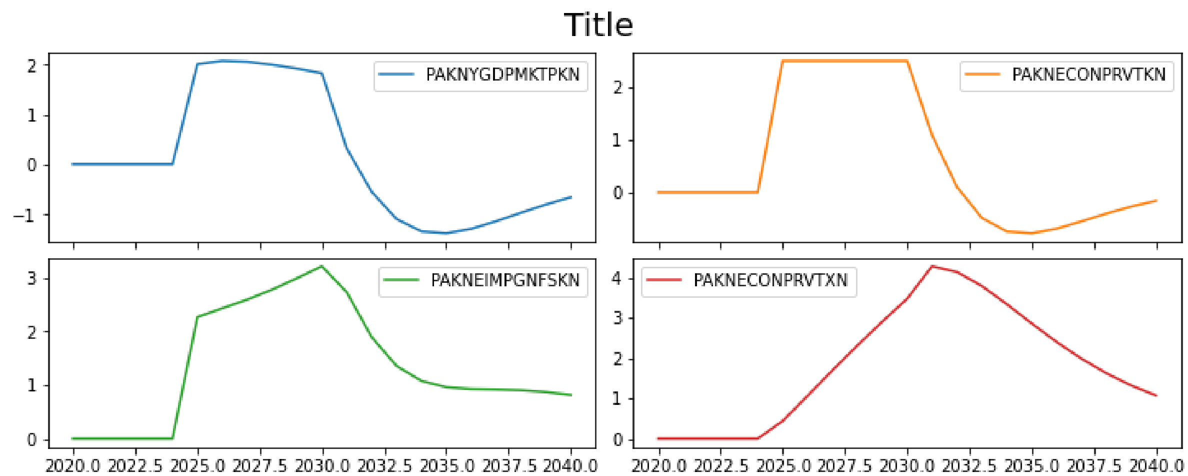
WE can change the value of PAKNECONPRVTYKN using the `.upd` method

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

Having made this change we can solve the model, by passing it the new CFixed dataframe.

`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 temporary $25 hike in oil prices")
```




```
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            0.32            1.09            2.73            4.27
2032           -0.55            0.11            1.89            4.13
2033           -1.10           -0.49            1.36            3.79
2034           -1.35           -0.74            1.08            3.33
2035           -1.39           -0.78            0.96            2.86
2036           -1.30           -0.69            0.92            2.40
2037           -1.14           -0.55            0.92            1.98
2038           -0.97           -0.40            0.90            1.62
2039           -0.81           -0.27            0.87            1.32
2040           -0.66           -0.16            0.81            1.07
```

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 zero by th end of the sample period.

15.3 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, we will do three scenarios.

1. We exogenize the variable for the whole period, but shock it for three years (2025-2027).
2. We exogenize the variable for the whole period, but shock it for three years (2025-2027)– but use the –kg option to keep the growth rates of the exogenized variable the same in the post-shock period.
3. We exogenize the variable only for the period during which we shock the dependent variable (2025-2027)

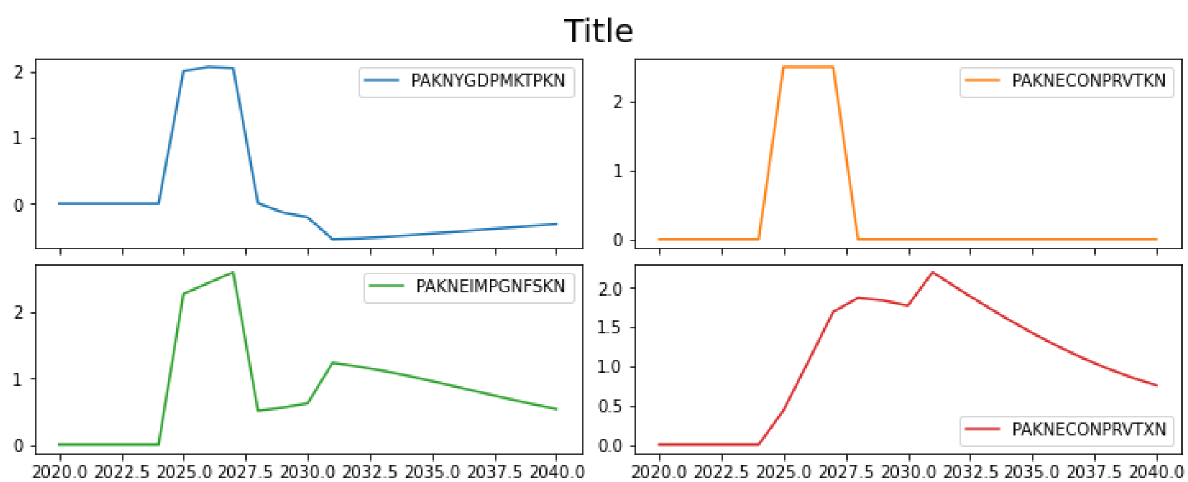
15.3.1 Temporary shock exogenized for the whole period

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(baseline,'PAKNECONPRVTKN')
CTempExogAll=CTempExogAll.upd("<2025 2027> PAKNECONPRVTKN_X * 1.025")

#Now we solve the model
CTempXAllRes = mpak(CTempExogAll,2020,2040,keep='2.5% increase in C 2025-27 -- exog␣
↪whiole period') # simulates the model
mpak['PAKNYGDPMKTPKN PAKNECONPRVTKN PAKNEIMPGNFSKN PAKNECONPRVTXN'].difpctlevel.
↪mul100.plot(Title="Impact of temporary $25 hike in oil prices")
```



Here 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).

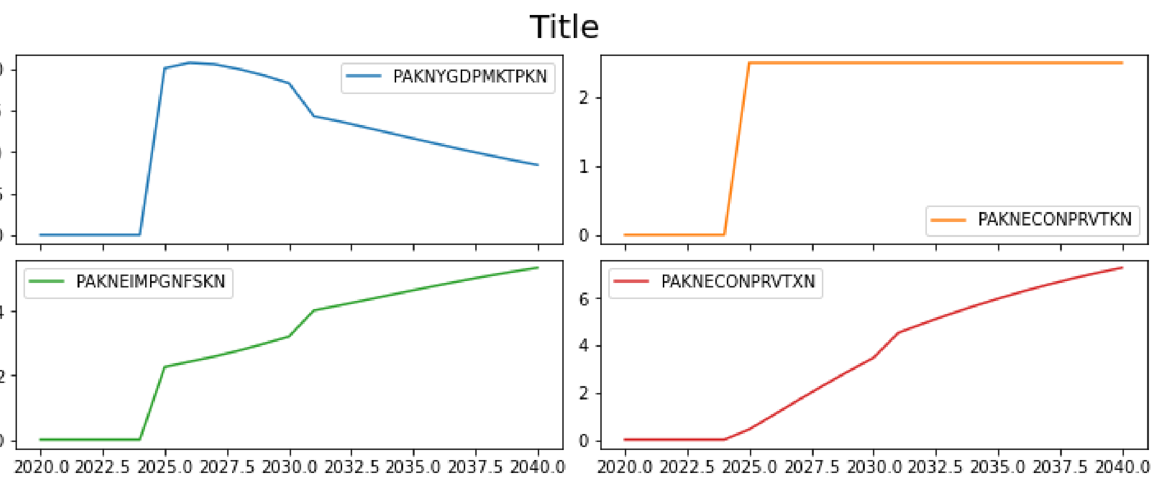
15.3.2 Temporary shock exogenized for the whole period

In this scenario we do exactly the same as in the previous but instead of mfcalc we use the upd command on the dataframe with the –KG (keep_growth) option to maintain the pre-shock growth rates of consumption in the post-shock period.

The set up is identical except replacing the mfcalc call with the upd call.

```
mpak.smpl() # reset the active sample period to the full model.
CTempExogAllKG=mpak.fix(baseline,'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 o cons 2025-27 --keep growth=true")
```



15.4 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.

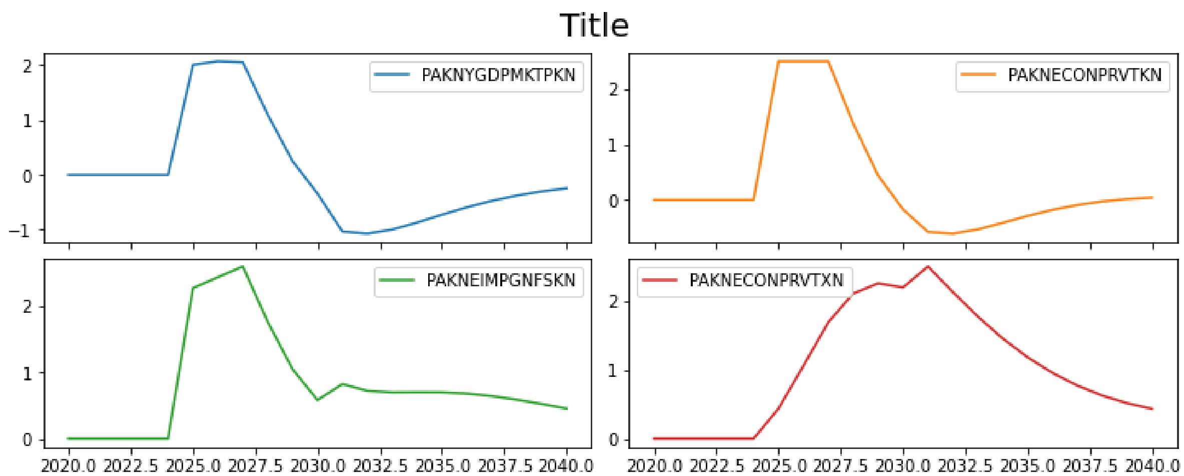
What this means is that the consumption function will be de-actvated for only three years (instead of the whole period as in the previous examples). As a result, the values consumption take in 2028, 2029, … 2040 will depend on the model, not the level it was set to when exogenized (which was the case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

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

#Now we solve the model
CExogTempRes = mpak(CExogTemp,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 o cons 2025-27 --keep growth=true")
```

(continues on next page)

(continued from previous page)



These results have subtle differences compared with the previous. The most obvious is visible in looking at the graph for Consumption. Rather than reverting immediately to its earlier pre-shock level, it falls more gradually and never falls all the way back 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(f'Value of GDP in 2028: {baseline.loc[2028,"PAKNYGDPMKTPCN"]:,.0f}')
print(f'Base value in 2028: {baseline.loc[2028,"PAKNECONPRVTKN"]:,.0f}. Alternative␣
↪value: {CExogTempRes.loc[2028,"PAKNECONPRVTKN"]:,.0f}.'
    f'Difference: {(CExogTempRes.loc[2028,"PAKNECONPRVTKN"]-baseline.loc[2028,
↪"PAKNECONPRVTKN"]):,.0f}.')
```

```
Value of GDP in 2028: 96,077,331
Base value in 2028: 27,241,278. Alternative value: 27,616,949.Difference: 375,671.
```

15.5 Access results

With each simulation we have stored the results in a unique dataframe. We can use our standard pandas routines and other python libraries like **Matplotlib** and **Plotly** to vusalize results.

Indeed in the preceiding apragraphs we have used these as well as some `modeflow` routines that extent the standard abilities of pandas.

15.6 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.

15.6.1 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.

Identify the functional form(s)

```
mpakinfo.
```

```
  Input In [22]
    mpakinfo.
             ^
SyntaxError: invalid syntax
```

Calculate the size of the required add factor shock

Run the shock

15.7 Using kept results to visualize results

With each of the simulations above we used the `keep=` option to store the results of the simulations within the model object.

In addition to the ability to store results in this way, modelflow includes methods to retrieve, display and compare results that were kept. This can be very useful when doing a number of similar simulations as was the case above.

For example the keep_plot routines will plot the value, growth rate or percent change in levels of different values across all of the kept solutions.

15.7.1 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', \
               savefig='testgraph/tax_impact_growth_.svg',legend=0);
```

15.7.2 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', \
               savefig='testgraph/tax_impact_pctdiflevel_.svg',legend=0);
```

15.8 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 will be 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
```

```
['PAKNECONPRVTKN_D']
```

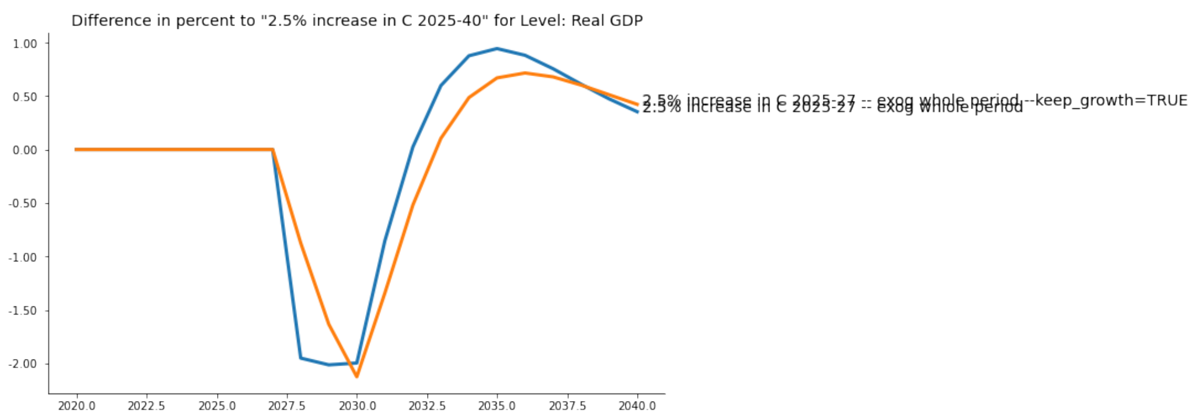
```
mpak['PAKNYGDPMKTPCN PAKNYGDPMKTPKN PAKGGEXPTOTLCN PAKGGREVTOTLCN PAKNECONGOVTKN']
```

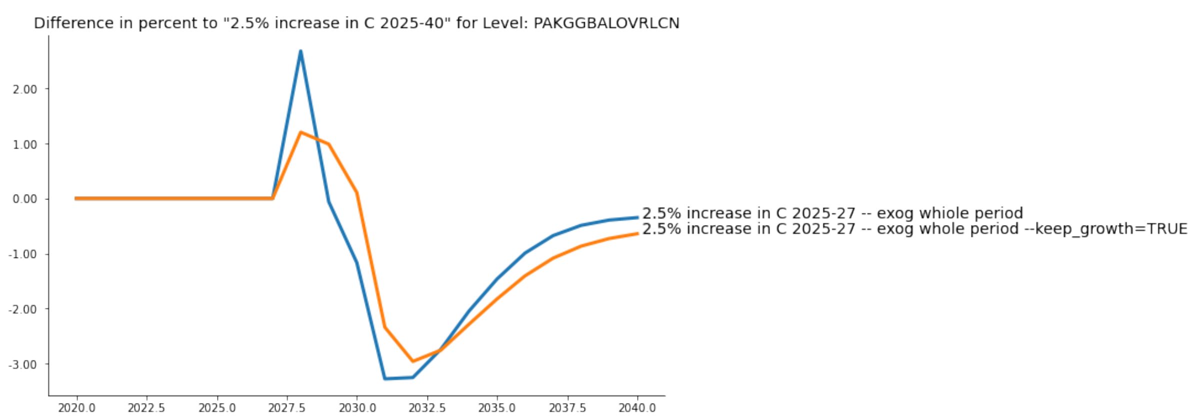
```
Tab(children=(Tab(children=(HTML(value='<?xml version="1.0" encoding="utf-8"␣
↪standalone="no"?>\n<!DOCTYPE svg …
```

```
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 whiole period', '2.5% increase in␣
↪C 2025-27 -- exog whole period --keep_growth=TRUE'])
```

```
with mpak.keepswitch(scenarios='2.5% increase in C 2025-40|2.5% increase in C 2025-27␣
↪-- exog whiole period|2.5% increase in C 2025-27 -- exog whole period --keep_
↪growth=TRUE'):
    mpak.keep_plot('PAKNYGDPMKTPKN PAKGGBALOVRLCN PAKGGDEBTTOTLCN',diffpct=1,showtype=
↪'level',legend=0);
```





CHAPTER SIXTEEN

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. As noted when, as was the case of the load, teh moedl is solved without changing any inputs we would expect that the model will return exactly the same data as before. To test this for the mpak we can use compare results from `basedf` and `lastdf` dataframes.

Below we are gratified to see that the percent difference between the variables is zero.

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

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

NameError: name 'mpak' is not defined
```

16.1 Different kinds of simulations

The `modelflow` package allows us to do 4 different kinds of simulations:

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.

16.1.1 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
```

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

Here we confirm that the shock we wanted to introduce was executed. The dif.df method returns the difference between the selected variable(s) as a dataframe, the `smpl method` restructs the time period of over which subsequent commands are effectuated.

```
mpak.smpl(2020,2030)
mpak['WLDFCRUDE_PETRO'].dif.df
```

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")
```

The graphs show the change in the level as a percent of the previous level. The graphs suggest that a temporary $25 oil price hike would reduce GDP in the first year by about 1.5 percent, that the impact would diminish in the second year to about -.25 percent and that the impact would 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 they reduce the overall impact on 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 above shows what is happening to the **price level**. To see the impact on inflation (the rate of growth of prices) we will have to do a separate graph using `difpct.mul100`, which shows teh 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")
```

This view, gives a more nuanced result. Inflation spikes initially by about 1.2 percent, but falls below 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 fro lower prices soon translates into an acceleration in growth and higher inflation.

16.2 A shock to a Behavioural variable

```
mpak.PAKGGREVGNFSCN.frml
```

The result of the equation can be fixes by calling mpak.fix(<dataframe>,PAKGGREVGNFSCN,2023,2023)

This will create a new dataframe where the value of PAKGGREVGNFSCN_X is set to the current value of PAKGGREVGNFSCN, and the value of PAKGGREVGNFSCN_D is set to 1 in the year 2023. When this dataframe is simulated the value of PAKGGREVGNFSCN will not depend on the ordinary right hand side variables, only on the value of PAKGGREVGNFSCN_X.

```
alternative_base = mpak.fix(baseline,'PAKGGREVGNFSCN',2023,2023)
```

Warning: In this experiment PAKGGREVGNFSCN is fixed in 2023. The value in all other years will be calculated using the original equation. To fix the value for all periods replace `2023,2023` with `2023,2100`

16.3 Create a scenario by shocking PAKGGREVGNFSCN

A new dataframe where PAKGGREVGNFSCN_X is increased by one percent of GDP is created

```
alternative = alternative_base.upd(f'<2023 2023> PAKGGREVGNFSCN_X + {baseline.
↪loc[2023,"PAKNYGDPMKTPCN"]*0.01 }')
```

The variable before and after the shock can be displayed

```
print(f'Value of GDP in 2023: {baseline.loc[2023,"PAKNYGDPMKTPCN"]:,.0f}')
print(f'Base value in 2023: {alternative_base.loc[2023,"PAKGGREVGNFSCN_X"]:,.0f}.␣
↪Alternative value: {alternative.loc[2023,"PAKGGREVGNFSCN_X"]:,.0f}.'
    f'Difference: {-(alternative_base.loc[2023,"PAKGGREVGNFSCN_X"]-alternative.
↪loc[2023,"PAKGGREVGNFSCN_X"]):,.0f}.')
```

16.4 Simulate the model

```
%matplotlib notebook
result = mpak(alternative,2020,2035,keep='Taxes on Goods and Services up by 1 pct of␣
↪GDP in 2023') # simulates the model
```

16.5 Access results

Now we have two dataframes with results `baseline` and `result`. These dataframes can be manipulated and visualized with the tools provided by the **pandas** library and other like **Matplotlib** and **Plotly**. However to make things easy the first and latest simulation result is also in the mpak object:

- **mpak.basedf**: Dataframe with the values for baseline
- **mpak.lastdf**: Dataframe with the values for alternative

The result can easily be visualized in Jupyter notebooks by using the `[.]` operator this will display the values of the variables in square brackets and useful transformations of the values including the impact. In addition the exotenous variables which has changed are displayed.

Click on the tabs to display the different output

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

16.6 Or use keep_plot to make more bespoken charts which can be saved in many formats

This method can display a number of different transformations of the series for more here Here only a few:

16.6.1 Differences of growth rates

```
mpak.keep_plot('PAKNYGDPMKTPCN PAKGGEXPTOTLCN',diff=1,showtype='growth',savefig=
↪'testgraph/tax_impact_growth_.svg',legend=0);
```

16.6.2 Differences in percent of baseline values

```
mpak.keep_plot('PAKNYGDPMKTPCN PAKGGEXPTOTLCN',diffpct=1,showtype='level',savefig=
↪'testgraph/tax_impact_difpct_.svg',legend=0);
```

16.7 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 will be 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.

```
!dir testgraph\
```

```
# fixed_alternative = mpak.fix(alternative,'PAKGGEXPCAPTCN PAKGGEXPGNFSCN␣
↪PAKGGEXPOTHRCN PAKGGEXPTRNSCN',2023,2035)
fixed_alternative = mpak.fix(alternative,'PAKGGEXPCAPTCN ',2023,2035)
```

```
result_fixed_expenditure = mpak(fixed_alternative,2020,2035,keep='Taxes on Goods and␣
↪Services up, expenditure fixed',silent=0,first_test=60) # simulates the model
```

```
mpak.fix_dummy_fixed
```

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

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

```
with mpak.keepswitch(scenarios='Taxes on Goods and Services up by 1 pct of GDP in␣
↪2023|Taxes on Goods and Services up, expenditure fixed'):
    mpak.keep_plot('PAKNYGDPMKTPCN PAKNYGDPMKTPKN PAKGGEXPTOTLCN PAKGGREVTOTLCN␣
↪PAKNECONGOVTKN',diff=1,showtype='level',legend=0);
```

Part V

References

CHAPTER

SEVENTEEN

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.