

The World Bank's MFMod Framework in Python with Modelflow

Andrew Burns and Ib Hansen

© 2025 International Bank for Reconstruction and Development / The World Bank
1818 H Street NW
Washington DC 20433
Telephone: 202-473-1000
Internet: www.worldbank.org

This work is a product of the staff of The World Bank with external contributions. The findings, interpretations, and conclusions expressed in this work do not necessarily reflect the views of The World Bank, its Board of Executive Directors, or the governments they represent.

The World Bank does not guarantee the accuracy, completeness, or currency of the data included in this work and does not assume responsibility for any errors, omissions, or discrepancies in the information, or liability with respect to the use of or failure to use the information, methods, processes, or conclusions set forth. The boundaries, colors, denominations, links/footnotes and other information shown in this work do not imply any judgment on the part of The World Bank concerning the legal status of any territory or the endorsement or acceptance of such boundaries. The citation of works authored by others does not mean the World Bank endorses the views expressed by those authors or the content of their works.

Nothing herein shall constitute or be construed or considered to be a limitation upon or waiver of the privileges and immunities of The World Bank, all of which are specifically reserved.

Rights and Permissions

The material in this work is subject to copyright. Because The World Bank encourages dissemination of its knowledge, this work may be reproduced, in whole or in part, for noncommercial purposes as long as full attribution to this work is given.

Any queries on rights and licenses, including subsidiary rights, should be addressed to World Bank Publications, The World Bank Group, 1818 H Street NW, Washington, DC 20433, USA; fax: 202-522-2625; e-mail: pubrights@worldbank.org.

Cover design: Marie-Anne Chambonniere

The World Bank's MFMod Framework in Python with Modelflow

Andrew Burns and Ib Hansen



Reproducible Research Repository

<https://reproducibility.worldbank.org>

A reproducibility package is available for this book
in the Reproducible Research Repository at:

[https://reproducibility.worldbank.org/index.php/catalog/324.](https://reproducibility.worldbank.org/index.php/catalog/324)

The World Bank's MFMod Framework in Python with Modelflow

Andrew Burns and Ib Hansen

Nov 15, 2025

CONTENTS

I The World Bank's MFMod Framework and Modelflow	3
1 Introduction	5
1.1 The MFMod Framework at the World Bank	6
1.2 Early steps to bring the MFMod system to the broader economics community	6
1.3 Moving the framework to an open-source footing	7
2 Macrostructural models	8
2.1 A system of equations	9
2.2 The MFMod Framework	10
3 Installation	13
3.1 Installation of Python	14
3.2 Installation of ModelFlow	15
3.3 Updating ModelFlow	18
II Some python essentials for using World Bank models with modelflow	19
4 Introduction to Jupyter Notebook	21
4.1 Starting Jupyter Notebook	22
4.2 Creating a notebook	23
4.3 Jupyter Notebook cells	24
5 Some Python basics	30
5.1 Starting python in windows	31
5.2 Python packages, libraries and classes	32
5.3 Importing packages, libraries, modules and classes	33
6 Introduction to Pandas, Series and dataframes	35
6.1 Import the pandas library	36
6.2 The Series class in Pandas	36
6.3 The DataFrame class in Pandas	39
6.4 Selected pandas methods	42
6.5 The .loc[] method	44
III Selected modelflow extensions to pandas	47
7 ModelFlow and Pandas DataFrames	49
7.1 Column names in ModelFlow	50
7.2 .index and time dimensions in ModelFlow	50

7.3	Leads and lags	51
7.4	The <code>.upd()</code> method returns a <code>DataFrame</code> with updated variables.	51
7.5	The <code>.mfcalc()</code> method. Return a dataframe with transformed variables.	69
IV	Using modelflow with World Bank models	75
8	Using ModelFlow with World Bank models	77
8.1	Publicly available World Bank models for use with ModelFlow	77
8.2	How to access the models the <code>.Worldbank_Models()</code> method	78
8.3	<code>.display_toc()</code> method	81
9	Working with a World Bank Model under ModelFlow	83
9.1	Prepare the work space	83
9.2	Load the model: Load a pre-existing model, data and descriptions	84
9.3	Groups	89
9.4	Information about equations	93
10	Equations in MFMod and ModelFlow	99
10.1	A behavioral equation	99
10.2	The add factor in behavioral equations	100
10.3	Excluding behavioral equations	100
10.4	Behavioral equations in ModelFlow	101
10.5	The ECM specification	102
11	Scenario analysis	107
11.1	Prepare the python session	107
11.2	Different kinds of simulations	109
11.3	The results visualization widget view	124
11.4	Save simulation results to a pcim file for later exploration	125
12	More complex scenarios	126
12.1	Load a pre-existing model, data and descriptions	126
12.2	The policy problem	127
12.3	Add variable descriptions	127
12.4	Simulating the impact of imposing a carbon price	128
12.5	Re-thinking the shock as an ex-ante real shock	131
12.6	Changing the model – modifying and or adding equations	132
13	A simulation that targets a specific outcome	138
13.1	Targeting in ModelFlow	138
13.2	Load a model, data and descriptions	139
13.3	Solve the model to create a baseline	139
13.4	Target CO2 emission - a very simple example	140
13.5	Targeting carbon emissions in a budget neutral manner	148
13.6	Define instruments	150
13.7	Solve the two-target targeting problem:	151
13.8	Weighting the instruments	153
13.9	Tuning the target input to get a result	156
13.10	Definition of Instruments	157
14	Report writing and scenario results	159
14.1	Preparing a ModelFlow Python environment	160
14.2	The <code>.table()</code> class	161
14.3	Complex tables	171

14.4	The <code>.plot()</code> class	174
14.5	The <code>.text()</code> class	187
14.6	Reports	190
14.7	Storing <code>.reports</code> definitions for later use	195
14.8	<code>[] .rtable</code> and <code>[] .rplot</code> - reports from <code>[]</code>	200
14.9	Some other supported outputs	201
V	Model Analytics	202
15	Model structure and causal chains	204
15.1	Setting up the python environment and loading a pre-existing model	204
15.2	Model information	205
15.3	Model structure	206
15.4	The dependencies of individual endogenous variables (the <code>.tracepre()</code> method)	207
16	Analyzing the impact of a shock	217
16.1	Load the existing model, data and descriptions	218
16.2	The mathematics of decomposition	218
16.3	Model-level decomposition or single equation decomposition?	219
16.4	Decomposing the source of changes to a single endogenous variable	219
16.5	A more complex example	223
16.6	The <code>get_att()</code> method provides more control over the outputs of <code>.dekomp()</code>	227
16.7	Trace and decomposition combined	236
16.8	Tabular output from <code>tracepre()</code>	237
16.9	Chart of the contributions over time	239
16.10	Chart of the contributions for one year	239
16.11	Sorted waterfall of contributions	241
16.12	Impacts at the model level: the <code>.totdif()</code> method	242
16.13	More advanced model attribution	250
VI	Technical how tos	255
17	Getting Help	257
17.1	Tutorials on:	257
17.2	Help on ModelFlow	258
18	Modelflow methods reference	262
18.1	Useful Jupyter Notebook commands and features	263
18.2	Working with the Model Object	263
18.3	Selected model properties	263
18.4	Selected Model methods	264
18.5	Equations	266
18.6	Visualize equations	269
18.7	Adding, Modifying, or Deleting Equations	270
18.8	Manipulating DataFrames	271
18.9	Performing Simulations	273
18.10	Explicit simulation methods	274
18.11	Modifying models	278
18.12	Saving results for comparison	278
18.13	Model Analytics	280
18.14	Setting time frame	281
18.15	Reports	283
18.16	Using the index operator <code>.[]</code> to select and visualize variables.	289

18.17 .plot chart the selected and transformed variables	297
18.18 [].rtable and [].rplot - reports from []	300
18.19 Plotting inspiration	300
18.20 .draw() Graphical presentation of relationships between variables	303
18.21 Attribution/decomposition	306
18.22 Bespoken plots using matplotlib (or plotly -later) (should go to a separate plot book)	313
18.23 Plot four separate plots of multiple series in grid	313

VII Backmatter	315
-----------------------	------------

Bibliography	317
---------------------	------------

Index	318
--------------	------------

Foreword

Over the decades, the World Bank has invested heavily in the tools available to its country economists to analyze, forecast, and monitor economic activity.

The MFMod framework is the ever-evolving fruit of those efforts. The framework is at the heart of the Bank's main MFMod model, a system of 184 macro-structural models of nearly all the economies in the world, that is the work-horse tool used by World Bank economists. The data and equations of MFMod are updated regularly and used to produce the twice-annual compendium **Macro Poverty Outlook** (<https://www.worldbank.org/en/publication/macro-poverty-outlook>), which presents concise statements of the Bank's views on the major challenges, outlook, and forecasts for almost every developing country in the world.

The MFMod framework is also used to generate customized models for individual countries and for complex policy analysis. Most recently it has been used extensively in the World Bank's **Country Climate Development Reports** (<https://www.worldbank.org/en/publication/country-climate-development-reports>) to model the impact of climate change in developing economies, and is also being used to analyze long-term prospects for the Bank's forthcoming **Country growth and Jobs Reports** (<https://www.worldbank.org/en/topic/jobsandgrowth>) (CGJRs).

Making these models available to a broader audience has always been a major focus of the Bank's modeling team. For years, the Bank has created customized models for developing country clients, and then transferred the models to client Ministries, training staff on the use, maintenance and modifications of these models. **The World Bank's MFMod Framework in Python with Modelflow** takes that process one step further by introducing an open-source tool for solving the models and by making models available on-line to anyone wanting to work with them.

While this manual is destined for a relatively small audience of macroeconomic modellers, it is hoped that it will generate significant benefits both for the Bank, via feedback on the models, and for clients who will have, for the first time, access to the Bank's models in a costless form.

Manuela Francisco

Director Economic Policy

The World Bank

Acknowledgements

This book and the development of ModelFlow would not have been possible without the contributions of many individuals.

Special thanks to Jens Boldt, formerly of Danmarks Nationalbank, for his participation in creating the first version of ModelFlow, which was developed initially for the top-down stress testing of banks.

We would also like to extend our gratitude to the following reviewers for their invaluable contributions and insights:

- Thanh Bui
- Unnada Chewpreecha
- Freya Casie
- Jacob Gyntenberg
- Charl Jooste
- Lasse Tryde

Their expertise and feedback have significantly enhanced the quality of this manual and the ModelFlow library.

Many thanks also to the World Bank Reproducibility Team (Luis Eduardo San Martin; Nihaa Sajid; Ankriti Singh; and Kanika Shokeen), who painstakingly went through the book and all of the charts and tables to make sure that the results can be reproduced. In so doing they exposed several issues that have since been resolved, contributing to the overall quality and reliability of the manual.

Remaining errors are the sole responsibility of the authors.

The MFMod framework itself reflects the inputs of many, over many years. Major contributors include but are not restricted to: Andrew Burns; Thanh Bui; Benoit Campagne; Paola Castillo; Unnada Chewpreecha; Young Il Choi; Charl Jooste; Francis Dennig; Alex Haider; Chung Gu Lee; Monika Matyja; Florent McIsaac; Theo Janse van Rensburg; Heather Ruberl; David Stephan; and Baris Tercioglu.

Preparation of this manual and the development of the World Bank extensions to ModelFlow have benefited enormously from the generous financial support of the Climate Support Facility <https://www.worldbank.org/en/programs/climate-support-facility>, a multi-donor Trust Fund administered by the World Bank that supports developing countries achieve a green recovery from the effects of COVID-19, implement their NDCs and develop long-term climate strategies.

The findings, interpretations, and conclusions expressed herein are entirely those of the authors and do not necessarily represent the views of the World Bank, its Executive Directors, or the governments of the countries they represent.

All remaining errors are the responsibility of the authors.

Part I

The World Bank's MFMod Framework and Modelflow

CHAPTER
ONE

INTRODUCTION

This manual describes the implementation of the World Bank's MFMod Framework [Burns *et al.* (2019)] using the open source modeling package ModelFlow. (Hansen, 2023 <https://ibhansen.github.io/doc/index>).

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 build and develop Macro-structural models¹ for developing countries.

This manual exists in a variety of formats:

- a PDF version
- an html version (available [here](https://worldbank.github.io/MFMod-ModelFlow/index.html) (<https://worldbank.github.io/MFMod-ModelFlow/index.html>)). The html version of the manual includes live links to:
 - the PDF version
 - the Jupyter Notebooks that comprise this book
 - links to the Jupyter notebooks that comprise the book on google colab where they can be run using the colab infrastructure
 - links to World Bank [github site](https://github.com/worldbank/MFMod-ModelFlow) (<https://github.com/worldbank/MFMod-ModelFlow>) that hosts the repository for the manual and the models that have been released

Note access to individual Jupyter notebooks and collab versions of the notebooks are provided by the icons that appear at the top of each html page:



The first icon links to colab. The second to the github site. The third allows downloads of the Jupyter Notebook and PDFs of the Jupyter Notebook currently being viewed in the html document. The final two icons shift the html to fulls-screen mode and switch between dark and light screen modes.

¹ Economic modeling has a long tradition at the World Bank. Initial World Bank economic models were linear programming planning models Chenery (1971). These were followed with CGE models Dervis and Robinson (1982). Indeed, the popular modeling package GAMS (<https://www.gams.com/about/company/>), which is widely used to solve CGE and Linear Programming models, started out as a project at the World Bank in the 1970s Meeraus (1982). The RMSM-X model Addison (1989) was an effort to provide a simpler consistency framework for economic forecasting. Work on the macrostructural models that were precursors to MFMod began in the early 2000s and were used within the Bank beginning in 2006, although versions were not released to the public until the end of the decade.

1.1 The MFMod Framework at the World Bank

MFMod is the World Bank's work-horse macro-structural economic modeling framework. It exists both as a linked system of 184 country-specific models that can be solved independently or as a larger system (MFMod), and as a series of standalone customized models, known collectively as MFMod Standalones (MFModSA). These Standalone models have been developed from the central model to fit the specific needs of individual countries. Both the central and Standalone models were developed using the EViews modeling language, and are run in that environment directly 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 <https://isimulate.worldbank.org> 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 model. Since 2015, MFMod has been the Bank's main tool for forecasting and economic analysis, and is used by all of the Bank's country economists for the World Bank's twice annual forecasting exercise *The Macro Poverty Outlook* (<https://www.worldbank.org/en/publication/macro-poverty-outlook>).

The main documentation for MFMod are Burns *et al.* (2019) and Burns and Jooste (2019).

In this Chapter – Introduction

This chapter provides a high-level overview of the MFMod system and the motivation for releasing some of the models in an open-source format using the ModelFlow python package.

1.1.1 Climate-aware version of MFMod

Most recently, the Bank has extended the standard MFMod framework to incorporate the main features of climate change Burns *et al.* (2021)—both in terms of the impact of the economy on climate (principally through green-house gas emissions, like CO_2 , N_2O , CH_4 , ...) and the impact of the changing climate (higher temperatures, changes in rainfall quantity and variability, increased incidence of extreme weather) on the economy (agricultural output, labor productivity, physical damages due to extreme weather events, sea-level rises etc.). So called co-benefits from climate policy, such as pollution reduction, changes in informality, health and productivity effects are also incorporated into the MFMod CC models. For the moment Burns *et al.* (2021) is the most up to date documentation of the MFMod CC models, but the models have evolved substantially from the initial climate model described there.

As of June 2025, variants of the model initially described in Burns *et al.* (2021), have been developed for 70 countries and underpin the economic analysis contained in the majority of the World Bank's *Country Climate Development Reports* (<https://www.worldbank.org/en/publication/country-climate-development-reports>) (<https://www.worldbank.org/en/publication/country-climate-development-reports>).

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

Bank staff were quick to recognize that the models built for its own needs could be of use to the broader economics community. An initial project, isimulate, made several versions of this earlier model available for simulation on the isimulate platform (<https://isimulate.worldbank.org>) (<https://isimulate.worldbank.org>) in 2007, and these models continue to be available there. The isimulate platform continues to house early versions of the MFMod system. While the platform allows simulation of these and other models, it 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 100 as of June 2025) customized stand-alone models (collectively known as MFModSA - MacroFiscalModel StandAlones) have been built from the main model. Typically developed for a country-client (Ministry of Finance, Economy or Planning or Central Bank), these Standalones 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. These features frequently include: 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.

In addition to making customized models available to client governments, since 2013 the World Bank has conducted training and dissemination around these customized versions of MFMod, designed to train government officials in the use of these models, their maintenance, modification and revision.

1.3 Moving the framework to an open-source footing

Models in the MFMod family are normally built and simulated using [EViews](https://www.eviews.com) (<https://www.eviews.com>), a proprietary econometric and modeling 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 Danish Central Banks, who over the years has developed [ModelFlow](#) a generalized solution engine for economic models written in Python. 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](https://ibhansen.github.io/doc/index) (<https://ibhansen.github.io/doc/index>). Nor is this paper documentation for the MFMod system, such documentation can be found [here](#) Burns *et al.* (2019), Burns and Jooste (2019), Burns *et al.* (2021), and [here](#) Burns *et al.* (2021)) for the specific models described and worked with below.

MACROSTRUCTURAL MODELS

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

Typically, organizations, including the World Bank, use all of these tools, privileging one or the other for specific purposes. Macrostructural models like those that comprise the MFMod framework are widely used by Central Banks, Ministries of Finance; and professional forecasters to generate forecasts and to undertake policy analysis.

While macrostructural models fell out of favor with academic economists, they remain central tools in policy making and forecasting circles. In a series of discussions and papers Olivier Blanchard Blanchard (2018), former Chief Economist at the International Monetary Fund, concluded that academic economists are wrong to discard out-of-hand policy models such as macro-structural models. Those conclusions were reinforced in a recent collection of papers by leading academics Vines and Wills (2020) that argued that until a better framework could be developed, “policy-makers need to rely on structural economic models and the detailed econometric work which they embody” rather than the DSGE models favored by academics.

In this chapter - Macrostructural Models

This Chapter provides a high-level introduction to macro-structural models in general and the MFMod system in particular. It presents them as systems that balance theoretical rigor with practical utility, enabling comprehensive insights into economic systems. Macrostructural models are widely used by policymakers for scenario analysis and forecasting and MFmod is the main framework used by the World Bank for analyzing the economic progress and policies of developing economies.

It notes that these models are effectively a system of equations that describe the economic transactions (flow of funds) that occur in the economy, including those captured by the main economic accounting systems, including:

- GDP and its subcomponents from the National Accounts expenditure, production, and income accounts.
- Detailed government, monetary policy, and balance of payments accounts.
- General equilibrium flow of funds linking households, firms, government, and foreign sectors.

The chapter introduces key concepts like:

- **Equation Types:**
 - Identities: Accounting rules that always hold (e.g., $GDP = C + I + G + X - M$).
 - Behavioral Equations: Relationships based on economic theory, that are estimated econometrically and subject to error.

variable Types:

- – Exogenous Variables: Inputs determined outside the model, such as global oil prices.
- Endogenous Variables: Variables determined by equations

The chapter also provides a brief description of ModelFlow the python package used to run World Bank models in an open source context.

2.1 A system of equations

Mathematically, a macro-structural model is a system of equations comprised of two kinds of equations and three kinds of variables. Variables that are determined by an equation are classified by the type of equation that determines them, variables without an equation are deemed exogenous as they are determined outside of the model.

Models in the MFMod framework are comprised of:

- **Identities** variables: that are determined by an identity: an equation that is a well-defined accounting rule that always holds. The famous GDP formula $Y=C+I+G+(X-M)$ is one such identity, it indicates that GDP at market prices is **definitionally** equal to Consumption plus Investment plus Government spending plus Exports less Imports. The equation is an identity and the variable (Y in this instance) is also called an identity.
- **Behavioral** variables that are determined by equations that attempt to statistically summarize the economic (vs accounting) relationship between variables, where the structure of the statistical relationship is derived from economic theory, but the sensitivities of different causal variables is estimated from the data. Thus, the neo-classical equation that says Real Consumption is determined by households maximizing their utility through the consumption of goods and services subject to a budget constraint is a behavioral equation. Because these behavioral equations only explain part of the variation in the variable they seek to explain, and because 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: do not have equations and are not determined by the model. Typically they are set either by assumption or from data external to the model. For an individual country model, the exogenous variables might include the global price of crude oil because the level of activity of a small economy itself is unlikely to affect the world price of oil. Similarly, the rate of growth of GDP in other economies may be treated as an exogenous variable, important to determining exports in the modeled developing country unlikely to be affected by activity in the modeled country (small country assumption).

Note

What dictates whether a variable is exogenous or endogenous is not set in stone. A variable that is exogenous in one model may be endogenous in another. For example in a single-country model, GDP growth of other countries may be exogenous, but in a multi-country model the GDP growth of those other countries would likely be endogenous (determined by the model).

Mathematically, a system of equations can be expressed as below:

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

where y_t^1 is one of n endogenous variables and x_t^1 is one of k exogenous variables. To have a solution the system must have as many equations as there are unknown (endogenous variables).

Substituting the variable mnemonics Y,C,I,G,X,M for the y's and x's above, a simple macrostructural model can be written as as a system of 6 equations in 6 unknowns:

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

Where Y_t is an identity and C_t, I_t, G_t, X_t, M_t are behavioral variables and P_t, P_t^f (domestic and foreign prices, respectively) are exogenous in this simple model.

Such a system of equations can then be solved for the endogenous variables Y_t, C_t, I_t, G_t, X_t and M_t as a function of the exogenous variables in the system, notably as written above P_t and P_t^f , and the estimated parameters and functional forms of the behavioral equations that link them, represented here by: $c()$, $i()$, $g()$, $x()$ and $m()$.

2.2 The MFMod Framework

World Bank models are somewhat more complex and comprise many more sectors, notably:

- GDP and sub components calculated from all three perspectives in Real, Nominal and implicit deflator terms
 - Expenditure Accounts
 - * $C + I + G + X - M$
 - Production accounts (the level of detail varies from model to model)
 - * Primary sector (Agriculture, Mining, Forestry)
 - * Secondary Sector (Manufacturing, Industry)
 - * Tertiary sector (Services, retail, Public Administration, Wholesale)
 - * Energy sector (primarily broken out in climate models)
 - Income accounts (Wage Bill, Gross Operating surplus (Profits), Combined incomes)
 - Government Accounts
 - – Revenues
 - * Personal income taxes
 - * Corporate income taxes
 - * Value added taxes (Sales taxes)
 - * Excise Taxes
 - * Trade taxes (Export taxes, import duties)
 - * Other taxes, (Fees and charges)
 - * Grants and Transfers

Expenditures

- * Goods and services
 - * Wages and salaries
 - * Transfers to households
 - * Subsidies to households
 - * Subsidies to firms
 - * Capital expenditures (New projects and repairs to existing capital stock)
 - * Grants and Transfers
 - * Interest payments on the debt
- Balances
 - * Overall fiscal balance
 - * Primary fiscal balance
 - * Debt (Domestic, Foreign)

Monetary Policy

- - Main policy rate
 - Money Supplies
 - International Reserves
 - Credit to the private sector
- Balance of Payments
 - Current Account
 - * Primary Exports and Imports (Merchandise and Services)
 - * Secondary Exports and Services (Remittances, repatriation of profits, labor)

Financial Account

- * Equity financing
 - * Debt financing
 - * FDI

Within the models, these accounts are related in a general equilibrium flow of funds perspective, where households (as the owners of the factors of production) supply labor and capital to firms and the government via factor markets and earn salaries and profits paid for by the firms. Their earnings are spent on goods and services, taxes or saved, with their savings being intermediated through financial markets where they are either loaned to domestic firms, households and the government or to foreign firms and governments. Domestic households, firms and the government also lend to and borrow from the foreign sector. The output of firms is sold to households, the government or the rest of the world and the intermediate inputs they require are purchased from other firms, or the rest of the world.

Importantly every expenditure of a given actor in the economy is a revenue of another and, as a result, has impacts on the rest of the economy. This flow of funds idea is common to most macroeconomic models and is illustrated in the following schematic.

The Flow of Funds in MFMod

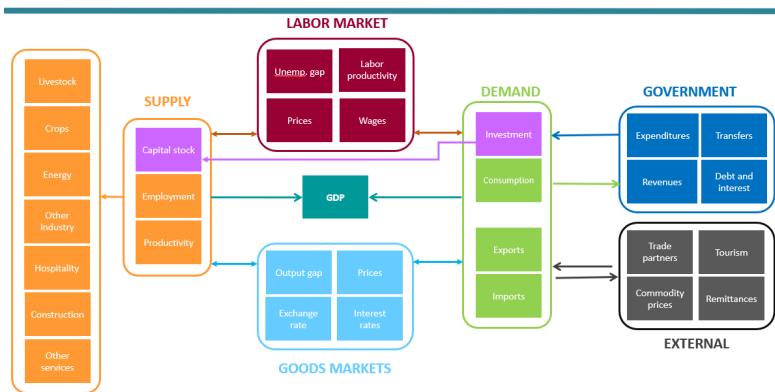


Fig. 2.1: The Flow of funds in MFMod

INSTALLATION

At the World Bank, models built using the MFMod framework are developed in EViews (<http://www.eviews.com>). When disseminated to clients, the models are solved and simulated using EViews - often through the intermediary of an easy-to-use customized excel environment developed by the World Bank. That said, as systems of equations and associated data, the models can be solved and operated under any software capable of solving a system of simultaneous equations. ModelFlow is such a package that permits not only solving the models, but also provides a rich and powerful suite of tools for analyzing the model and reporting results.

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 Central Banks into the 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.

ModelFlow 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 or MiniConda variants). Then install the ModelFlow package and several supporting packages.

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 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, was written and executed in the Jupyter Notebook environment using the [Jupyter Book](https://jupyterbook.org/) (<https://jupyterbook.org/>) package.

To use the ModelFlow package **First** Python has to be installed. **Then** the ModelFlow package can be installed together with all the libraries on which it depends.

In this chapter - Installation

This chapter provides step-by-step instructions for setting up the ModelFlow framework to work with World Bank macroeconomic models. It provides both novice and experienced python users with clear instructions on the steps necessary to install a reliable and reproducible ModelFlow environment for advanced modeling tasks.

Key covered topics include:

- **Python Installation:**
 - Using the Anaconda or MiniConda environments.
 - A discussion of the benefits of each notes that Anaconda offers a comprehensive package ecosystem (perhaps best suited for newcomers to python, while MiniConda is more lightweight and customizable and may be best suited for more experienced python users).

Installing the ModelFlow Environment:

- Create a dedicated Python environment for ModelFlow to avoid package conflicts.
- Install ModelFlow and its dependencies using the provided commands.

• System Compatibility:

- Supported on Windows, MacOS, and Linux.
- Installation is straightforward, with Windows being the primary platform tested.

Updating ModelFlow:

- Using the `conda` command to update ModelFlow packages within the created environment.

3.1 Installation of Python

Python is a powerful, versatile and extensible open-source programming language. It is widely used for artificial intelligence applications, interactive web sites, and scientific processing. As of May 2024, the Python Package Index (PyPI), the official repository for third-party Python software, contained over 530,000 packages that extend its functionality¹. ModelFlow is one of these packages.

Python comes in many flavors and ModelFlow will work with most of them. Nevertheless, users are **strongly advised** to use either the **Anaconda** distribution of Python or the closely related **Miniconda** distribution.

Anaconda is a full-featured distribution of python that comes with a comprehensive set of pre-installed packages and tools for scientific computing and data analysis. It is best suited for users who want a ready-to-use platform with a wide range of packages and don't mind the larger installation size.

Miniconda is a more streamlined distribution, providing only the essential components for running python. It offers faster installation and a smaller footprint, making it suitable for users who prefer a more streamlined environment and want more control over package selection. Features that are in Anaconda, but not installed by miniconda by default, can be added manually.

3.1.1 Which is better for me

In general, **Anaconda** is preferred by users who value convenience and desire a comprehensive package ecosystem. **Miniconda** is preferred by users who seek a minimalistic setup and have specific package requirements. If you are already familiar with python, plan to use it with ModelFlow and have space limitations Miniconda is probably the best solution for you. For users new to python Anaconda may be a better solution.

The processes for installing **Anaconda** and **Miniconda** are very similar. The main difference is which installer is downloaded and then where the distribution is stored in the file system.

It is possible to install both distributions on the same machine without them interfering with each other.

Both are available for the Windows, MacOS and Linux operating systems and ModelFlow should work equally well under all three operating systems. However, only the Windows version has been thoroughly tested and was used in producing this manual.

¹ Wikipedia article on python ([https://en.wikipedia.org/wiki/Python_\(programming_language\)](https://en.wikipedia.org/wiki/Python_(programming_language))).

3.1.2 Anaconda/MiniConda installation instructions

Windows

The definitive source for installing **Anaconda under windows** can be found [here](https://docs.anaconda.com/anaconda/install/windows/) (<https://docs.anaconda.com/anaconda/install/windows/>).

The definitive source for installing **MiniConda under windows** can be found [here](https://docs.conda.io/projects/miniconda/en/latest/miniconda-install.html) (<https://docs.conda.io/projects/miniconda/en/latest/miniconda-install.html>).

The version of ModelFlow and the examples in this manual were developed and tested using the 3.10 version of python.

Warning

It is strongly advised that Anaconda/Miniconda be installed for a single user (Just Me). This is much easier to maintain over time, especially in a professional environment where users may not have administrator rights on their computers. Installing “For all users on this computer” (the other option offered by the installers) will substantially increase the complexity of maintaining python on your computer.

MacOS

The definitive source for installing **Anaconda under macOS** can be found here: <https://docs.anaconda.com/anaconda/install/mac-os/>.

The definitive source for installing **MiniConda under macOS** can be found here: <https://docs.anaconda.com/free/miniconda/miniconda-other-installer-links/>.

Linux

The definitive source for installing **Anaconda under Linux** can be found here: <https://docs.anaconda.com/anaconda/install/linux/>.

The definitive source for installing **MiniConda under Linux** can be found here: <https://docs.conda.io/projects/conda/en/latest/user-guide/install/linux.html>.

3.2 Installation of ModelFlow

ModelFlow is a python package that defines the model class, its methods and a number of other functions that extend and combine pre-existing python functions to allow the easy solution of complex systems of equations including macro-structural models like MFMod. To work with ModelFlow, a user needs to first install python (preferably the Anaconda or MiniConda variants). Then install the ModelFlow package and several other supporting packages.

3.2.1 Creation of a ModelFlow environment

Although it is not strictly necessary, it is a good idea to create a specific python environment[^environ] for ModelFlow, this will be a space where the ModelFlow package itself will be installed and in which the specific dependencies on which ModelFlow relies will be installed. Other environments may require packages that conflict with ModelFlow. Declaring a separate environment for ModelFlow will help prevent conflicts between different versions of packages from arising.

Note

Both Anaconda and Miniconda support the idea of “environments”. Environments are ring-fenced areas on your computer that can have different versions of Python and/or packages installed in them. If one application requires a specific version of pandas to run, and a second application requires a different version, by housing them in separate environment the dependencies of each application can be respected without generating a conflict between them. See [here](https://conda.io/activation) (<https://conda.io/activation>) for more.

The commands below create a ModelFlow environment and install into it the ModelFlow package and several supporting packages that are useful when using ModelFlow in conjunction with EViews or Jupyter Notebook.

Note

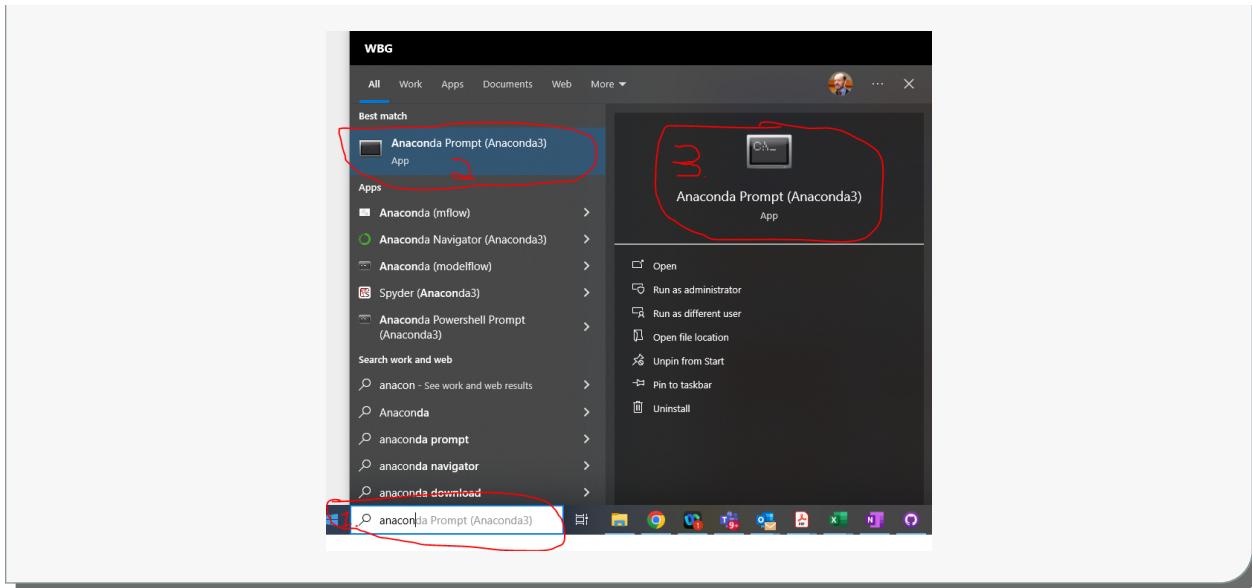
ModelFlow works equally well under both Jupyter Notebook 6 and Jupyter Notebook 7. However, it was developed using JN6 and this book and its examples were produced using JN6. The installation instructions below will set up your ModelFlow environment to use JN6

The commands should be cut and paste (either one by one, or as a block) into the Anaconda or Miniconda command prompt environment. Instructions for activating the environment in windows are included in the box.

Box 1. Opening the Anaconda/MiniConda prompt under windows

To open the Anaconda/MiniConda prompt:

1. In the windows command prompt, type Anaconda (or Miniconda)
2. If Anaconda/Miniconda have been successfully installed an icon entitled Anaconda(Miniconda) Prompt will be available in windows. Click on this.
3. This will open a python command shell where the commands listed in the main text can be entered.



```

conda create -n ModelFlow -c ibh -c conda-forge ModelFlow_book -y
conda activate ModelFlow
#####
# Note skip the next line if installing under linux or Macintosh
#####
conda install py2eviews -c eviews
#####
# Continue from here on all Operating Systems
#####
pip install dash_interactive_graphviz
jupyter contrib nbextension install --user
jupyter nbextension enable hide_input_all/main
jupyter nbextension enable splitcell/splitcell
jupyter nbextension enable toc2/main
jupyter nbextension enable varInspector/main
    
```

i Note

Meaning of the command lines

- The first line in the code snippet above simultaneously creates the ModelFlow environment (the `-n` argument names the environment) and installs the `ModelFlow_book` package from `conda-forge` into this environment.[NB:There also exists a `ModelFlow_stable` package. The book package will be frozen at its current state of evolution to ensure that all the examples in this book will run using `ModelFlow_book`, the stable version may evolve over time and potentially break specific examples. Longer-term users may wish to switch to the stable version, once they have absorbed the material in this book.]
- The second command “activates” the newly created ModelFlow environment.
- The subsequent commands install some additional packages into the ModelFlow environment that are used by ModelFlow.
 - The `py2eviews` package allows python to execute EViews commands on a machine where EViews has also been installed (this is only useful if the user intends to import existing EViews workfiles into the python environment of ModelFlow).
 - the `dash_interactive_graphviz` package enables a series of interactive dashboards which can

- be handy when using ModelFlow in Jupyter Notebooks.
- The last 5 commands activate some useful jupyter notebook extensions.

Depending on the speed of your computer and of your internet connection, installation could take as little as 5 minutes or more than 1/2 an hour.

3.3 Updating ModelFlow

ModelFlow gets new features and fixes these will be reflected in the `ModelFlow_stable` package. In order to use new features compared to the version on which this book is created the user might create a new environment to use this. This would have the advantage of preserving the current `ModelFlow` environment for recreating the book examples.

The command below creates a new environment `-n ModelFlow_new` and installs the most recent version of `ModelFlow` from the `modelflow_stable` repository. As noted above, this might introduce changes that break some of the examples in this book – although every effort will be made to maintain backward compatibility.

```
conda deactivate  
conda create -n ModelFlow_new -c ibh -c conda-forge modelflow_stable -y  
conda activate ModelFlow_new
```

And perform the additional commands specified in the previous section.

Part II

Some python essentials for using World Bank models with modelflow

INTRODUCTION TO JUPYTER NOTEBOOK

Jupyter Notebook is an application for creating, annotating, simulating and working with computational documents. Originally developed for python, the latest versions of EVViews 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 desired 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 the [Jupyter Book package](https://jupyterbook.org/en/stable/intro.html) (<https://jupyterbook.org/en/stable/intro.html>).

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 is a great environment in which to follow work done by others, try out code, and tweak the codes of others to fit your own needs.

In this chapter - Introduction to Jupyter Notebook

This chapter introduces Jupyter Notebook. It can be safely skipped by readers already familiar with Jupyter Notebook (an interactive environment for programming, documenting, and analyzing data).

Key points in the chapter include:

- **What is Jupyter Notebook:**

- A tool for creating and sharing computational documents that integrate code, text, and visualizations.
- Ideal for prototyping, debugging, and documenting workflows.

- **Getting Started:**

- Activate the Python environment and launch Jupyter Notebook from the command line.
- Navigate to your desired working directory and start a new notebook.

- **Notebook Structure:**

- Composed of cells that can contain code, markdown, or outputs.
- Cells have two modes: Edit (for writing code or text) and Command (for running cells or managing notebook structure).

- **Key Features:**

- Write and execute Python code interactively.
- Use Markdown for formatted text, including headers, bullet points, and inline mathematics.
- Render complex mathematical equations with LaTeX.

- **Shortcuts and Tips:**

- Use keyboard shortcuts to navigate, edit, and execute cells efficiently.
- Combine code, results, and documentation in one notebook for replicability.

There are many fine tutorials on Jupyter Notebook on the web, and The official Jupyter site (<https://docs.jupyter.org/en/latest/>) is a good starting point. Another good reference is here (<https://jupyter.brynmawr.edu/services/public/dblank/Jupyter%20Notebook%20Users%20Manual.ipynb>).

The remainder of this chapter aims to provide enough information to get a user started.

4.1 Starting Jupyter Notebook

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

- 1) Opening the Anaconda command prompt window (Windows Start->Type Anaconda->Select Anaconda Command Prompt)
- 2) Activate the ModelFlow environment we just created by executing the following command: `conda activate ModelFlow`
- 3) Navigate to a position on his/her computer's directory structure where they want to store (or where they are already stored) their Jupyter Notebooks. (e.g. `cd c:\users\MyUserName\MyJupyterNotebookDirectory`)
- 4) Finally, the Jupyter Notebook python program must be started by executing the following command from the conda command line:

```
jupyter notebook
```

This will launch the Jupyter environment in your default web browser, which should look something like the image below, where the directory structure presented is that of the directory from which the `jupyter notebook` command was executed (e.g. `c:\users\MyUserName\MyJupyterNotebookDirectory`).

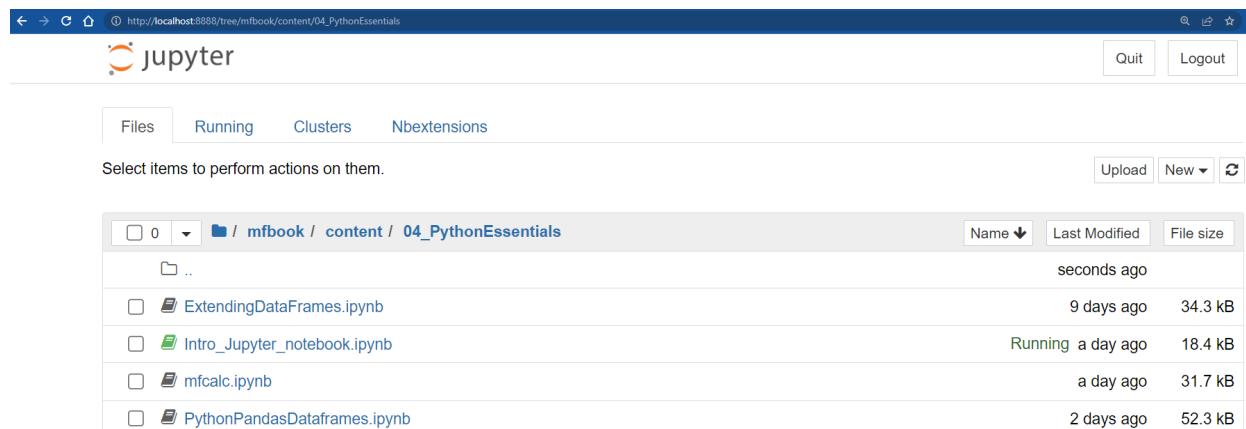


Fig. 4.1: The Jupyter File explorer view

⚠ Warning

Note the directory from which you execute the `jupyter notebook` will be the **root** directory for the jupyter session. **Only directories and files below this root directory will be accessible by Jupyter.**

4.2 Creating a notebook

The idea behind Jupyter Notebook is to create an interactive version of the physical notebooks that scientists use(d) to:

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

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

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

File-> New Notebook

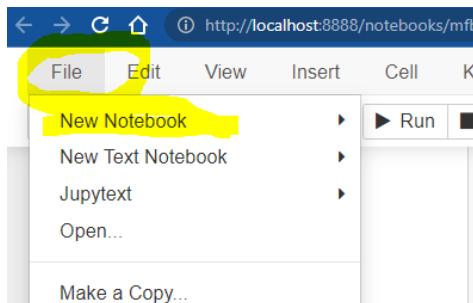


Fig. 4.2: A newly created Jupyter Notebook session

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

💡 Note

Each notebook has associated with it a “Kernel”, which is an instance of the computing environment in which code will be executed. For Jupyter Notebooks that work with ModelFlow this will be a python Kernel. If your computer has more than one “kernel” installed on it, you may be prompted when creating a new notebook for the kernel with which to associate it. Typically this should be the Python Kernel under which your ModelFlow was built – currently python 3.13 in June 2025.

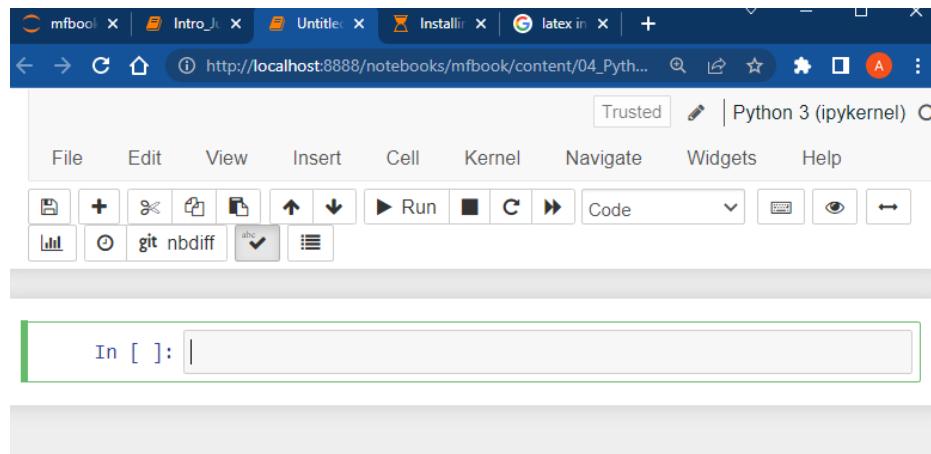


Fig. 4.3: A newly created Jupyter Notebook

4.3 Jupyter Notebook cells

A Jupyter Notebook is comprised of a series of cells.

Jupyter Notebook cells can contain:

- **computer code:** typically python code, but, as noted, other kernels – like EViews – can also be used with Jupyter.
- **markdown text:** plain text that can include special characters that make some text appear as bold, or indicate the text is a header, or instruct Jupyter to render the text as a mathematical formula. All of the text in this document was entered using Jupyter Notebook’s markdown language (more on this below).
- 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. Command mode – indicated by a blue vertical bar. This will be the 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) is displayed in an output cell that will be generated in the space immediately below the code cell that generated the output. Also the keyboard is mapped to actions which allows the user to perform tasks like selecting and copying on the cells.

Users can switch between Edit and Command Mode by hitting Esc (to Command mode) or Enter (to Edit mode). With the mouse a double-click in the cell content will switch to Edit Mode and a click in the cell margin will switch to the Command mode.

```{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.

### 4.3.1 How to add, delete and move cells

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

#### Using the Toolbar

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

#### Using keyboard short cuts

- **esc + a:** add a cell above the current cell
- **esc + b:** add a cell below the current cell
- **esc + d+d:** delete the currently selected cell(s)

### 4.3.2 Change the type of a cell

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

#### Using the Toolbar

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

#### Using keyboard short cuts

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

#### Auto-complete and context-sensitive help

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

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

### 4.3.3 Execution of cells

Every cell in a Jupyter Notebook can be executed. Executing a markdown cell will cause the cell’s content to be rendered as html. Executing a python code cell, will cause its content to be executed. Cells 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. The cursor will stay on the cell that was executed.
- **shift + enter:** Executes the code in the cell or formats the markdown of a cell. The focus (cursor) jumps to the next cell.

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

### Executing python code

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

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

45

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

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

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

Another way to display results is to use the print function. In this instance even if there is a semicolon at the end of the line, the result of the print command is still output.

```
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

The natural order is to execute them sequentially in the order they appear in the Jupyter Notebook. However, when debugging or developing code it may be useful to execute cells out of their natural order.

The persistence of data (the fact that a variable y defined in one cell can be used in another cell) depends on the order in which cells were executed, not the order they appear in the notebook.

#### 4.3.4 Markdown cells and the markdown scripting language in Jupyter Notebook

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

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

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

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

##### Common markdown commands

Some of the most common of these include:

| symbol         | Effect                       |
|----------------|------------------------------|
| #              | Header                       |
| ##             | second level                 |
| ###            | third level etc.             |
| **Bold text**  | <b>Bold text</b>             |
| *Italics text* | <i>Italics text</i>          |
| * text         | * Bulleted text or dot notes |
| 1. text        | 1. Numbered bullets          |

##### Tables in markdown

Tables like the one above can be constructed using the symbol | as a separator.

Below is the markdown code that generated the above table:

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

The |:-|:-| on the second line tells the Table generator how to justify the contents of columns. The \ before the #s above tells markdown to ignore the normal meaning of # and just show the symbol.

| Symbol | Meaning                                                                       |
|--------|-------------------------------------------------------------------------------|
| :--    | left justify                                                                  |
| :-:    | center justify                                                                |
| --:    | right justify.                                                                |
| \      | Literal operator. Display what follows do not interpret it in the normal way. |

### Displaying code

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

### Inline code blocks

For inline code references ‘ a single back tick at the beginning and end suffices. For example, the below line

An example sentence with some back-ticked `text as code` in the middle **will render as:**

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

### Multiline code block

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

The below block of code:

```

Multi line

text to be rendered as code

```

will render as:

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

### Rendering mathematics in markdown

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

Maths can be displayed either *inline* – surrounded by non mathematical text on the same line, or as a standalone equation.

Below is an example of inline mathematics, which was generated by surrounding the latex maths expression by the \$ sign:

The inline code `$y_t = \beta_0 + \beta_1 x_t + u_t$` will renders as:  $y_t = \beta_0 + \beta_1 x_t + u_t$   
if enclosed in `$$` the encapsulated javascript will render on its own line.

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

will be rendered centered on its own line – as here.

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

### Complex and multi-line math

Multiline expressions and equations can also be rendered:

```
\begin{aligned*}
Y_t &= C_t + I_t + G + t + (X_t - M_t) \\
C_t &= c(C_{t-1}, C_{t-2}, I_t, G_t, X_t, M_t, P_t) \\
I_t &= i(I_{t-1}, I_{t-2}, C_t, G_t, X_t, M_t, P_t) \\
G_t &= g(G_{t-1}, G_{t-2}, C_t, I_t, X_t, M_t, P_t) \\
X_t &= x(X_{t-1}, X_{t-2}, C_t, I_t, G_t, M_t, P_t, P^f_t) \\
M_t &= m(M_{t-1}, M_{t-2}, C_t, I_t, G_t, X_t, P_t, P^f_t)
\end{aligned*}
```

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

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

#### 4.3.5 links to more info on markdown

There are many very good markdown cheatsheets and tutorials on the internet, one cheatsheet can be found [here](https://www.markdownguide.org/cheat-sheet/) (<https://www.markdownguide.org/cheat-sheet/>).

## 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, and how to access them.

### **In this chapter - Python Basics**

This chapter provides an introduction to essential Python concepts for using World Bank models in ModelFlow. It provides a foundational understanding of Python, equipping users with the skills to engage in macroeconomic modeling tasks.

**Readers familiar with python can safely skip this chapter.**

Key points include:

- **Getting Started with Python:**

- Launch Python through the terminal, an IDE, or Jupyter Notebook.
  - Use the Anaconda environment for managing dependencies and versions.

- Core Concepts:**

- Variables and data types: Store and manipulate data using integers, floats, strings, and lists.
  - Control structures: Use loops and conditionals for automating repetitive tasks and decision-making.

- **Libraries and Packages:**

- Leverage Python libraries like pandas for data handling and matplotlib for visualization.
  - Import libraries, modules, and classes to extend Python's functionality.

- Interactivity in Jupyter Notebook:**

- Write, execute, and document Python code interactively.
  - Use context-sensitive help and auto-completion for efficient coding.

- **Best Practices:**

- Organize your code for readability and reusability.
  - Validate results step-by-step to ensure correctness.

## 5.1 Starting python in windows

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

### 5.1.1 Anaconda navigator

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

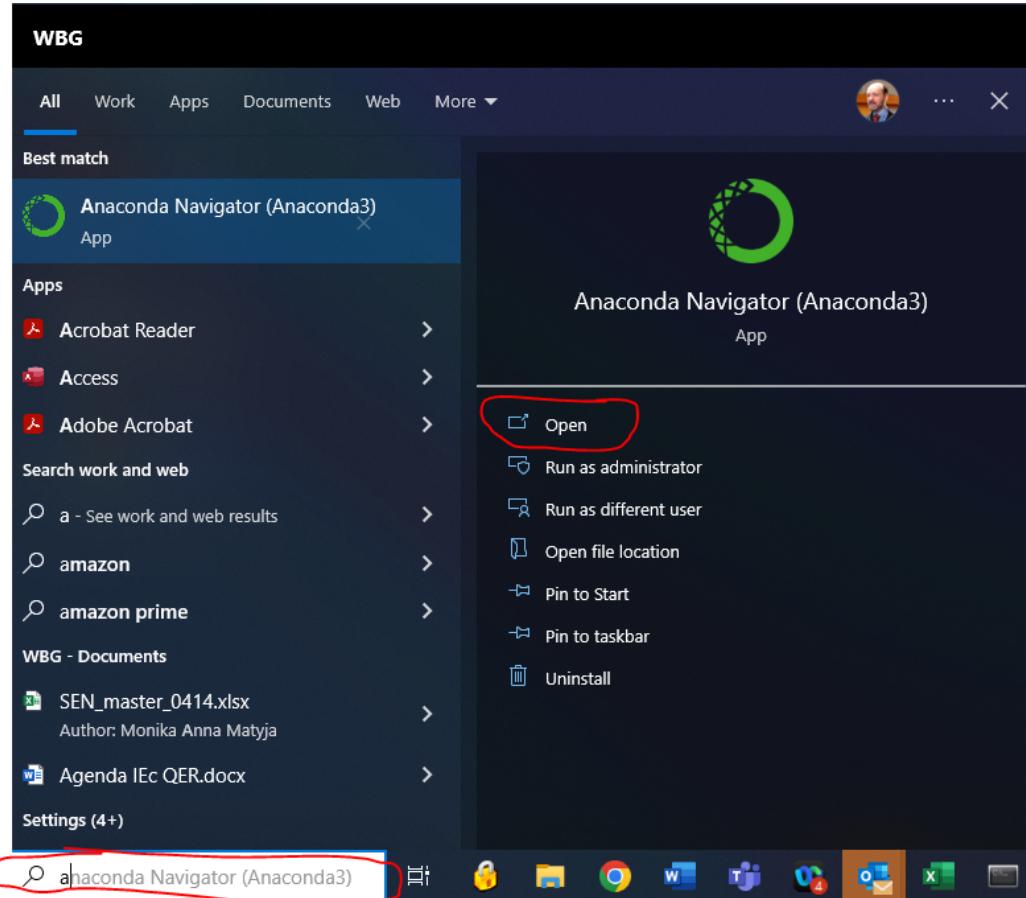


Fig. 5.1: A newly created Jupyter Notebook session

3. Once the environment is selected the user can either select a command line environment, an IDE or Jupyter Notebook by clicking on the appropriate icon. Options typically include:
  1. Jupyter Notebook environment
  2. The command line environment
  3. A programming IDE environment

In the figure below the user is selecting the Jupyter Notebook environment.

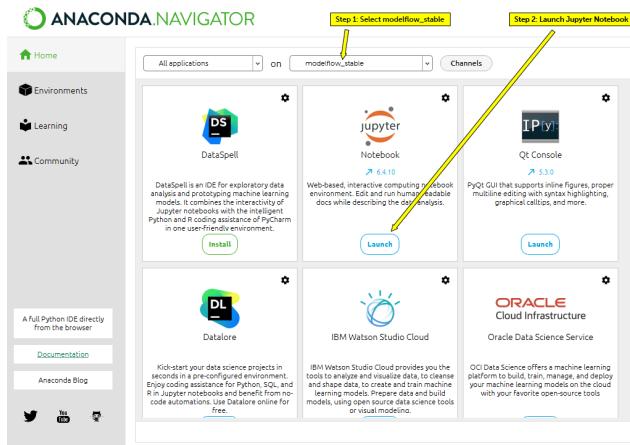


Fig. 5.2: A newly created Jupyter Notebook session

### Note

If you start Jupyter Notebook with Anaconda Navigator you will need to select which environment you want jupyter to be running in (see the drop-down at the top center of the above screen shot). To have access to ModelFlow, you will need to select an environment into which ModelFlow has been installed (there could be more than one).

## 5.2 Python packages, libraries and classes

Some features of `python` are built-in. Others (like ModelFlow) are optional. Optional packages build on basic python features (and those of other packages) but have to be installed separately.

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

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

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

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

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

## 5.3 Importing packages, libraries, modules and classes

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

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

Installation downloads a package's programs from the internet into a specific environment on the user's machine, making them available to be imported when required. Once it has been installed, the environment into which it was installed must be activated and the package must be imported into each python session where it is to be used.

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

### 5.3.1 Import a package

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 call the functions and data that are defined in it (math is a standard package so we do not need to install it separately).

```
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

#### Note

Running code in Jupyter Notebook causes the results of the code to be displayed. As discussed in Chapter 4, a code cell in Jupyter Notebook can be executed either by keystroke short-cut (Control-Enter or Shift-Enter) or by clicking on the Run button on the Jupyter menu bar.

### 5.3.2 Import specific elements or classes from a module or library

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

Rather than importing the whole package (as above), these classes can be imported directly using the **from syntax**.

```
from math import pi,cos,sin
```

When imported in this fashion, the user does not have to precede the class or method with the name of their library. The above `from math import pi,cos,sin` command imports the `pi` constant and the two functions `cos` and `sin` from the `math` package directly and allow the user to call them using their names without preceding them with `math..`

Compare 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
```

### 5.3.3 import a library but give it an alias

An imported library/packages can also be given an alias, that is hopefully shorter than its official name 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 succinct syntax `m.py`.

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

```
3.141592653589793
-0.9899924966004454
```

### 5.3.4 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                               | functionality                                                            |
|-------|-----------------|---------------------------------------|--------------------------------------------------------------------------|
| pd    | pandas          | <code>import pandas as pd</code>      | Pandas are used for storing and retrieving data                          |
| np    | numpy           | <code>import numpy as np</code>       | Numpy gives access to some advanced mathematical features                |
| plt   | matplotlib      | <code>import matplotlib as plt</code> | matplotlib provides a wide-range of routines for plotting numerical data |

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

## INTRODUCTION TO PANDAS, SERIES AND DATAFRAMES

ModelFlow is built on top of the Pandas library. Pandas is the Swiss Army knife of data science and can perform an impressive array of data oriented tasks.

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

- *Pandas homepage* (<https://pandas.pydata.org/>)
- *Pandas community tutorials* ([https://pandas.pydata.org/pandas-docs/stable/getting\\_started/tutorials.html](https://pandas.pydata.org/pandas-docs/stable/getting_started/tutorials.html))

### In this chapter - Introduction to Pandas

This chapter introduces pandas, a powerful Python library for data manipulation and analysis, essential for working with World Bank models. It equips users with basic skills needed to manage and analyze data effectively, and work with the ModelFlow package.

**This chapter can be safely skipped by users familiar with Python and Pandas.**

Key points include:

- **Core Data Structures:**

- **Series:** One-dimensional arrays for handling single-variable data.
- **DataFrames:** Two-dimensional, tabular data structures with labeled rows and columns.

- **Basic Operations:**

- Create Series and DataFrames from lists, dictionaries, or external files (e.g., CSV, Excel).
- Access and manipulate data using indexing, slicing, and filtering.

- **Common Methods:**

- Analyze data with methods like `.mean()`, `.sum()`, and `.describe()`.
- Clean data using `.dropna()` to handle missing values and `.apply()` for transformations.
- Select and modify data subsets using `.loc[]` and `.iloc[]`.

- **Advanced Features:**

- Merge and join datasets for integrating multiple sources.
- Reshape data using pivot tables and group operations.

- **Best Practices:**

- Structure data clearly for easier analysis.
- Use pandas' extensive functionality for efficient handling of large datasets.

### 6.1 Import the pandas library

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

```
import pandas as pd
```

#### Note

By importing pandas with the alias pd (`import pandas as pd`), future references to the pandas library can be truncated as by using `pd.` instead of `pandas.`

The Pandas library contains many classes and methods. The discussion below focuses on **Series** and **DataFrames**, two classes that are part of the pandas library. Both series and dataframes are containers that can be used to store time-series data and that have associated with them a number of very useful methods for displaying and manipulating time-series data.

Unlike other statistical packages neither `series` nor `dataframes` are inherently or exclusively time-series in nature. ModelFlow and macro-economists use them in this way, but the classes themselves are not necessarily dated or numerical.

### 6.2 The Series class in Pandas

`Series` is a class that is part of the pandas package and can be used to instantiate an object that holds a two dimensional array comprised of values and an index.

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

#### 6.2.1 Series declared from a list

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

#### Note

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

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

```
mylist=[2,7,8,9]
```

```
mylist2=[“Some text”, “Some more Text”, 2,3]
```

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

In the examples below `Simplest`, `Simple`, `Simple2` and `simple3` are all python data objects of type `Pandas.Series`. Each Series object is instantiated (created) in a different way. `Simplest` is instantiated by passing it a python list of numbers, `simple2` is instantiated by passing a list object (values), `series3` is instantiated indirectly by passing a list mixing text and numeric values. `Series3` would be hard to interpret as an economic series, but is a perfectly valid python series.

 Note

All series have an index. If no index is explicitly declared (see following section) python will assign a zero-based index (a numerical index that starts with 0).

```
values=[7,8,9,10,11]
weird=["Some text","Some more Text",2,3]
Here the constructor is passed a numeric list
Simplest=pd.Series([2,3,4,5,6,7])
Simplest
```

```
0 2
1 3
2 4
3 5
4 6
5 7
dtype: int64
```

```
In this case the constructor is passed a variable that was defined above as a list
simple2=pd.Series(values)
simple2
```

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

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

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

Note that all three series have different length (6 datapoints for `simplest`; 5 for `simple2` and 4 for `simple3`).

### 6.2.2 Series declared using a specific index

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

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

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

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

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

Now these Series look more like time-series data!

### 6.2.3 Create Series from a dictionary

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

#### Note

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

Thus mydict={"1966":2,"1967":3,"1968":4,"1969":5,"2000":-15} creates a dictionary 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:

In the following example the keys are numeric

```
mydict2={1966:2,1967:3,1968:4,1969:5,2000:-15}
```

Here, mydict2 links (maps) the key 1966 to the value 2 and "1969" links to the value 5.

The series constructor can also accept a dictionary. In this instance, the key of the dictionary becomes the index of the series that was used to instantiate the series.

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

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

## 6.3 The DataFrame class in Pandas

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

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

Much more detail on standard pandas dataframes can be found on the *official pandas website* <https://pandas.pydata.org/docs/reference/frame.html>.

### 6.3.1 Creating or instantiating a dataframe

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

The `pandas.DataFrame()` method is constructor for the DataFrame class. It can take several forms (as with Series), but always returns (instantiates) an instance of a DataFrame object – i.e. a variable whose contents are a DataFrame.

The code example below creates a DataFrame called `df` comprised of three columns B,C and E; indexed between 2018 and 2021. Macroeconomists may interpret the index as dates, but for pandas they are just numbers.

The DataFrame is instantiated from a dictionary and assigned a specific index by passing a list of years as the index.

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

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

#### Note

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

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

### 6.3.2 Alternative ways to set the time period of a dated index

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 `number_of_rows` times (6 times in this example) setting `v` equal to `2020+0, 2020+1,...,2020+5` (recall, that python is zero-indexed so the first value of `v` is 0, not one). 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 then change the staring date in the loop from 2020 to 1990.

```
#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']) # Assign the output to two_
˓→columns,named A and B
df
```

|      | A   | B   |
|------|-----|-----|
| 2020 | 100 | 100 |
| 2021 | 100 | 100 |
| 2022 | 100 | 100 |
| 2023 | 100 | 100 |
| 2024 | 100 | 100 |
| 2025 | 100 | 100 |

This second example simplifies the creation even further by specifying the begin and end point as a range.

```
df1 = pd.DataFrame(200,
 index=[v for v in range(2020,2030)], # create row index
 # equivalent to index=[2020,2021,...,2029]
 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 |

#### ⚠ Warning

the `range(2020,2030)` creates an indexed series from 2020 through 2029, not 2030 as might be expected. If you want

a value created for 2030 then the range would have to be range(2020,2031)

### 6.3.3 Adding a column to a dataframe

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

#### Note

The size of the object assigned to the new column must match the size (number of rows) of the pre-existing DataFrame. Originally df has 6 rows so we must supply 6 data points for this command to run error free.

```
df["NEW"]=[10,12,10,13,14,15] #df originally has 6 rows so we must supply 6 data_
˓→points for this command to run error free
df
```

|      | A   | B   | NEW |
|------|-----|-----|-----|
| 2020 | 100 | 100 | 10  |
| 2021 | 100 | 100 | 12  |
| 2022 | 100 | 100 | 10  |
| 2023 | 100 | 100 | 13  |
| 2024 | 100 | 100 | 14  |
| 2025 | 100 | 100 | 15  |

### 6.3.4 Revising values

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

#### Warning

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

Below we change the values of the NEW Column that we generated earlier.

```
df["NEW"]=[11,12,10,14,2,1]
df
```

|      | A   | B   | NEW |
|------|-----|-----|-----|
| 2020 | 100 | 100 | 11  |
| 2021 | 100 | 100 | 12  |
| 2022 | 100 | 100 | 10  |
| 2023 | 100 | 100 | 14  |
| 2024 | 100 | 100 | 2   |
| 2025 | 100 | 100 | 1   |

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

|      | A   | B  | NEW |
|------|-----|----|-----|
| 2020 | 100 | 17 | 11  |
| 2021 | 100 | 17 | 12  |
| 2022 | 100 | 17 | 10  |
| 2023 | 100 | 17 | 14  |
| 2024 | 100 | 17 | 2   |
| 2025 | 100 | 17 | 1   |

### Note

The declaration of the column name, can either use single or double quotes – but they cannot be mixed. Thus

```
df['A']=7
```

is fine, and so is `df["B"] = 7` but `df["Error"] = 7` would generate an error because the string created with " is not closed by another matching ". Similarly the string created with ' is not closed by a matching '.

## 6.4 Selected pandas methods

Pandas has a number of methods (functions that operate on the underlying data of a pandas object). Several of them are used when working with a World Bank model in ModelFlow and these are discussed below.

### 6.4.1 .columns lists the column names of a dataframe

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

```
df.columns
```

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

### 6.4.2 .size indicates the dimension of a list

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

```
df.columns.size
```

```
3
```

The dataframe `df` has 3 columns.

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

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

```
df.eval('X = B*NEW')
```

|      | A   | B  | NEW | X   |
|------|-----|----|-----|-----|
| 2020 | 100 | 17 | 11  | 187 |
| 2021 | 100 | 17 | 12  | 204 |
| 2022 | 100 | 17 | 10  | 170 |
| 2023 | 100 | 17 | 14  | 238 |
| 2024 | 100 | 17 | 2   | 34  |
| 2025 | 100 | 17 | 1   | 17  |

### 6.4.4 multiple expressions can be evaluated with .eval()

With .eval() multiple expressions can be evaluated at the same time. In the example below three apostrophe's are used to indicate a multiple-line block of text. Each line is executed separately by eval.

In this example, two new variables X and THE\_ANSWER are created in a single call to .eval().

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

|      | A   | B  | NEW | X   | THE_ANSWER |
|------|-----|----|-----|-----|------------|
| 2020 | 100 | 17 | 11  | 187 | 42         |
| 2021 | 100 | 17 | 12  | 204 | 42         |
| 2022 | 100 | 17 | 10  | 170 | 42         |
| 2023 | 100 | 17 | 14  | 238 | 42         |
| 2024 | 100 | 17 | 2   | 34  | 42         |
| 2025 | 100 | 17 | 1   | 17  | 42         |

#### Note

In python three apostrophes ''' signals a block of text. Of course the block needs to be opened and closed – in each case by three apostrophes. New lines that appear within a block (if any) are an integral part of the block.

Because the result of the .df.eval() call was not assigned to anything, least of all the dataframe df, the value of df is unchanged.

```
df
```

|      | A   | B  | NEW |
|------|-----|----|-----|
| 2020 | 100 | 17 | 11  |
| 2021 | 100 | 17 | 12  |
| 2022 | 100 | 17 | 10  |
| 2023 | 100 | 17 | 14  |
| 2024 | 100 | 17 | 2   |
| 2025 | 100 | 17 | 1   |

To store the results of the calculation, the expression must be assigned to a variable. The pre-existing DataFrame can be overwritten by assigning it the result of the eval statement or a new DataFrame could be created.

```
df=df.eval(''X = B*NEW
 Y = 42'''
df
```

|      | A   | B  | NEW | X   | Y  |
|------|-----|----|-----|-----|----|
| 2020 | 100 | 17 | 11  | 187 | 42 |
| 2021 | 100 | 17 | 12  | 204 | 42 |
| 2022 | 100 | 17 | 10  | 170 | 42 |
| 2023 | 100 | 17 | 14  | 238 | 42 |
| 2024 | 100 | 17 | 2   | 34  | 42 |
| 2025 | 100 | 17 | 1   | 17  | 42 |

With this operation the new columns, X and Y 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 in the next chapter are ModelFlow features that extend the functionalities native to `dataframe` that allows such calculations to be performed.

```
df2=df.eval(''X = B*NEW
 Y = 42'''
df2
```

|      | A   | B  | NEW | X   | Y  |
|------|-----|----|-----|-----|----|
| 2020 | 100 | 17 | 11  | 187 | 42 |
| 2021 | 100 | 17 | 12  | 204 | 42 |
| 2022 | 100 | 17 | 10  | 170 | 42 |
| 2023 | 100 | 17 | 14  | 238 | 42 |
| 2024 | 100 | 17 | 2   | 34  | 42 |
| 2025 | 100 | 17 | 1   | 17  | 42 |

Here the results are assigned to a new dataframe `df2`.

## 6.5 The `.loc[]` method

The `.loc[]` method is a powerful pandas routine used extensively in ModelFlow. `.loc[]` selects a portion (slice) of a dataframe and allows the user to display and/or revise specific sub-sections of a single or several columns or rows in a dataframe.

### 6.5.1 Selecting a single element in a dataframe: `.loc[row, column]`

`.loc[row, column]` operates on a single cell in the dataframe. Thus, the below displays the value of the cell with index=2023 observation from the column NEW.

```
df.loc[2023, 'NEW']
```

```
np.int64(14)
```

### 6.5.2 Selecting a single columns: `.loc[:, column]`

The lone colon in a `.loc[]` statement indicates all the rows or columns. Here all of the rows of the column labeled NEW are displayed.

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

```
2020 11
2021 12
2022 10
2023 14
2024 2
2025 1
Name: NEW, dtype: int64
```

### 6.5.3 Selecting a single row: `.loc[row, :]`

Here all of the columns, for the selected row are displayed.

```
df.loc[2023, :]
```

```
A 100
B 17
NEW 14
X 238
Y 42
Name: 2023, dtype: int64
```

### 6.5.4 Selecting specific rows and columns: `.loc[[xxx,xxx],[names...]]`

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[[2021,2024],['B','NEW']]
```

```
 B NEW
2021 17 12
2024 17 2
```

### 6.5.5 Select a range: the “:” operator in a ‘.loc’ statement

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

Here from 2018 to 2019.

```
df.loc[2021:2023, ['B', 'NEW']]
```

|      | B  | NEW |
|------|----|-----|
| 2021 | 17 | 12  |
| 2022 | 17 | 10  |
| 2023 | 17 | 14  |

### 6.5.6 Revising a sub-section of a DataFrame, using .loc[] on the left hand side to assign values to specific cells

This can be handy when revising a DataFrame in preparation for running a scenario.

```
df.loc[2022:2024, 'NEW'] = 20
df
```

|      | A   | B  | NEW | X   | Y  |
|------|-----|----|-----|-----|----|
| 2020 | 100 | 17 | 11  | 187 | 42 |
| 2021 | 100 | 17 | 12  | 204 | 42 |
| 2022 | 100 | 17 | 20  | 170 | 42 |
| 2023 | 100 | 17 | 20  | 238 | 42 |
| 2024 | 100 | 17 | 20  | 34  | 42 |
| 2025 | 100 | 17 | 1   | 17  | 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 a single value that is then broadcast into all of the elements of the left hand slice.

For more on broadcasting see here (<https://jakevdp.github.io/PythonDataScienceHandbook/02.05-computation-on-arrays-broadcasting.html>)

#### For more info on the .loc[] method

- Description <https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.loc.html>.
- Search <https://www.google.com/search?q=pandas+dataframe+loc&newwindow=1> (<https://www.google.com/search?q=pandas+dataframe+loc&newwindow=1>).

#### For more info on pandas:

- Pandas homepage <https://pandas.pydata.org/>.
- Pandas community tutorials [https://pandas.pydata.org/pandas-docs/stable/getting\\_started/tutorials.html](https://pandas.pydata.org/pandas-docs/stable/getting_started/tutorials.html).

## **Part III**

### **Selected modelflow extensions to pandas**



## MODELFLOW AND PANDAS DATAFRAMES

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 ways. Some of the methods below are standard pandas methods, others have been added to by ModelFlow features.

Data in a `DataFrame` can be modified directly with built-in pandas functionalities like `.loc[]` and `eval()` discussed in the previous chapter but ModelFlow extends these capabilities in important ways.

### In this chapter - ModelFlow and Pandas DataFrames

This chapter explores the integration of `DataFrames` into ModelFlow and the extensions to standard pandas by ModelFlow that facilitate working with economic models.

Key points include:

- **Pandas and ModelFlow:**

- `DataFrames` are central to organizing and manipulating model data.
  - ModelFlow extends pandas functionality for time-series and macroeconomic analysis.

- Key Features:**

- **Column Names:** Ensure consistent variable naming conventions in `DataFrames`.
  - **Index and Time Dimensions:** Use indexed `DataFrames` to handle time-based data effectively.
  - **Leads and Lags:** ModelFlow provides built-in methods to manage lead and lag relationships between variables.

- Core Methods:**

- `.upd()`: Updates a `DataFrame` with new values or transformations for variables.
  - `.mfcalc()`: Calculates transformed variables based on the model's equations.

## 7.1 Column names in ModelFlow

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

```{note} ModelFlow variable names ModelFlow places more restrictions on column names than do pandas *per se*.

ModelFlow variable names must **start with a letter** and **be upper case**.

Thus while all the below are legal column names in pandas, some are illegal in ModelFlow.

| Variable Name | Legal in modelflow? | 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 |

7.2 .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 (quarterly, monthly, weekly,daily, etc.) the index can be one of several pandas date types, but users are encouraged to use `pd.period_range()` to create date indexes, because the ModelFlow reporting methods (tables and graphs) work well with indexes generated using this method.

⚠ Warning

Not all python datatypes work well with the graphics routines of ModelFlow. Users are advised to use the `pd.period_range()` method to generate date indexes for higher-frequency data (i.e. monthly or quarterly data).

For example:

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

7.3 Leads and lags

Pandas does not support the economic idea of leads and lags *per se* (although the `.shift()` operator can be used to emulate the same idea in ordered DataFrames).

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

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

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

7.4 The `.upd()` method returns a DataFrame with updated variables.

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.

`.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

Note

`.upd()` does not change the value of the DataFrame upon which it operates, but its results can be assigned to a DataFrame. In the examples below the originating DataFrame `df` is never overwritten. It could be, by assigning the result of an `upd()` command to `df`, i.e.

```
df.upd('B = 7')
```

would set B from the DataFrame `df` equal to 7 and the function returns a temporary DataFrame whose results can be visualized. The values of `df` are not changed in this example.

By contrast:

```
df=df.upd('B = 7')
```

performs the same operation but assigns the results of the operation to `df` – overwriting its earlier values.

Warning

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.

7.4.1 some examples of the `.upd()` method

First, 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': [1,1,1,1,1,1], 'C':[1,2,3,6,8,9], 'E':[4,4,4,4,4,4]}, index=[v
↪for v in range(2020,2026)])
```

```
df
```

| | B | C | E |
|------|---|---|---|
| 2020 | 1 | 1 | 4 |
| 2021 | 1 | 2 | 4 |
| 2022 | 1 | 3 | 4 |
| 2023 | 1 | 6 | 4 |
| 2024 | 1 | 8 | 4 |
| 2025 | 1 | 9 | 4 |

Use `.upd` to create a new variable

With standard pandas a user can add a column (series) to a DataFrame simply by assigning a value to an new column name (NEW2 had existed its values would have been updated. For example:

```
df['NEW2']=[17,12,14,15]
.upd() provides this functionality as well.
```

```
df.upd('c = 142.0')
```

| | B | C | E |
|------|---|-------|---|
| 2020 | 1 | 142.0 | 4 |
| 2021 | 1 | 142.0 | 4 |
| 2022 | 1 | 142.0 | 4 |
| 2023 | 1 | 142.0 | 4 |
| 2024 | 1 | 142.0 | 4 |
| 2025 | 1 | 142.0 | 4 |

Note

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

The automatic creation of new variables can be suspended by setting the option: `create = False`. For more look see the discussion of the '`upd()` option `create=True`.

7.4.2 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 (for muliple lines the needs to be passed using triple apostrophes '''' or three quote symbols """ as below).

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 two new variables, C and D.

Note

Inheritance of time periods in the `.upd()` method The third line inherits the time period of the previous line.

Note also, the submitted string can include comments as well (denoted with the standard python # indicator).

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

| | B | C | E | A | D |
|------|-------|-------|---|------|------|
| 2020 | 200.0 | 500.0 | 4 | 0.0 | 33.0 |
| 2021 | 1.0 | 2.0 | 4 | 42.0 | 33.0 |
| 2022 | 1.0 | 3.0 | 4 | 44.0 | 33.0 |
| 2023 | 1.0 | 6.0 | 4 | 45.0 | 33.0 |
| 2024 | 1.0 | 8.0 | 4 | 46.0 | 33.0 |
| 2025 | 1.0 | 9.0 | 4 | 0.0 | 33.0 |

Note

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

7.4.3 Update several variables 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 `.upd()` can accept several left-hand side 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
''')
```

| | B | C | E | H | I | J | K | P | Q | R | \ |
|------|---|---|---|-----|-----|-----|-----|-------------|-------------|-------------|---|
| 2020 | 1 | 1 | 4 | 0.0 | 0.0 | 0.0 | 0.0 | 1000.000000 | 1000.000000 | 1000.000000 | |
| 2021 | 1 | 2 | 4 | 0.0 | 0.0 | 0.0 | 0.0 | 1020.000000 | 1020.000000 | 1020.000000 | |

(continues on next page)

(continued from previous page)

| | | | | | | | | | | |
|------|---|---|---|-------------|------|------|------|-------------|-------------|-------------|
| 2022 | 1 | 3 | 4 | 40.0 | 40.0 | 40.0 | 40.0 | 1040.400000 | 1040.400000 | 1040.400000 |
| 2023 | 1 | 6 | 4 | 40.0 | 40.0 | 40.0 | 40.0 | 1061.208000 | 1061.208000 | 1061.208000 |
| 2024 | 1 | 8 | 4 | 40.0 | 40.0 | 40.0 | 40.0 | 1082.432160 | 1082.432160 | 1082.432160 |
| 2025 | 1 | 9 | 4 | 0.0 | 0.0 | 0.0 | 0.0 | 1104.080803 | 1104.080803 | 1104.080803 |
| | | | | S | | | | | | |
| 2020 | | | | 1000.000000 | | | | | | |
| 2021 | | | | 1020.000000 | | | | | | |
| 2022 | | | | 1040.400000 | | | | | | |
| 2023 | | | | 1061.208000 | | | | | | |
| 2024 | | | | 1082.432160 | | | | | | |
| 2025 | | | | 1104.080803 | | | | | | |

Box 2. Time scope of .upd() commands

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 specified the operation is applied over a period of time.

The selected time period will persist until re-set with a new time specification. This is useful if several variables are going to be updated for the same time period, but must be kept in mind if subsequent commands are to operate over a different time period.

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

- -0 indicates the start of the dataframe
- -1 indicates the end of the dataframe

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

The = operator causes the left-hand side variable to be set equal to the values following the = operator.

Note

Either:

- Only one data point is provided. In this case its value is applied to all dates in the indicated period, or
- The number of data points provided must match the number of dates in the period specified.

In the example below:

- The first line sets the variable A during the specified period to specific values
- The second line sets the variable B to 200 in 2023
- The third line inherits the time period set in the second line (2023) and sets the variable C equal to 500.

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

(continues on next page)

(continued from previous page)

```
c = 500
"""
# inherits previous time-period specification (2023)
```

| | B | C | E | A |
|------|-------|-------|---|------|
| 2020 | 1.0 | 1.0 | 4 | 0.0 |
| 2021 | 1.0 | 2.0 | 4 | 42.0 |
| 2022 | 1.0 | 3.0 | 4 | 44.0 |
| 2023 | 200.0 | 500.0 | 4 | 45.0 |
| 2024 | 1.0 | 8.0 | 4 | 46.0 |
| 2025 | 1.0 | 9.0 | 4 | 0.0 |

7.4.4 Operators of the .upd() method

The .upd() method takes a variety of mathematical operators, some of these are described below.

Types of update:

| Update to perform | Use this operator |
|--|-------------------|
| Set a variable equal to the input | = |
| Add the input to the LHS variable | + |
| Set the variable to itself multiplied by the input | * |
| Increase/Decrease the variable by a percent of itself - i.e. multiplies itself by (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 | +growth |
| Set the amount by which the variable should increase from its previous period level ($\Delta = var_t - var_{t-1}$) | =diff |

The “=” operator: assign a value(s) to a variable

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']=[11,17,12,14,15,17]
```

df.upd('NEW2 = 11 17 12 14 15 17') provides this functionality as well.

Note that with .upd() the multiple values are space delimited, versus standard pandas where they are passed as a comma delimited list.

```
df.upd('NEW2 = 11 17 12 14 15 17')
```

| | B | C | E | NEW2 |
|------|---|---|---|------|
| 2020 | 1 | 1 | 4 | 11.0 |
| 2021 | 1 | 2 | 4 | 17.0 |
| 2022 | 1 | 3 | 4 | 12.0 |
| 2023 | 1 | 6 | 4 | 14.0 |
| 2024 | 1 | 8 | 4 | 15.0 |
| 2025 | 1 | 9 | 4 | 17.0 |

The “+” operator adds to the existing values in the specified range

In the example below 42 is added to the pre-existing values of the variable B over the period 2022 and 2024, and a different value is subtracted from each of the three rows selected.

```
df.upd(''  
# Or one number to all years in between start and end  
<2022 2024> B + 42 # one value broadcast to 3 years  
<2022 2024> E + -1 -2 -3 # add (subtract) a different value to each of the three  
→rows specified  
'')
```

| | B | C | E |
|------|------|---|-----|
| 2020 | 1.0 | 1 | 4.0 |
| 2021 | 1.0 | 2 | 4.0 |
| 2022 | 43.0 | 3 | 3.0 |
| 2023 | 43.0 | 6 | 2.0 |
| 2024 | 43.0 | 8 | 1.0 |
| 2025 | 1.0 | 9 | 4.0 |

The * operator multiplies all values in a range by the specified values

In the example below the pre-existing values of the variable A for years 2021, 2022 and 2023 are multiplied by three different numbers (42, 44 and 45 respectively).

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

| | B | C | E |
|------|------|---|---|
| 2020 | 1.0 | 1 | 4 |
| 2021 | 42.0 | 2 | 4 |
| 2022 | 44.0 | 3 | 4 |
| 2023 | 55.0 | 6 | 4 |
| 2024 | 1.0 | 8 | 4 |
| 2025 | 1.0 | 9 | 4 |

The % operator increases all values in a range by a specified percent amount

In this example:

- A new column is generated with value 1 in every year
- A is increased by 42 and 44% over the range 2021 through 2022.
- B is increased by 10 percent in all years
- C The values of C are overwritten and set to 100 for the whole range (because the previous line set the active range to <-0 -1>)
- C is decreased by 12 percent over the range 2023 through 2025.

```
df.upd(''  
<-0 -1> A = 1
```

(continues on next page)

(continued from previous page)

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

| | B | C | E | A |
|------|-----|-------|---|------|
| 2020 | 1.1 | 100.0 | 4 | 1.00 |
| 2021 | 1.1 | 100.0 | 4 | 1.42 |
| 2022 | 1.1 | 100.0 | 4 | 1.44 |
| 2023 | 1.1 | 88.0 | 4 | 1.00 |
| 2024 | 1.1 | 88.0 | 4 | 1.00 |
| 2025 | 1.1 | 88.0 | 4 | 1.00 |

The =GROWTH operator sets the percent growth rate to specified values

The =GROWTH operator sets the growth rate of the variable to the indicated level.

```
res = df.upd('''
# Same number of values as years
<-0 -1> A = 100
<2021 2022> A =GROWTH 1 5
<2020> c = 100
<2021 2025> c =GROWTH 2
'''')
# print the resulting DataFrame (res) first as levels and then as a growth rate
# using the pandas pct_change() method
print(f'Dataframe:\n{res}\n\nGrowth:\n{res.pct_change()*100.0}\n')
```

| Dataframe: | | | | |
|------------|---|------------|---|--------|
| | B | C | E | A |
| 2020 | 1 | 100.000000 | 4 | 100.00 |
| 2021 | 1 | 102.000000 | 4 | 101.00 |
| 2022 | 1 | 104.040000 | 4 | 106.05 |
| 2023 | 1 | 106.120800 | 4 | 100.00 |
| 2024 | 1 | 108.243216 | 4 | 100.00 |
| 2025 | 1 | 110.408080 | 4 | 100.00 |

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

The +GROWTH operator adds or subtracts from the existing percent growth rate

The below example is a bit more complicated, reflecting the fact that each line is executed sequentially.

The first line sets the value of A to 1 for the whole period.

The second line sets the growth rate of A and STEP2 in 2021 to 1% (STEP2 is changed so that we can inspect the intermediate value that A would have had after execution of line 2 but before the execution of line 3).

The third line adds 2 to the growth rates of A in each period after 2021. For 2021 the growth rate was and remains unchanged. The value of A in 2022 following the execution of line 2 is 1 (see the values for STEP2). The pre-existing growth rate of A is actually negative (see growth rate of STEP2).

Adding 2 to this growth rate results in a growth rates of a little more than 1 in 2022. The growth rate in the following years was zero (see B) and is now 2 percent.

```
res =df.upd('''
<-0 -1> A STEP2 = 1
<2021 > A STEP2 =GROWTH 1 # All selected years set to the same growth rate
<2022 -1> A +growth 2 # Add to the existing growth rate these numbers
''')
print(f'Dataframe:\n{res}\n\nGrowth:\n{res.pct_change()*100}\n')
```

| | B | C | E | A | STEP2 |
|------|---|---|---|----------|-------|
| 2020 | 1 | 1 | 4 | 1.000000 | 1.00 |
| 2021 | 1 | 2 | 4 | 1.010000 | 1.01 |
| 2022 | 1 | 3 | 4 | 1.020200 | 1.00 |
| 2023 | 1 | 6 | 4 | 1.040604 | 1.00 |
| 2024 | 1 | 8 | 4 | 1.061416 | 1.00 |
| 2025 | 1 | 9 | 4 | 1.082644 | 1.00 |

| | B | C | E | A | STEP2 |
|------|-----|------------|-----|----------|-----------|
| 2020 | NaN | NaN | NaN | NaN | NaN |
| 2021 | 0.0 | 100.000000 | 0.0 | 1.000000 | 1.000000 |
| 2022 | 0.0 | 50.000000 | 0.0 | 1.009901 | -0.990099 |
| 2023 | 0.0 | 100.000000 | 0.0 | 2.000000 | 0.000000 |
| 2024 | 0.0 | 33.333333 | 0.0 | 2.000000 | 0.000000 |
| 2025 | 0.0 | 12.500000 | 0.0 | 2.000000 | 0.000000 |

The =diff operator recursively sets the value of a pre-existing variable to rise or fall by the specified amount

The command =diff (mathematically $\Delta = var_t - var_{t-1} = some\ number$) below 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 recursively adds 2 to UPBY2’s previous period’s revised value. As a result it increments over time by 2.

```
df.upd('''
<-0 -1> A = 1
<2021 2022> A =diff 2 4 # Same number of values as years
<2020 > UpBy2 = 100 # sets 2020 value of UPBy2 to 100
<2021 2025> UpBy2 =diff 2 # increases by 2 from 2021 to 2025
'''')
```

| | B | C | E | A | UPBY2 |
|------|---|---|---|-----|-------|
| 2020 | 1 | 1 | 4 | 1.0 | 100.0 |
| 2021 | 1 | 2 | 4 | 3.0 | 102.0 |
| 2022 | 1 | 3 | 4 | 7.0 | 104.0 |
| 2023 | 1 | 6 | 4 | 1.0 | 106.0 |
| 2024 | 1 | 8 | 4 | 1.0 | 108.0 |
| 2025 | 1 | 9 | 4 | 1.0 | 110.0 |

7.4.5 Options of the `.upd()` method

In addition to the operators discussed above the `.upd()` method has several options that affect the way that it behaves. The most important of these are summarized below.

| Option | Example | Effect |
|-------------|---|--|
| keep_growth | keep_growth=True | When set, the post-shock growth rate of variables updated over a sub-period will be preserved, implying that in the period after the growth operation their levels will change (be higher because of higher growth in earlier periods) but their growth rates will remain unchanged. When not set their levels remain unchanged. |
| -kg | df=df.upd("<2022
2023> A
+ 5 -kg") | A line level option that has the same effect as <code>keep_growth=True</code> |
| -nkg | df=df.upd("<2022
2023> A
+ 5 -nkg") | A line level option that has the same effect as <code>keep_growth=False</code> |
| scale | scale=0.5 | The update value(s) are multiplying by the scale before the update is performed. Useful for sensitivity analysis. Defaults to 1.0, i.e. no sensitivity analysis and the full value is passed to update. |
| lprint | lprint=True | When set, causes the pre and post values of affected variables to be printed. |
| create | create=False | When set (default), will cause the LHS variable in an update command to be created. If False update will throw an error if the LHS variable does not exist. |

The `keep_growth` option (`-kg` and `-nkg`)

When changing data and for certain kinds of simulations, it can sometime be useful to be able to update variables but keep the growth rate in subsequent periods unchanged. In database management this is frequently done when two time-series with different levels are spliced together. When forecasting this is useful if you have updated historical data but your views on future growth rates are unchanged.

The `-kg` or `-keep_growth` option instructs ModelFlow 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.

The default keep_growth behavior

The `keep_growth` option 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 are 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*.

By default `keep_growth` is set to `False`.

Note

At the line level:

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

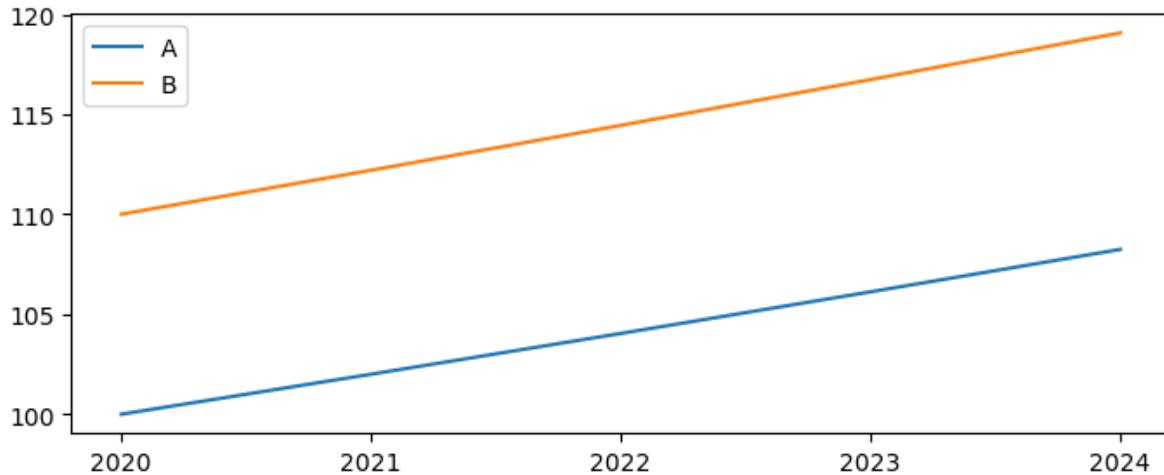
Consider the following concrete example. A DataFrame `df` has two variables `A` and `B`, that each grow by 2% per period, with `A` initialized at a level of 100 and `B` at a level of 110 so that we can see each separately on a graph.

```
df = pd.DataFrame(100.,
                  index=[v for v in range(2020, 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 |

```
df[['A', 'B']].plot(xticks=df.index, figsize=(7,3)); #the xticks option forces
#mathplotlib to only print x-axis values that exist in the index (no decimals)
```



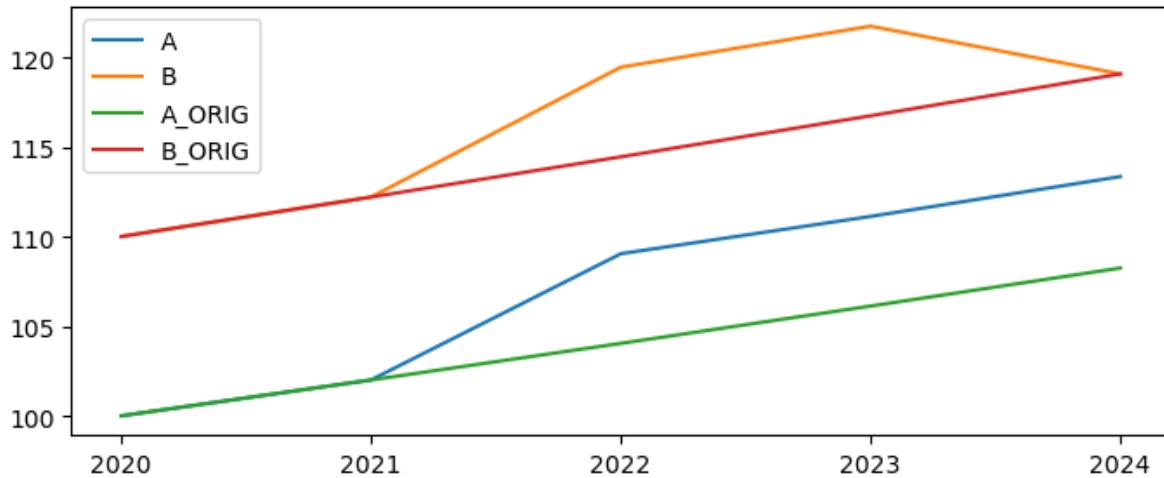
The `.upd()` command below modifies both A and B by adding 5 to their levels in each of 2022 and 2023.

For A, this is done with the `keep_growth` option set to `True` – the `--kg` option in the code below. This means that for A the growth rate after the shock period 2022-23 will be unchanged at 2 percent.

For series B the same shock is applied but with `keep_growth` set to `False` using the `-nkg` option.

The `keep_growth` global variable is ignored in this instance as each line in the update is overriding it 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(xticks=df.index, figsize=(7,3));
```



In the first example 'A' (the green and blue lines) the level of A is increased by 5 for two periods (2021-2022). The levels of the subsequent values are also increased because the previous growth rate (2%) is now applied to the new higher level of the data in 2022.

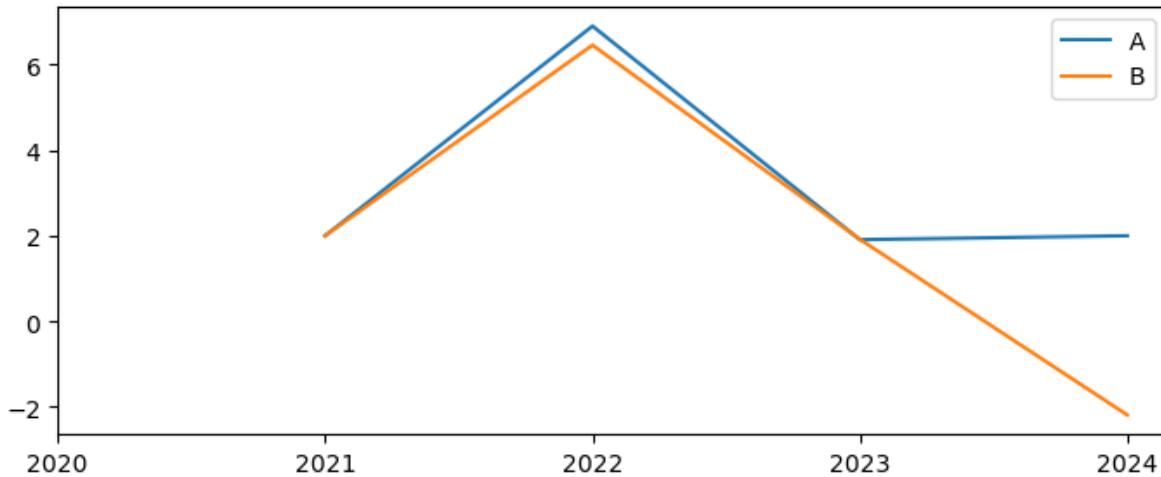
For the 'B' variable the same level change was input but because of the `--nkg` (equivalent to `keep_growth=False`) the periods after the change were unaffected. the shocked variable returns to its pre-shock level immediately in 2023.

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

The World Bank's MFMod Framework in Python with Modelflow

Here the growth in both series accelerates in 2022, by slightly less than 5 percentage points because a) the base of each is more than 100 in 2021 (because of the 2 percent growth in 2021). Substantially more in the case of B, which was initialized at 110. In 2023 the growth rate of A returns to 2 percent, while the growth rate of B is actually negative because the level (see earlier graph) has fallen back to its original level.

```
dfg=df[['A','B']].pct_change()*100  
dfg.plot(xticks=dfg.index,figsize=(7,3));
```



Keep_growth additional examples

Initialize a new dataframe, with some growth rate

```
# instantiate a new dataframe with one column 'A' with a value 100 everywhere and  
# index 2020-2025  
dftest = pd.DataFrame(100.0,  
                      index=[v for v in range(2020,2026)], # create row index  
                      # equivalent to index=[2020,2021,2022,2023,2024,2025]  
                      columns=['A']) # create column name  
dftest
```

| | A |
|------|-------|
| 2020 | 100.0 |
| 2021 | 100.0 |
| 2022 | 100.0 |
| 2023 | 100.0 |
| 2024 | 100.0 |
| 2025 | 100.0 |

```
# Update a to have growth rate accelerationg linearly by 1 from 1 Percent 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')
```

| | A |
|------|------------|
| 2020 | 100.000000 |
| 2021 | 101.000000 |
| 2022 | 103.020000 |

(continues on next page)

(continued from previous page)

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

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

```
nokg = original.upd('''
<2021 2025> a =growth 1 2 3 4 5
<2021 2023> a = 120
''',lprint=0)
kg = original.upd('''
<2021 2025> a =growth 1 2 3 4 5
<2021 2023> a = 120 --kg
''',lprint=0)
kg=kg.rename(columns={"A":"KG"}) #rename cols to facilitate display
nokg=nokg.rename(columns={"A":"NOKG"}) #rename cols to facilitate display
df=original.rename(columns={"A":"Orig"}) #rename cols to facilitate display
combo=pd.concat([kg,nokg,df], axis=1)
combo
print(f'Levels\n{combo}\n\nGrowth\n{combo.pct_change()*100}')
```

| Levels | | | |
|--------|--------|------------|------------|
| | KG | NOKG | Orig |
| 2020 | 100.00 | 100.000000 | 100.000000 |
| 2021 | 120.00 | 120.000000 | 101.000000 |
| 2022 | 120.00 | 120.000000 | 103.020000 |
| 2023 | 120.00 | 120.000000 | 106.110600 |
| 2024 | 124.80 | 110.355024 | 110.355024 |
| 2025 | 131.04 | 115.872775 | 115.872775 |

| Growth | | | |
|--------|------|-----------|------|
| | KG | NOKG | Orig |
| 2020 | NaN | NaN | NaN |
| 2021 | 20.0 | 20.000000 | 1.0 |
| 2022 | 0.0 | 0.000000 | 2.0 |
| 2023 | 0.0 | 0.000000 | 3.0 |
| 2024 | 4.0 | -8.03748 | 4.0 |
| 2025 | 5.0 | 5.000000 | 5.0 |

Understanding the results

In the first example where KG (keep_growth) was set, 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=True the KG variable growth at 2 percent per year in 2024 and 2025.

In the **--nkg** example, the levels of NOKG are the same as KG for 2020 through 2023, but because **--nkg** was selected the levels revert to their pre-shock values, which are lower than the 120 in 2023. As a result the growth rate for NOKG is negative in 2024. The growth rate for 2024 remains 5 because neither the 2024 or 2025 data changed and therefore the 2025 the growth rate does not change.

keep_growth option set globally

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

This works because by default the global **Keep_growth** options was set to false. In that context, implementing **--kg** at the line level temporarily set the **keep_growth** flag to true for the specific line (and those following).

In the example below we set the **keep_growth** flag to True globally and then use **nkg** at the line level.

To set **keep_growth** to True globally enter it as a specific option for the update command , **keep_growth=True**.

In this context, 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(100.0,
                      index=[v for v in range(2020,2025)], # create row index
                      # equivalent to index=[2020,2021,2022,2023,2024]
                      columns=['A','B','C','D','E']) # create column
→name
dftest
```

| | A | B | C | D | E |
|------|-------|-------|-------|-------|-------|
| 2020 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 |
| 2021 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 |
| 2022 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 |
| 2023 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 |
| 2024 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 |

Note

In the below **.upd()** command

```
dfres = dftest.upd('''
<2022 2023> c = 200
<2022 2023> d = 300
<2022 2023> e = 400 --no_keep_growth
'',keep_growth=True) # <= Set keep_growth to True for the entirety of the
→command, # except for e where it is overridden by the --no_keep_
→growth flag
```

There are two **keep_growth** commands. The final one is the global option (global to the execution of this update). It is passed as an argument to the **.upd()** method “**,keep_growth=True**” and applies to every line in the command string (unless overridden). In contrast, the single line command **--no_keep_growth** is inside the string passed to **.upd()** and applies only to the line on which it occurs.

```

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

| | A | B | C | D | E |
|------|-------|-------|-------|-------|-------|
| 2020 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 |
| 2021 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 |
| 2022 | 100.0 | 100.0 | 200.0 | 300.0 | 400.0 |
| 2023 | 100.0 | 100.0 | 200.0 | 300.0 | 400.0 |
| 2024 | 100.0 | 100.0 | 200.0 | 300.0 | 100.0 |

| | 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 |

Scale option of the .upd() method

When running scenarios it can be useful to perform sensitivity analyses of model results, to better understand how the model responds when varying the intensity of a shock.

The scale option provides a mechanism for calculating a range of shocks as different proportions of the initially indicated one.

When using the scale option, scale=0 implies no change (effectively the baseline) while scale=0.5 is a scenario half of the full severity.

In the example below .upd() is executed three times for severity equals 0, 0.5 and 1. If the list passed to scale (named severity in this case) had five items in it, the update would be run five times – one time for each item in the list.

This example just prints outputs, a more interesting example would involve the solving a model using these different levels in a series of simulations.

```

dfinput=df.upd('A = 100')
print(f'input dataframe: \n{dfinput}\n\n')
for severity in [0,0.5,1]:
    # First make a dataframe with some growth rate
    res = dfinput.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:  
      Orig      A  
2020  100.000000  100.0  
2021  101.000000  100.0  
2022  103.020000  100.0  
2023  106.110600  100.0  
2024  110.355024  100.0  
2025  115.872775  100.0  
severity=0  
Dataframe:  
      Orig      A      B  
2020  100.000000  100.0   0.0  
2021  101.000000  100.0   0.0  
2022  103.020000  100.0   0.0  
2023  106.110600  100.0   0.0  
2024  110.355024  100.0   0.0  
2025  115.872775  100.0   0.0  
Growth:  
      Orig      A      B  
2020  NaN  NaN  NaN  
2021  1.0  0.0  NaN  
2022  2.0  0.0  NaN  
2023  3.0  0.0  NaN  
2024  4.0  0.0  NaN  
2025  5.0  0.0  NaN  
severity=0.5  
Dataframe:  
      Orig          A      B  
2020  100.000000  100.000000  0.0  
2021  101.000000  100.500000  5.0  
2022  103.020000  101.505000  5.0  
2023  106.110600  103.027575  5.0  
2024  110.355024  105.088126  5.0  
2025  115.872775  107.715330  5.0  
Growth:  
      Orig      A      B  
2020  NaN  NaN  NaN  
2021  1.0  0.5  inf  
2022  2.0  1.0  0.0  
2023  3.0  1.5  0.0  
2024  4.0  2.0  0.0  
2025  5.0  2.5  0.0  
severity=1  
Dataframe:  
      Orig          A      B  
2020  100.000000  100.000000  0.0  
2021  101.000000  101.000000  10.0  
2022  103.020000  103.020000  10.0  
2023  106.110600  106.110600  10.0  
2024  110.355024  110.355024  10.0  
2025  115.872775  115.872775  10.0  
Growth:  
      Orig      A      B  
2020  NaN  NaN  NaN  
2021  1.0  1.0  inf  
2022  2.0  2.0  0.0  
2023  3.0  3.0  0.0
```

(continues on next page)

(continued from previous page)

| | | | |
|------|-----|-----|-----|
| 2024 | 4.0 | 4.0 | 0.0 |
| 2025 | 5.0 | 5.0 | 0.0 |

Iprint option of the .upd() method prints pre- and post- update values update

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

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

| Update * [42.0, 44.0] 2021 2022 | | Before | After | Diff |
|---------------------------------|--|----------|-----------|-----------|
| A | | 100.0000 | 4200.0000 | 4100.0000 |
| 2021 | | 100.0000 | 4400.0000 | 4300.0000 |

7.4.6 The Create option of the .upd() determines behavior when a LHS variable does not exist

By default Create=True. As a result, in the examples above, when a left-hand side variable did not exist the .upd() commands created the previously undeclared variables.

To catch misspellings the parameter create can be set to False. When create=False, if update is executed on a variable that does not exist the specified variable(s) will not be created. Instead, an exception (an error) will be raised alerting to the user that the variable does not exist.

In the example below, the error is captured in a try catch statement (a python method for handling errors) so that the notebook will continue to run despite the error. If the try Catch had not been in place the notebook would stop executing and thrown the error.

Below the cell for reference is the error that would be thrown if the error had not been caught.

```
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:
```

Below the python error generated in the absence of the try / catch expression above.

Note that the most informative part of the message appears at the end, which is typical of most python errors. The preceding lines give a detailed listing of what steps were being executed when the error was generated which may or may not be of interest.

```
-----
Exception                                                 Traceback (most recent call last)
Cell In[15], line 1
----> 1 xx = df.upd('''
      2     # Same number of values as years
      3     <2021 2022> Aa * 42 44
      4     '',create=False)
File C:\WBG\Anaconda3\envs\ModelFlow\lib\site-packages\ModelFlow-1.0.8-py3.10.egg\
    modelclass.py:8342, in upd.__call__(self, updates, lprint, scale, create, keep_
    growth)
    8339 def __call__(self, updates, lprint=False, scale = 1.0, create=True, keep_
    growth=False,):
    8341     indf = self._obj
-> 8342     result = model.update(indf, updates=updates, lprint=lprint, scale =
    scale, create=create, keep_growth=keep_growth,)
    8343     return result
File C:\WBG\Anaconda3\envs\ModelFlow\lib\site-packages\ModelFlow-1.0.8-py3.10.egg\
    modelclass.py:1741, in Model_help_Mixin.update(indf, updates, lprint, scale, create,
    keep_growth)
    1738         multiplier = list(accumulate([(1+i) for i in growth_rate],operator.
    mul))
    1740 # print(varname,op,value,arg,sep=' ')
-> 1741     update_var(df, varname.upper(), op, value,time1,time2 ,
    1742                 create=create, lprint=lprint, scale = scale)
    1744 if update_growth:
    1745     lastvalue = df.loc[time2,varname]
File C:\WBG\Anaconda3\envs\ModelFlow\lib\site-packages\ModelFlow-1.0.8-py3.10.egg\
    modelhelp.py:40, in update_var(databank, xvar, operator, inputval, start, end,
    create, lprint, scale)
    38 if not create:
    39     errmsg = f'Variable to update not found:{var}, timespan = [{start} {end}]'
    40 \nSet create=True if you want the variable created: '
--> 40     raise Exception(errmsg)
    41 else:
    42     if 0:
Exception: Variable to update not found:AA, timespan = [2021 2022]
```

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

```
df
```

| | Orig |
|------|------------|
| 2020 | 100.000000 |
| 2021 | 101.000000 |
| 2022 | 103.020000 |
| 2023 | 106.110600 |
| 2024 | 110.355024 |
| 2025 | 115.872775 |

7.5 The `.mfcalc()` method. Return a dataframe with transformed variables.

Like `.upd()`, the `.mfcalc()` method of ModelFlow extends 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 later in this volume.

The purpose of `mfcalc()` is to perform quick and dirty calculations and modify dataframes.

7.5.1 Workspace initialization

Set up 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).

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 For more see here <https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html#pandas.DataFrame>
- 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(  
    100.000,  
    index=[v for v in range(2020,2026)],  
    columns=['A'])  
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 |

7.5.2 Create a new series from an existing series

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

The below call creates a new column x.

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

* Take care. Lags or leads in the equations, `mfcalc` run for 2021 to 2025

| | A | X |
|------|-------|-------|
| 2020 | 100.0 | 0.0 |
| 2021 | 100.0 | 100.0 |

(continues on next page)

(continued from previous page)

| | | |
|------|-------|-------|
| 2022 | 100.0 | 200.0 |
| 2023 | 100.0 | 300.0 |
| 2024 | 100.0 | 400.0 |
| 2025 | 100.0 | 500.0 |

⚠ Warning

By default `.mfcalc` will initialize a new variable with zeroes.

Moreover, if a formula passed to `.mfcalc` contains a lag, the result of the operation will be calculated for a row only if there is data in the series for the preceding row.

These two behaviors affect how calculations generated with `.mfcalc` are executed and can generate results that may sometimes be unexpected.

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

As with `.upd()` unless we assign the result of `.mfcalc()` to a variable the resulting dataframe is lost. The above did not change `df`.

```
df
```

| | A |
|------|-------|
| 2020 | 100.0 |
| 2021 | 100.0 |
| 2022 | 100.0 |
| 2023 | 100.0 |
| 2024 | 100.0 |
| 2025 | 100.0 |

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

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

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

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

* Take care. Lags or leads in the equations, `mfcalc` run for 2021 to 2025

| | 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 |

Note

As discussed before, mfcalc initiates a new variables with zeroes. The lag in 2020 of X is not defined so the calculation is actually run from 2021-2025. As a result, we have a zero in 2020, and then this number is increased by 100 for each following year.

7.5.4 Recalculate A so it grows by 2 percent

mfcalc() understands lagged variables and can do recursive calculations. Recall that if a lagged value does not exist the calculation will not be made for that period. Below this results in the warning:

*** Take care. Lags or leads in the equations, mfcalc results calculated for the period 2021 to 2025

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

* Take care. Lags or leads in the equations, mfcalc run for 2021 to 2025

| | 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 |

In this example, mfcalc() knows that it can not start to calculate in 2020, because 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 behavior differs from other programs or python methods (such as .pct_change() above) that might return a n/a value for the 2020 observation.

7.5.5 Display the normalization of an equation (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=True);
```

```
* Take care. Lags or leads in the equations, mfcalc run for 2021 to 2025
FRML <> A=EXP(LOG(A(-1))+0.02) $
```

Note

ModelFlow the expression `dlog(a)` refers to the difference in the natural logarithm $d\log(x_t) \equiv \ln(x_t) - \ln(x_{t-1})$ and is equal to the growth rate for the variable. The `dlog()` syntax is borrowed from EViews.

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

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

which expressed in the business logic language of ModelFlow is:

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

7.5.6 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 value of the variable passed to the `diff` operator. Thus, `diff(a)=x` normalizes to `a=a(-1)+x`

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

```
* Take care. Lags or leads in the equations, mfcalc run for 2021 to 2025
FRML <> A=A(-1)+(2) $
```

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

7.5.7 mfcalc with several equations and arguments

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

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

For example:

```
res = df.mfcalc('''
diff(a) = 2
x = a + 42
''')
res
# use res.diff() to see the difference
```

* Take care. Lags or leads in the equations, mfcalc run for 2021 to 2025

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

In this example the variable A in the DataFrame df was initialized to 100 for the period 2020 through 2025.

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

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

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

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

Compare the results above with the results (below) when the same calculations are undertaken in two separate calls to .mfcalc().

```
res1 = df.mfcalc('''
diff(a) = 2
''')
res2 = res1.mfcalc('''
x = a + 42
''')
res2
```

* Take care. Lags or leads in the equations, mfcalc run for 2021 to 2025

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

Danger

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

7.5.8 Setting a time frame with `mfcalc`.

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

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

Note

like `.upd()` time frames set in one line are inherited by subsequent lines unless reset explicitly.

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

Using modelflow with World Bank models

CHAPTER
EIGHT

USING MODELFLOW WITH WORLD BANK MODELS

The ModelFlow python package has been developed to solve a wide range of models, see the ModelFlow github <https://github.com/IbHansen/ModelFlow> web site for working examples of the Solow Model, the United States' Federal Reserve FRB/US 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 using the EViews Model Object for simulation. Although derived from these EViews models, the models available on the World Bank web site and through the mechanisms outlined below are **pure python** models and use ModelFlow and various python libraries for solution visualization and data management.

This chapter illustrates how to access these models. Subsequent chapters explain how:

- to load ModelFlow models into the Anaconda environment on your computer
- to explore various features of these models (Chapter 9)
- World Bank behavioural equations are designed and manipulated in ModelFlow (Chapter 10)
- to perform a variety of simulations (Chapters 10, 11 and 12)
- to extract and compare results from your simulations (Chapter 13).

In this chapter - Using ModelFlow with World Bank Models

This chapter demonstrates how to access World Bank models that have been made publicly available using the ModelFlow python system.

- **Available Models:**

- Access a range of pre-built World Bank models tailored for macroeconomic analysis.
- Publicly available models can be loaded and customized in ModelFlow.

8.1 Publicly available World Bank models for use with ModelFlow

Several World Bank macrostructural models are currently available for download and use with ModelFlow (more will be added over time). The available models include:

- Bolivia (Annual model)
- Croatia (Quarterly model)
- Iraq (Annual model)
- Nepal (Annual model with Climate features)

- Pakistan (Annual model with climate features)
- Turkiye (Annual model with climate features)

Each of these models has been developed as part of the outreach work of the World Bank. The basic modeling framework of each of these models is outlined in Burns *et al.* (2019) with individual models having specific extensions reflecting features of the individual country and the questions for which the model was designed to respond. The approach for the models with climate features is laid out in Burns *et al.* (2021), although several additional features are included in some of these models that have not yet been documented.

As additional models are released they will be made available using the mechanism described below.

8.2 How to access the models the `.Worldbank_Models()` method

The World Bank models can be downloaded on to a user's computer using the `model.Worldbank_Models()` method, which downloads the models from the World Bank's GitHub repository at <https://github.com/worldbank/MFMod-ModelFlow>.

By default, the models are downloaded to a directory on the user's computer entitled `WorldbankModels` that is located just below the directory from which it is executed. The content of the downloaded folders will be displayed in a table of content, and the user can access the notebooks by clicking on them.

```
model.Worldbank_Models()
```

Avaible notebooks

Bolivia

[Bolivia readme](#)

[StandardShocks](#)

Croatia

[Croatia Readme](#)

[StandardShocks](#)

Iraq

[Iraq ReadMe](#)

[StandardShocks](#)

Nepal

[Nepal readme](#)

[StandardShocks](#)

Pakistan

[Pakistan readme](#)

[StandardShocks](#)

Türkiye

[StandardShocks](#)

[Untitled](#)

8.2.1 WorldbankModels() Options

The `model.Worldbank_Models()` method includes several options, which affect the way the download of the models is executed.

The `destination` option

To change the directory to which the files are downloaded the `destination` parameter can be set (by default it points to `./WorldBankModels`).

`model.Worldbank_Models('./mydirectory')` (or equivalently `model.Worldbank_Models(destination='./mydirectory')`) would instead download the GitHub repository to a directory called `mydirectory` below the location from which the command was executed. Note: For security reasons, the `destination` option will not accept a location that is above the directory from which it is executed.

The `silent` option

By default the `silent` option is set to `True`. Setting it to `False` generates a more verbose indication of the directories and files downloaded and erased (of any).

Options controlling the treatment of existing files in the download directory

| option | de-
fault | Level of
aggres-
sion | Action |
|---------|--------------|-----------------------------|---|
| replace | False | Medium | replace=True will replace any file in the local store that also exists on the World Bank github site with the version on the World Bank site. |
| replace | False | Lowest | replace=False will not change or delete any files on the user's store. Only files that exist on the World Bank web site (and not on the local copy) will be downloaded. |

To replace the all of the content first delete the destination directory (by default `./WorldBankModels`). <to do this use the operation system.

Options to change the repository that is downloaded

It is possible to choose a repository other than the World Bank site to be downloaded, although the use case for this option is limited. A user that wishes to do this can specify the repository to be downloaded by setting the following options

| option | Default value | Explanation |
|-----------|-----------------|---|
| owner | WorldBank | Name of the owner of the github repository to be downloaded |
| repo_name | MFMod-ModelFlow | The name of the repository |
| branch | main | The branch to download. |

8.3 .display_toc() method

The `.display_toc()` method can be used to displays a list of the directories and files included in the subfolder to which the World Bank models were previously downloaded.

```
model.display_toc(folder='WorldbankModels');
```

Avaible notebooks

Bolivia

[Bolivia readme](#)

[StandardShocks](#)

Croatia

[Croatia Readme](#)

[StandardShocks](#)

Iraq

[Iraq ReadMe](#)

[StandardShocks](#)

Nepal

[Nepal readme](#)

[StandardShocks](#)

Pakistan

[Pakistan readme](#)

[StandardShocks](#)

Türkiye

[StandardShocks](#)

[Untitled](#)

Help on the accessing World Bank models can be generated by using the `help()` function.

```
help(model.Worldbank_Models)
```

```
Help on function Worldbank_Models in module modelclass:  
Worldbank_Models(owner: str = 'worldbank', repo_name: str = 'MFMod-ModelFlow',  
↳branch: str = 'main', destination='./WorldbankModels', go=True, silent=True,  
↳replace=False)  
    Download an entire GitHub repository and extract it to a specified location.  
Parameters:  
    owner: The owner of the GitHub repository.  
    repo_name: The name of the repository.  
    branch: The branch to download.  
    destination: The local path where the repository should be extracted.  
    go: display toc of notebooks  
    silent: keep silent  
Returns:  
    A message indicating whether the download was successful or not.
```

WORKING WITH A WORLD BANK MODEL 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 preceding discussions of ModelFlow features.

Process:

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

9.1 Prepare the work space

To use ModelFlow it must be installed as per the instructions in Chapter 3 (a one-time operation). The the python environment into which ModelFlow was installed must be activated. For users that installed ModelFlow according to the earlier instructions this can be achieved by executing `conda activate ModelFlow` in line with earlier instructions.

Once the python environment is activated, the ModelFlow and pandas packages must be imported into your workspace. Once this is done (accomplished by the code below), the user is ready to work with ModelFlow.

In this chapter - Working with a World Bank Model under ModelFlow

This chapter provides practical guidance on using World Bank models within the ModelFlow framework.

Key points include:

- **Setup and Preparation:**

- Prepare the workspace by setting up the Python environment.
 - Load the model, associated data, and variable descriptions.

- Model Exploration:**

- - Extract information about the model, including:
 - * its equations
 - * its variables
 - * its structure

* its data

Organize variables into groups for streamlined exploration and analysis.

• Key Methods:

- Use ModelFlow's built-in functions to query, modify, and analyze data.
- Explore relationships between variables and their role in the model.

```
# Prepare the notebook for use of ModelFlow
# Jupyter magic command to improve the display of charts in the Notebook
%matplotlib inline
# Import pandas
import pandas as pd
# Import the model class from the modelclass module
from modelclass import model
# functions that improve rendering of ModelFlow outputs
model.widescreen()
model.scroll_off();
```

9.2 Load the model: Load a pre-existing model, data and descriptions

To load a model use the `model.modelload()` method of ModelFlow. In the example below, the model has been saved to the models folder located one level above the directory from which the Jupyter Notebook has been executed (but within the scope of Jupyter itself, i.e. below the directory from which the Jupyter system was launched).

9.2.1 The `.modelload()` method

The command below

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

instantiates (creates an instance of) a ModelFlow `model` object using the `model` object (equiations and data) contained in the file `pak.pcim` and assigns it to the variable name `mpak`.

The `run=1` option executes the model and assigns the result of the model execution to the DataFrame `bline`.

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

The `keep` option instructs ModelFlow to maintain in the `model` object (`mpak`) the results of the initial scenario, assigning it the text name `Baseline`. As written, `modelload` returns both the `model` object `mpak`, but also a DataFrame `bline` that is assigned the results of the simulation. This DataFrame is distinct from the one that is stored inside the `mpak` `model` object by the `keep=` command, although the data inside each of these DataFrames will have the same numerical values. The `keep` option is described in more detail in the following chapter on scenarios.

Warning

If ModelFlow cannot find the file at the position indicated it will look for it in the global Model repository on line.

Upon return, the `modelload` command indicates the location from which the model was retrieved. In this case, from the requested local file store.

```
#Replace the path below with the location of the pak.pcim file (or some other world
→bank model file) on your computer
mpak,bline = model.modelload('../models/pak.pcim', \
                           alfa=0.7,run=1,keep= 'Baseline')
```

Zipped file read: ..\models\pak.pcim

9.2.2 Extracting information about the model

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

A World Bank model in ModelFlow contains a wide range of objects.

- variables – time series variables comprised of mnemonics and data
- dataframes – data for each variable generated in different simulations
- groups – lists of variables
- equations – identities and behavioral
- model – the model object itself

Extracting information about each of these objects is central to working with WBG models in ModelFlow.

The model object contains information about the model itself, its name, its structure (does it contain simultaneous equations or is it recursive), the number of variables it contains and the number that are exogenous and endogenous (have associated equations). Executing the unadorned name of a model object, i.e. `mpak` displays summary information about the model object.

`mpak`

```
<
Model name           :          PAK
Model structure      :    Simultaneous
Number of variables  :          839
Number of exogeneous variables :          461
Number of endogeneous variables :          378
>
```

The model work space also has a time dimension, its sample period. This can be retrieved and changed.

`mpak.current_per`

```
Index([2016, 2017, 2018, 2019, 2020, 2021, 2022, 2023, 2024, 2025, 2026, 2027,
       2028, 2029, 2030, 2031, 2032, 2033, 2034, 2035],
      dtype='int64')
```

Here the model is currently set up to solve over the period 2016 through 2030. That period can be changed assuming, as is the case with the Pakistan model, that additional data are available.

9.2.3 Information about variables

The model object mpak contains lists of all the variables that form part of the model, and these lists can be interrogated to garner information about the model. The Table below indicates some of the most important of these queries. The variables for which information is sought can be specified directly or through a wildcard specification (see note).

| Method | Example | Information returned |
|------------------|---------------------------------------|---|
| .names | mpak
['PAKNECON*XN']
.name | A python list of the mnemonics of all the variables defined and contained in the model object that match the search paremeters in the [] |
| .des | mpak
['PAKNECONPRVT?
N'] .des | A dictionary of mnemonics and their variable descriptions |
| .<var name>.show | mpak.
PAKNECONPRVTXN.
show | Lists the equation (formula), variable descriptions and data values of a specific variable |

Note

Wildcards

Most of the variable and equation information commands accept wildcard specifications in the search parameter.

The * character in the command mpak ['PAKNECON*XN'].names example is a wildcard character and the expression will return all variables that begin PAKNECON and end XN.

The ? in the .des example is another wildcard expression. It will match only single characters. Thus mpak ['PAKNECONPRVT?N'].names would return three variables: PAKNECONPRVTKN, PAKNECON-PRVTXN, and PAKNECONPRVTXN. The real, current value, and deflators for household consumption expenditure.

Note the final show example uses a slightly different syntax where the variable to be operated upon is specified directly: modelname.PAKNECONPRVTXN.show.

The example below returns the mnemonics and descriptions of all variables matching the pattern PAKNYGDP*KN, i.e. Pakistani variables (PAK) from the National Income Accounts (NY) from the main sub-category GDP that are also real (K) expressed in local currency units (N) variables.

Box 3. World Bank Mnemonics

A typical World Bank model will have in excess of 300 variables. Each has a mnemonic typically comprised of 14 characters that is structured in a specific way. The root for almost all variables is the three letters of the ISO code for the country to which the variable pertains. Other letters describe the variable in ever finer detail (see below).

12345678901234

CCC₁AA₂MMM₃NNNN₄U₅C

where:

| Letters | Meaning |
|---------|---|
| CCC | The three-letter 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 (see following Table for more info) |
| MMM | The three-letter major sub-category of the data - i.e. GDP, EXP - expenditure |
| NNNN | The four-letter minor sub-category MKTP for market prices |
| U | The measure (K: real variable; C: Current Values; X: Prices) |
| C | denotes the Currency (N: National currency; D: USD; P: PPP) |

Common major accounting systems mnemonics:

The, AA from above 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 |

Less common terminations: Occasionally you will see variables with and ‘_’ appended to the name. This indicates that the variable is being expressed as a percent of something (usually GDP). Thus PAKBNNCABFUND_CD_ means Pakistan (PAK) Balance of Payments, net (BN) of the Current Account (CAB) IMF definition (FUND) in Current (C) Dollars (D) expressed as a percent of GDP.

Others less common terminations include ER (Effective rate) and (SR) Statutory rate used to denote the average tax rate (ER) of a tax versus the legal rate (SR).

Thus:

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

If executed, the command `mpak ['*'].des` would return a dictionary of all the mnemonics and descriptions of all the variables in the mpak model object.

The below command is more restrictive and returns only the variables that start PAKNYGDP and KN.

```
mpak [ 'PAKNYGDP*KN' ].des
```

```
PAKNYGDPDISCKN : GDP Disc., 2000 LCU mn
PAKNYGDPFCSTKN : GDP Factor Cost Local Currency units Volumes National base year
PAKNYGDPMKTPKN : Real GDP
PAKNYGDPPOTLKN : Potential Output, constant LCU
```

The ! operator – searching on the variable description

The ! operator allows the same methods to be used to retrieve information about variables, but based on their descriptions. Pre-pending the search string with the ! operator, tells it to try and match (and display) information about variables based on their descriptions not their mnemonics.

Note

The ! operator

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

The below expression returns the mnemonics of all variables whose description includes the word Carbon.

```
mpak [ '!*Carbon*' ].names
```

```
[ 'PAKCCEMISCO2TKN', 'PAKGGREVCO2CER', 'PAKGGREVCO2GER', 'PAKGGREVCO2OER' ]
```

The following expression returns the mnemonics and descriptions of the same variables.

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

```
PAKCCEMISCO2TKN : Total Carbon emissions (tons)
PAKGGREVCO2CER : Carbon tax on coal (USD/t)
PAKGGREVCO2GER : Carbon tax on gas (USD/t)
PAKGGREVCO2OER : Carbon tax on oil (USD/t)
```

The following expression returns the descriptions of a specific variable.

```
mpak.var_description['PAKGGREVCO2OER']
```

```
'Carbon tax on oil (USD/t)'
```

9.3 Groups

ModelFlow incorporates a variant of the idea of groups from EViews. In ModelFlow the groups defined in an imported EViews workfile are converted into entries in a dictionary called `var_groups` which can be interrogated, added to and amended like any dictionary in python.

The command `mpak.var_groups` will return all of the groups already defined in mpak.

```
mpak.var_groups
```

```
{'Headline': '{cty}NYGDPMKTPKN {cty}NYGDPMKTPXN {cty}NRTOTLCN {cty}LMUNRTOTLCN
↳{cty}BFFINCABCD {cty}BFBOPTOTLCD {cty}GGBALEXGRCN {cty}GGDBTTOTLCN_ {cty}
↳GGDBTTOTLCN {cty}BNCA BLOC LCD_ {cty}FPCPITOTLXN {cty}CCEMISCO2TKN',
'National income accounts': '{cty}NY*',
'National expenditure accounts': '{cty}NE*',
'Value added accounts': '{cty}NV*',
'Balance of payments exports': '{cty}BX*',
'Balance of payments exports and value added ': '{cty}BX* {cty}NV*',
'Balance of Payments Financial Account': '{cty}BF*',
'General government fiscal accounts': '{cty}GG*',
'World all': 'WLD*',
'All variables': '*'}
```

A group can be added to the dictionary by giving it a unique identifier (key) and associating with it a string defining the group, using a wildcard specification or just a space de-limited list of mnemonics.

Thus the first command below will generate a new group called 'MyGroup' that contains all variables beginning PAKG-GREV and ending CN, plus the variable PAKGGBALOVRLCN to the dictionary `var_groups` that is part of the model object `mpak`. The second creates a group called LaborMarket which contains the variables for Employment and the Unemployment rate.

```
mpak.var_groups['MyGroup']= 'PAKGGREV*CN PAKGGBALOVRLCN'
mpak.var_groups['LaborMarket']= 'PAKLMEMPTOTLCN PAKLMUNRTOTLCN'
```

9.3.1 The # operator – searching on the variable description

The # operator allows the same methods to be used to retrieve information about groups. Pre-pending the search string with the # operator, tells it to try and match (and display) information about the variables in groups that match the search expression following the #.

Note

The # operator

If a wildcard is preceded by an exclamation mark # the search will be done over the groups in the model object and will return information about the members of the returned groups

The below expression returns the mnemonics of all variables that are a member of the MyGroup Group.

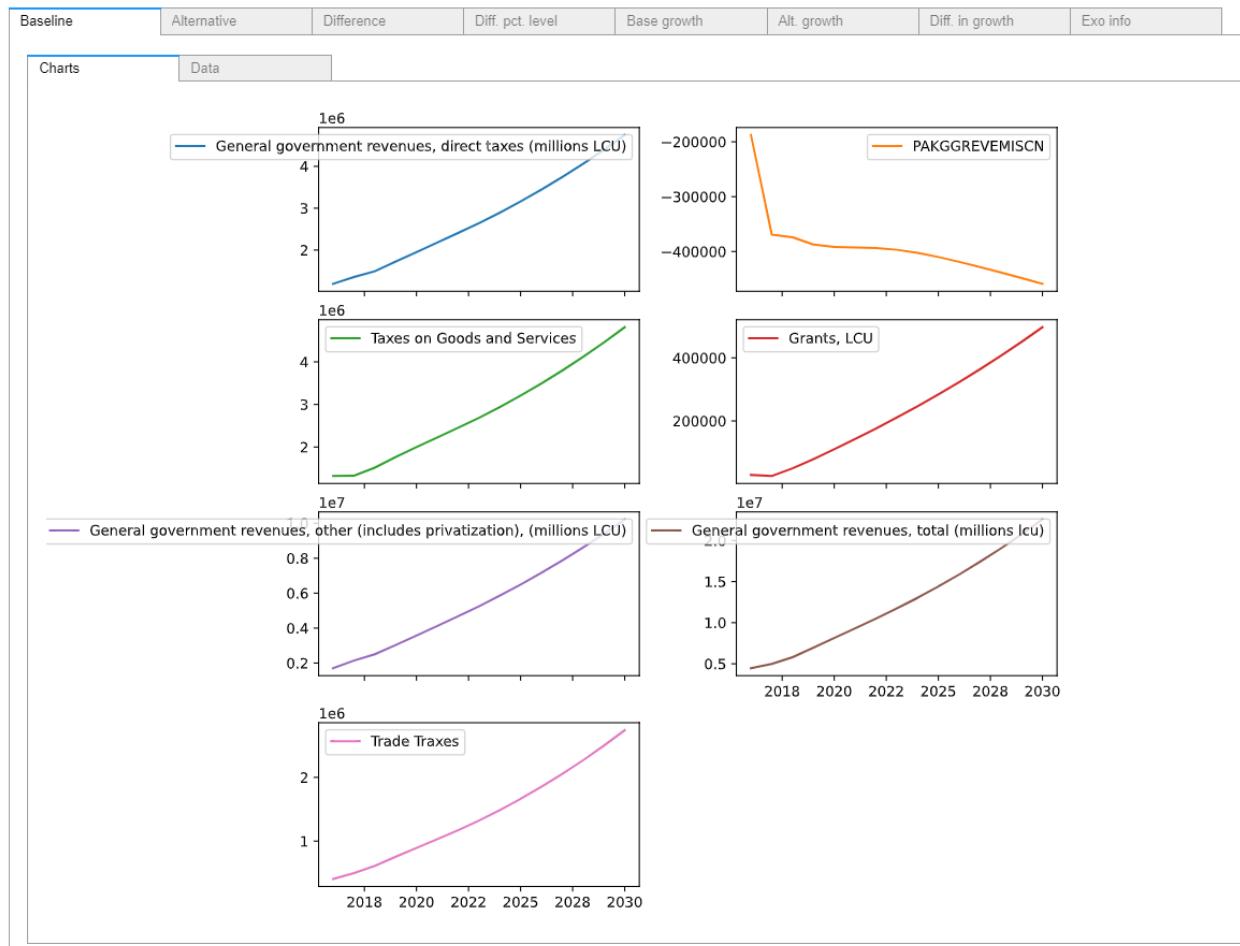
```
mpak ['#MyGroup'].names
```

```
[ 'PAKGREVDRCTCN',
  'PAKGREVEMISCN',
  'PAKGREVGNFSCN',
  'PAKGREVGRNTCN',
  'PAKGREVOOTHRCN',
  'PAKGREVTOTLCN',
  'PAKGREVRDECN',
  'PAKGGBALOVRLCN' ]
```

9.3.2 Information about the data of series in a group

The unadorned command `mpak [#MyGroups]` invokes a widget that shows all of the data in the group `MyGroup` and various representations (level and growth rates) both as tables and charts.

```
mpak ['#MyGroup']
```



Alternatively just the graphs and or tables can be returned, by appending the `.df` method (tables) or `.plot()` methods (charts). Modifying the command further by including the `.pct` command would display the data as growth rates.

```
mpak ['#MyGroup'].df
```

| | PAKGGREVDRCTCN | PAKGGREVEMISCN | PAKGGREVGNFSCN | PAKGGREVGRNTCN | \ |
|------|----------------|----------------|----------------|----------------|---|
| 2016 | 1.192249e+06 | -187487.145911 | 1.319524e+06 | 28696.665825 | |
| 2017 | 1.354036e+06 | -368998.340503 | 1.327860e+06 | 25349.188671 | |
| 2018 | 1.492389e+06 | -373989.822891 | 1.516485e+06 | 49838.249131 | |
| 2019 | 1.721883e+06 | -387328.506687 | 1.764865e+06 | 78978.628390 | |
| 2020 | 1.950849e+06 | -391591.258722 | 1.998747e+06 | 110163.222890 | |
| 2021 | 2.178938e+06 | -392424.135886 | 2.225111e+06 | 142678.758699 | |
| 2022 | 2.407303e+06 | -393527.268710 | 2.450129e+06 | 176071.658354 | |
| 2023 | 2.644233e+06 | -396768.489162 | 2.685256e+06 | 210616.945645 | |
| 2024 | 2.894933e+06 | -402416.006307 | 2.936720e+06 | 246606.677324 | |
| 2025 | 3.161598e+06 | -409971.057281 | 3.206328e+06 | 284195.083320 | |
| 2026 | 3.444363e+06 | -418754.040372 | 3.493313e+06 | 323384.807252 | |
| 2027 | 3.743124e+06 | -428241.872484 | 3.796738e+06 | 364156.630890 | |
| 2028 | 4.058704e+06 | -438151.424003 | 4.116861e+06 | 406583.540873 | |
| 2029 | 4.393195e+06 | -448385.655807 | 4.455458e+06 | 450879.142260 | |
| 2030 | 4.749689e+06 | -458938.010233 | 4.815425e+06 | 497380.152375 | |
| 2031 | 5.131800e+06 | -469817.783803 | 5.200188e+06 | 546498.760163 | |
| 2032 | 5.543290e+06 | -481016.427390 | 5.613280e+06 | 598678.767743 | |
| 2033 | 5.987899e+06 | -492506.888551 | 6.058177e+06 | 654372.159746 | |
| 2034 | 6.469364e+06 | -504258.235259 | 6.538358e+06 | 714036.436715 | |
| 2035 | 6.991538e+06 | -516250.612903 | 7.057461e+06 | 778144.673255 | |
| | PAKGGREVOHRCN | PAKGGREVTOTLCN | PAKGGREVTRDECN | PAKGBALOVRRLCN | |
| 2016 | 1.704982e+06 | 4.463113e+06 | 4.051485e+05 | -1.322586e+06 | |
| 2017 | 2.141104e+06 | 4.976845e+06 | 4.974946e+05 | -1.833428e+06 | |
| 2018 | 2.505467e+06 | 5.802066e+06 | 6.118760e+05 | -1.814775e+06 | |
| 2019 | 3.033525e+06 | 6.967846e+06 | 7.559233e+05 | -1.764188e+06 | |
| 2020 | 3.574406e+06 | 8.136424e+06 | 8.938484e+05 | -1.798450e+06 | |
| 2021 | 4.122857e+06 | 9.307621e+06 | 1.030460e+06 | -1.857821e+06 | |
| 2022 | 4.677542e+06 | 1.048922e+07 | 1.171700e+06 | -1.939507e+06 | |
| 2023 | 5.252367e+06 | 1.171840e+07 | 1.322695e+06 | -2.033015e+06 | |
| 2024 | 5.856855e+06 | 1.301870e+07 | 1.486000e+06 | -2.146245e+06 | |
| 2025 | 6.495230e+06 | 1.439974e+07 | 1.662364e+06 | -2.279101e+06 | |
| 2026 | 7.167704e+06 | 1.586160e+07 | 1.851585e+06 | -2.433659e+06 | |
| 2027 | 7.874000e+06 | 1.740321e+07 | 2.053431e+06 | -2.610006e+06 | |
| 2028 | 8.615802e+06 | 1.902797e+07 | 2.268174e+06 | -2.806904e+06 | |
| 2029 | 9.397577e+06 | 2.074544e+07 | 2.496720e+06 | -3.022765e+06 | |
| 2030 | 1.022607e+07 | 2.257011e+07 | 2.740488e+06 | -3.256688e+06 | |
| 2031 | 1.110928e+07 | 2.451915e+07 | 3.001198e+06 | -3.508801e+06 | |
| 2032 | 1.205564e+07 | 2.661059e+07 | 3.280716e+06 | -3.780096e+06 | |
| 2033 | 1.307361e+07 | 2.886255e+07 | 3.580994e+06 | -4.072057e+06 | |
| 2034 | 1.417167e+07 | 3.129327e+07 | 3.904098e+06 | -4.386355e+06 | |
| 2035 | 1.535858e+07 | 3.392175e+07 | 4.252273e+06 | -4.724724e+06 | |

Below the same logic is used to display the data from variables matching a mnemonic search. The results have been placed inside a `with mpak.set_smpl()` clause to restrict the output to a shorter period. If it was not used the output would cover the whole time period of the `.lastdf` DataFrame from which all of these data are drawn.

Note

When using a `with` clause, an explicit print statement is required.

```
with mpak.set_smpl(2020,2025):
    print(round(mpak['#MyGroup'].pct.df,2)) # round restricts the display to 2 decimal points
```

| | PAKGGREVDRCTCN | PAKGGREVEMISCN | PAKGGREVGNFSCN | PAKGGREVGRNTCN | \ |
|------|----------------|----------------|----------------|----------------|---|
| 2020 | 13.30 | 1.10 | 13.25 | 39.48 | |
| 2021 | 11.69 | 0.21 | 11.33 | 29.52 | |
| 2022 | 10.48 | 0.28 | 10.11 | 23.40 | |
| 2023 | 9.84 | 0.82 | 9.60 | 19.62 | |
| 2024 | 9.48 | 1.42 | 9.36 | 17.09 | |
| 2025 | 9.21 | 1.88 | 9.18 | 15.24 | |
| | PAKGGREVOTHRCN | PAKGGREVOTLCN | PAKGGREVTRDECN | PAKGGBALOVRLCN | |
| 2020 | 17.83 | 16.77 | 18.25 | 1.94 | |
| 2021 | 15.34 | 14.39 | 15.28 | 3.30 | |
| 2022 | 13.45 | 12.69 | 13.71 | 4.40 | |
| 2023 | 12.29 | 11.72 | 12.89 | 4.82 | |
| 2024 | 11.51 | 11.10 | 12.35 | 5.57 | |
| 2025 | 10.90 | 10.61 | 11.87 | 6.19 | |

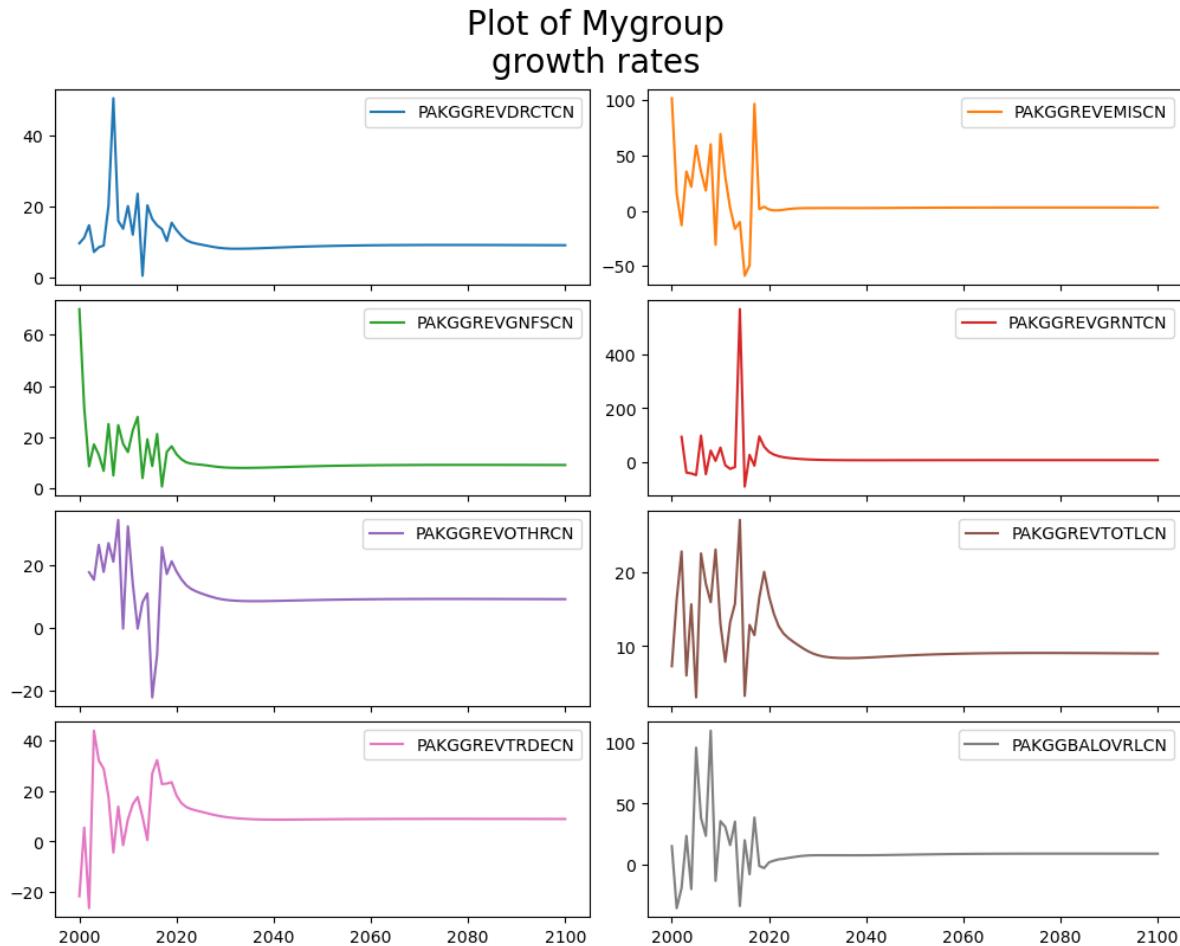
When displaying a dataframe or a manipulation of a dataframe in cases where the output might include very many lines of output, Jupyter will, by default, truncate the output by showing the first and last five observations of the active sample period when the same call is made without the with clause.

```
mpak.smpl(2000,2100) # change the default view to cover 100 observations
round(mpak['#MyGroup'].pct.df,2) #Jupyter will truncate the output
```

| | PAKGGREVDRCTCN | PAKGGREVEMISCN | PAKGGREVGNFSCN | PAKGGREVGRNTCN | \ |
|------------------------|----------------|----------------|----------------|----------------|---|
| 2000 | 9.55 | 101.83 | 70.02 | NaN | |
| 2001 | 11.14 | 15.37 | 31.46 | inf | |
| 2002 | 14.66 | -13.23 | 8.55 | 94.82 | |
| 2003 | 7.11 | 35.47 | 17.12 | -36.96 | |
| 2004 | 8.43 | 21.64 | 13.05 | -39.98 | |
| ... | ... | ... | ... | ... | |
| 2096 | 9.03 | 2.85 | 9.07 | 9.03 | |
| 2097 | 9.02 | 2.84 | 9.06 | 9.02 | |
| 2098 | 9.02 | 2.84 | 9.06 | 9.02 | |
| 2099 | 9.01 | 2.84 | 9.06 | 9.01 | |
| 2100 | 9.01 | 2.84 | 9.05 | 9.01 | |
| | PAKGGREVOTHRCN | PAKGGREVOTLCN | PAKGGREVTRDECN | PAKGGBALOVRLCN | |
| 2000 | NaN | 7.30 | -21.68 | 15.14 | |
| 2001 | inf | 16.34 | 5.52 | -35.57 | |
| 2002 | 17.59 | 22.84 | -26.44 | -19.00 | |
| 2003 | 15.20 | 6.04 | 43.96 | 23.55 | |
| 2004 | 26.30 | 15.68 | 32.11 | -19.92 | |
| ... | ... | ... | ... | ... | |
| 2096 | 9.03 | 9.03 | 8.96 | 9.03 | |
| 2097 | 9.02 | 9.03 | 8.96 | 9.03 | |
| 2098 | 9.02 | 9.02 | 8.95 | 9.02 | |
| 2099 | 9.01 | 9.02 | 8.95 | 9.02 | |
| 2100 | 9.01 | 9.01 | 8.95 | 9.01 | |
| [101 rows x 8 columns] | | | | | |

9.3.3 Display data from a group graphically

```
mpak['#MyGroup'].pct.plot(title="Plot of Mygroup\ngrowth rates");
```



9.4 Information about equations

Information about specific equations can also be extracted and displayed.

9.4.1 The `endogene` property

The `endogene` property returns a list of all variables in the model that are endogenous (have an equation). It can also be used to test whether a specific mnemonic has an equation associated with it.

The `endogene` property returns a list. For brevity only the first 5 elements are show below.

```
sorted(mpak.endogene) [:5]
```

```
['CHNEXR05', 'CHNPCEXN05', 'DEUEXR05', 'DEUPCEXN05', 'FRAEXR05']
```

The expression '`PAKNECONPRVTKN`' in `mpak.endogene` returns True if the passed mnemonic is in the list returned by `mpak.endogene`.

```
'PAKNECONPRVTKN' in mpak.endogene
```

```
True
```

9.4.2 Retrieving info on equations

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

| Command | Effect |
|----------------------------------|--|
| mpak ['PAKNECONPRVTKN'].frml | Returns a normalized version of the equation (the one actually used in ModelFlow) |
| mpak ['PAKNECONPRVTKN'].eviews | In models imported from Eviews, reports the original eviews specification |
| mpak.PAKNECONPRVTXN.show | Displays the equation (formula); variable descriptions; and variable values. |

9.4.3 The .eviews method

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

```
mpak [ 'PAKNECONPRVTKN' ].eviews
```

```
PAKNECONPRVTKN :
DLOG(PAKNECONPRVTKN) == 0.2*(LOG(PAKNECONPRVTKN(-1)) - LOG(1.21203101101442) -_
LOG(((PAKBFSTREMTCD(-1) - PAKBMFSTREMTCD(-1))*PAKPANUSATLS(-1)) +_
PAKGEXPTRNSCN(-1) + PAKNYYWBTO TLCN(-1)*(1 - PAKGGREVDRCTXN(-1)/100))/_
PAKNECONPRVTXN(-1)) + 0.763938860758873*DLOG(((PAKBFSTREMTCD -_
PAKBMFSTREMTCD)*PAKPANUSATLS) + PAKGGEXPTRNSCN + PAKNYYWBTO TLCN*(1 -_
PAKGGREVDRCTXN/100))/PAKNECONPRVTXN) - 0.0634474791568939*DURING("2009") - 0.3*(PAKFMLBLPOLYXN/100 - DLOG(PAKNECONPRVTXN))
```

9.4.4 The .frml property

The .frml method returns the normalized equation that is actually used in ModelFlow.

In this instance the variable to be displayed is referenced directly (not as the result of a search operation ['partial*variablename'] syntax).

In addition to the equation of the variable, The .frml method also returns a long-text description of all the variables in the equation (assuming that one was defined for each variable), followed by a listing of all the dependent variables of the equation and their descriptions (below, the DURING_2019 variable has had no description defined so it returns a blank).

```
mpak.PAKNECONPRVTXN.frml
```

```

Endogeneous: PAKNECONPRVTKN: HH. Cons Real
Formular: FRML <DAMP,STOC> PAKNECONPRVTKN = (PAKNECONPRVTKN(-1)*EXP(PAKNECONPRVTKN_-
˓→A+ (-0.2*(LOG(PAKNECONPRVTKN(-1))-LOG(1.21203101101442)-LOG(((PAKBXFSTREMTCD(-
˓→1)-PAKBMFSTREMTCD(-1))*PAKPANUSATLS(-1))+PAKGGELEXTRNSCN(-1)+PAKNYYWBTOTLCN(-
˓→1)*(1-PAKGGREVDRCTXN(-1)/100))/PAKNECONPRVTXN(-1)))+0.
˓→763938860758873*((LOG(((PAKBXFSTREMTCD-
˓→PAKBMFSTREMTCD)*PAKPANUSATLS)+PAKGGELEXTRNSCN+PAKNYYWBTOTLCN*(1-PAKGGREVDRCTXN/-
˓→100))/PAKNECONPRVTXN))-(LOG(((PAKBXFSTREMTCD(-1)-PAKBMFSTREMTCD(-
˓→1))*PAKPANUSATLS(-1))+PAKGGELEXTRNSCN(-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
PAKGGELEXTRNSCN : Current Transfers
PAKGGREVDRCTXN : Direct Revenue Tax Rate
PAKNECONPRVTKN_A: Add factor:HH. Cons Real
PAKNECONPRVTKN_D: Fix dummy:HH. Cons Real
PAKNECONPRVTKN_X: Fix value:HH. Cons Real
PAKNECONPRVTXN : Implicit LCU defl., Pvt. Cons., 2000 = 1
PAKNYYWBTOTLCN : Total Wage Bill
PAKPANUSATLS : Exchange rate LCU / US$ - Pakistan

```

9.4.5 The .show method

The .show method returns:

1. The description of the variable
2. The normalized equation that is actually used in ModelFlow.
3. A listing of the mnemonics and descriptions of the RHS variables
4. The data of that variable (drawn from the basedf and .lastdf DataFrames in the model object as well as the data of the RHS variables of the equation from both the basedf and .lastdf DataFrames.

```

mpak.smpl(2020,2025) #change the actual sample range to limit the number of columns
˓→displayed
mpak.PAKNECONPRVTKN.show

```

The World Bank's MFMod Framework in Python with Modelflow

Endogeneous: PAKNECONPRVTKN: HH. Cons Real
 Formular: FRML <DAMP,STOC> PAKNECONPRVTKN = (PAKNECONPRVTKN(-1)*EXP(PAKNECONPRVTKN_A+ (-0.2*(LOG(PAKNECONPRVTKN(-1))-LOG(1.21203101101442)-LOG(((PAKBFSTREMTCD(-1)-PAKBFSTREMTCD(-1))*PAK PANUSATLS(-1))+PAKGEXPTRNSCN(-1)+PAKNYYWBOTLCN(-1)*(1-PAKGREVDRCTXN(-1)/100))/PAKNECONPRVTXN(-1))+0.763938860758873*((LOG(((PAKBFSTREMTCD-PAKBFSTREMTCD)*PAK PANUSATLS)+PAKGEXPTRNSCN+PAKNYYWBOTLCN*(1-PAKGREVDRCTXN/100))/PAKNECONPRVTXN))-LOG(((PAKBFSTREMTCD(-1)-PAKBFSTREMTCD(-1))*PAK PANUSATLS(-1))+PAKGEXPTRNSCN(-1)+PAKNYYWBOTLCN(-1)*(1-PAKGREVDRCTXN/100))/PAKNECONPRVTXN)))-(LOG(((PAKBFSTREMTCD(-1)-PAKBFSTREMTCD(-1))*PAK PANUSATLS(-1))+PAKGEXPTRNSCN(-1)+PAKNYYWBOTLCN(-1)*(1-PAKGREVDRCTXN/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 :
 PAKBFSTREMTCD : Imp., Remittances (BOP), US\$ mn
 PAKBFSTREMTCD : Exp., Remittances (BOP), US\$ mn
 PAKFMLBLPOLYXN : Key Policy Interest Rate
 PKGGEXPTRNSCN : Current Transfers
 PKGGREVDRCTXN : Direct Revenue Tax Rate
 PAKNECONPRVTKN_A: Add factor:HH. Cons Real
 PAKNECONPRVTKN_D: Fix dummy:HH. Cons Real
 PAKNECONPRVTKN_X: Fix value:HH. Cons Real
 PAKNECONPRVTXN : Implicit LCU defl., Pvt. Cons., 2000 = 1
 PAKNYWBOTLCN : Total Wage Bill
 PAK PANUSATLS : Exchange rate LCU / US\$ - Pakistan

Values :

| | 2020 | 2021 | 2022 | 2023 | 2024 | 2025 |
|------|---------------|---------------|---------------|---------------|---------------|---------------|
| Base | 23,672,888.34 | 23,972,815.36 | 24,164,128.02 | 24,427,863.05 | 24,818,524.47 | 25,323,255.17 |
| Last | 23,672,888.34 | 23,972,815.36 | 24,164,128.02 | 24,427,863.05 | 24,818,524.47 | 25,323,255.17 |
| Diff | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |

Input last run:

| | 2020 | 2021 | 2022 | 2023 | 2024 | 2025 |
|--------------------|---------------|---------------|---------------|---------------|---------------|---------------|
| DURING_2009 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| PAKBMFSTREMTCD | 79.99 | 80.28 | 80.17 | 80.09 | 80.24 | 80.72 |
| PAKBMFSTREMTCD(-1) | 79.31 | 79.99 | 80.28 | 80.17 | 80.09 | 80.24 |
| PAKBXFSTREMTCD | 25,608.16 | 28,184.68 | 30,751.54 | 33,417.42 | 36,250.07 | 39,278.43 |
| PAKBXFSTREMTCD(-1) | 23,080.41 | 25,608.16 | 28,184.68 | 30,751.54 | 33,417.42 | 36,250.07 |
| PAKFMLBLPOLYXN | 6.67 | 7.06 | 7.29 | 7.38 | 7.36 | 7.29 |
| PAKGGEXPTRNSCN | 473,448.51 | 515,359.57 | 553,263.89 | 588,831.93 | 623,434.85 | 658,244.15 |
| PAKGGEXPTRNSCN(-1) | 425,736.84 | 473,448.51 | 515,359.57 | 553,263.89 | 588,831.93 | 623,434.85 |
| PAKGGREVDRCTXN | 4.22 | 4.22 | 4.22 | 4.22 | 4.22 | 4.22 |
| PAKGGREVDRCTXN(-1) | 4.22 | 4.22 | 4.22 | 4.22 | 4.22 | 4.22 |
| PAKNECONPRVTKN(-1) | 23,018,638.24 | 23,672,888.34 | 23,972,815.36 | 24,164,128.02 | 24,427,863.05 | 24,818,524.47 |
| PAKNECONPRVTKN_A | 0.01 | 0.00 | 0.00 | -0.00 | -0.00 | -0.00 |
| PAKNECONPRVTKN_D | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| PAKNECONPRVTKN_X | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| PAKNECONPRVTXN | 1.67 | 1.82 | 1.98 | 2.14 | 2.30 | 2.45 |
| PAKNECONPRVTXN(-1) | 1.52 | 1.67 | 1.82 | 1.98 | 2.14 | 2.30 |
| PAKNYYWBTOTLCN | 30,639,578.34 | 33,667,219.15 | 36,907,411.71 | 40,536,234.92 | 44,615,868.39 | 49,126,944.56 |
| PAKNYYWBTOTLCN(-1) | 27,633,148.53 | 30,639,578.34 | 33,667,219.15 | 36,907,411.71 | 40,536,234.92 | 44,615,868.39 |
| PAKPANUSATLS | 107.18 | 107.01 | 106.84 | 106.69 | 106.57 | 106.45 |
| PAKPANUSATLS(-1) | 106.95 | 107.18 | 107.01 | 106.84 | 106.69 | 106.57 |

The World Bank's MFMod Framework in Python with Modelflow

Input base run:

| | 2020 | 2021 | 2022 | 2023 | 2024 | 2025 |
|--------------------|---------------|---------------|---------------|---------------|---------------|---------------|
| DURING_2009 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| PAKBMFSTREMTCD | 79.99 | 80.28 | 80.17 | 80.09 | 80.24 | 80.72 |
| PAKBMFSTREMTCD(-1) | 79.31 | 79.99 | 80.28 | 80.17 | 80.09 | 80.24 |
| PAKBXFSTREMTCD | 25,608.16 | 28,184.68 | 30,751.54 | 33,417.42 | 36,250.07 | 39,278.43 |
| PAKBXFSTREMTCD(-1) | 23,080.41 | 25,608.16 | 28,184.68 | 30,751.54 | 33,417.42 | 36,250.07 |
| PAKFMLBLPOLYXN | 6.67 | 7.06 | 7.29 | 7.38 | 7.36 | 7.29 |
| PAKGGEPRTRNSCN | 473,448.51 | 515,359.57 | 553,263.89 | 588,831.93 | 623,434.85 | 658,244.15 |
| PAKGGEPRTRNSCN(-1) | 425,736.84 | 473,448.51 | 515,359.57 | 553,263.89 | 588,831.93 | 623,434.85 |
| PAKGGREVDRCTXN | 4.22 | 4.22 | 4.22 | 4.22 | 4.22 | 4.22 |
| PAKGGREVDRCTXN(-1) | 4.22 | 4.22 | 4.22 | 4.22 | 4.22 | 4.22 |
| PAKNECONPRVTKN(-1) | 23,018,638.24 | 23,672,888.34 | 23,972,815.36 | 24,164,128.02 | 24,427,863.05 | 24,818,524.47 |
| PAKNECONPRVTKN_A | 0.01 | 0.00 | 0.00 | -0.00 | -0.00 | -0.00 |
| PAKNECONPRVTKN_D | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| PAKNECONPRVTKN_X | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| PAKNECONPRVTXN | 1.67 | 1.82 | 1.98 | 2.14 | 2.30 | 2.45 |
| PAKNECONPRVTXN(-1) | 1.52 | 1.67 | 1.82 | 1.98 | 2.14 | 2.30 |
| PAKNYYWBTOTLCN | 30,639,578.34 | 33,667,219.15 | 36,907,411.71 | 40,536,234.92 | 44,615,868.39 | 49,126,944.56 |
| PAKNYYWBTOTLCN(-1) | 27,633,148.53 | 30,639,578.34 | 33,667,219.15 | 36,907,411.71 | 40,536,234.92 | 44,615,868.39 |
| PAKPANUSATLS | 107.18 | 107.01 | 106.84 | 106.69 | 106.57 | 106.45 |
| PAKPANUSATLS(-1) | 106.95 | 107.18 | 107.01 | 106.84 | 106.69 | 106.57 |

```
# Prepare the notebook for use of ModelFlow
# Jupyter magic command to improve the display of charts in the Notebook
%matplotlib inline
# Import pandas
import pandas as pd
# Import the model class from the modelclass module
from modelclass import model
# functions that improve rendering of ModelFlow outputs
model.widescreen()
model.scroll_off();
```

```
#Replace the path below with the location of the pak.pcim file (or some other world_
→bank model file) on your computer
mpak,bline = model.modelload('../models/pak.pcim', \
                           alfa=0.7,run=1,keep= 'Baseline')
```

EQUATIONS IN MFMod AND ModelFlow

As noted above an MFMod is comprised of two types of equations: identities and behavioral equations. Identities are mathematical or accounting relationships that are always true. The GDP accounting identity is a well known one:

$$Y_t = C_t + I_t + G_t + X_t - M_t$$

The general government deficit as revenues less spending is another.

Behavioral equations are also endogenous equations, but in a macrostructural model they describe an economic not an accounting relationship. Typically these relationships are estimated econometrically and do not hold exactly.

In this chapter - Equations in MFMod and ModelFlow

This chapter provides a very brief overview of the type of equation in MFMod models (identities and behavioral equations), and a deep dive into the behavioral or econometrically estimated equations in these models.

Three main issues are discussed:

- 1) The importance of **Addfactors** (`_A` variables) in behavioral equations.
- 2) The mechanism by which behavioral equations can be exogenized or de-activated in the ModelFlow environment using the `_D` and `_X` variables.
- 3) An explanation of the Error Correction Model used to estimate many of the econometric relations in World Bank MFMod models.

10.1 A behavioral equation

Normally a behavioral equation is comprised of a left-hand-side variable (the regressand or dependent variable), right-hand side variables (the regressors in the econometric relationship, or explanatory variables), estimated parameters, perhaps some imposed parameters, and an error term.

Assume y_t is the dependent variable, X_t a vector of explanatory variables and η_t the error term, then a simple regression can be written as:

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

where α and β are parameters to be estimated or in most cases β will be a vector of estimated parameters.

Once the estimation has occurred α , β and η_t take on precise values and the equation is rewritten as:

$$y_t = \hat{\alpha} + \hat{\beta} X_t + \hat{\eta}_t$$

where the hats “^” signify the specific value for the parameter that emerged from the estimation process.

We can also write an expression for \hat{y}_t the fitted value from the regression as:

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

Substituting this expression into the previous expression and re-arranging gives us

$$y_t - \hat{y}_t = \hat{\eta}_t$$

All of which are fairly elementary results from econometrics.

10.2 The add factor in behavioral equations

The econometrics used to estimate the equation ensure that the expected value of η_t is zero. So the expectation of the above equation during the forecast period is

$$\begin{aligned} E(y_t - \hat{y}_t) &= E(\hat{\eta}_t) \\ y_t - \hat{y}_t &= 0 \end{aligned}$$

In Macrostructural models the first of these equations is rewritten by substituting AF_t for $\hat{\eta}_t$.

$$y_t = \hat{y}_t + AF_t$$

By imposing a nonzero value on AF_t , the modeller can **add** her judgment to the model's fitted value, either to reflect a view that the forecast value of y will deviate from the fitted value, or because some change in circumstances (say a policy change) will cause the underlying equation to be different in the future than it was when the parameters were estimated (regime change or structural break).

In World Bank models using Modelflow the addfactor of an equation is given the same mnemonic as the dependent variable with an _A appended to it. Thus, in the above simplified version, the equation would be written as

$$y_t = \hat{\alpha} + \hat{\beta}X_t + y_A_t$$

10.3 Excluding behavioral equations

In Modelflow behavioral equations can be excluded “de-activated” or included (“activated”). This is achieved by adding two additional variables to each equation. The first is given the name of the dependent variable with _D appended. The second is given the name of the dependent variable with _X appended.

The preceding equation is then re-written as below

$$y_t = (1 - y_D_t) \cdot \underbrace{\left[\hat{\alpha} + \hat{\beta}X_t + y_A_t \right]}_{\text{Econometric equation}} + y_D_t \cdot \underbrace{y_X_t}_{\substack{\text{Exogenized} \\ \text{value}}}$$

When $y_D_t = 0$, the second part of the equation $y_D_t * y_X_t$ evaluates to zero and drops out, while the expression $(1 - y_D_t)$ evaluates to one. Thus the whole equation simplifies to the standard behavioral equation.

$$y_t = 1 \cdot \left[\hat{\alpha} + \hat{\beta}X_t + y_A_t \right] + 0$$

$$y_t = \hat{\alpha} + \hat{\beta}X_t + y_A_t$$

When $y_D_t = 1$, the $(1 - y_D_t)$ evaluates to zero so the first part of the equation drops out, and the equation simplifies to:

$$y_t = 0 \cdot \left[\hat{\alpha} + \hat{\beta}X_t + y_A_t \right] + 1 \cdot y_X_t$$

$$y_t = y_X_t$$

Thus, when $y_D_t = 1$ the whole equation simply sets the endogenous variable y_t equal to the exogenous variable y_X_t .

10.4 Behavioral equations in ModelFlow

It follows therefore that equations in ModelFlow have three special variables associated with them.

Special variables in ModelFlow behavioral equations

| Terminator | Meaning | Role |
|------------|-------------------|--|
| _A | Add factor: | Special variable to allow judgment to be added to an equation |
| _X | Exogenized value: | Special variable that stores the value that the equation should return if exogenized |
| _D | Exogenous dummy: | Dummy variable. When set to one, the equation will return the value of the _X variable, if zero, it returns the fitted value of the equation plus the Add factor. |

Below the EViews and ModelFlow representations of the Household consumption equation are extracted from the model object using `.frm1()` and `.eviews()` methods discussed in the previous chapter.

In the EViews representation we do not see the special variables but in the `frm1` representation (which is the one actually used by ModelFlow) they are visible.

```
mpak.PAKNECONPRVTKN.eviews
```

```
DLOG(PAKNECONPRVTKN) == 0.2*(LOG(PAKNECONPRVTKN(-1)) - LOG(1.21203101101442) -  
↳ LOG(((PAKBFSTREMTCD(-1) - PAKBMFSTREMTCD(-1))*PAKPANUSATLS(-1)) +  
↳ PAKGGEPRNNSCN(-1) + PAKNYWBTOTLCN(-1)*(1 - PAKGGREVDRCTXN(-1)/100))/  
↳ PAKNECONPRVTXN(-1))) + 0.763938860758873*DLOG(((PAKBFSTREMTCD -  
↳ PAKBMFSTREMTCD)*PAKPANUSATLS) + PAKGGEPRNNSCN + PAKNYWBTOTLCN*(1 -  
↳ PAKGGREVDRCTXN/100))/PAKNECONPRVTXN) - 0.0634474791568939*@DURING("2009") - 0.  
↳ 3*(PAKFMLBLPOLYXN/100 - DLOG(PAKNECONPRVTXN))
```

```
mpak.PAKNECONPRVTKN.frm1
```

```

Endogeneous: PAKNECONPRVTKN: HH. Cons Real
Formular: FRML <DAMP,STOC> PAKNECONPRVTKN = (PAKNECONPRVTKN(-1)*EXP(PAKNECONPRVTKN_-
˓→_A+ (-0.2*(LOG(PAKNECONPRVTKN(-1))-LOG(1.21203101101442)-LOG(((PAKBXFSTREMTCD(-
˓→_1)-PAKBMFSTREMTCD(-1))*PAK PANUSATLS(-1))+PAKGGE X PTRNSCN(-1)+PAKNYYWBTOTLCN(-
˓→_1)*(1-PAKGGREVDRCTXN(-1)/100))/PAKNECONPRVTXN(-1)))+0.
˓→_763938860758873*((LOG(((PAKBXFSTREMTCD-
˓→_PAKBMFSTREMTCD)*PAK PANUSATLS)+PAKGGE X PTRNSCN+PAKNYYWBTOTLCN*(1-PAKGGREVDRCTXN/
˓→_100))/PAKNECONPRVTXN))-(LOG(((PAKBXFSTREMTCD(-1)-PAKBMFSTREMTCD(-
˓→_1))*PAK PANUSATLS(-1))+PAKGGE X PTRNSCN(-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
PAKGGE X PTRNSCN : Current Transfers
PAKGGREVDRCTXN : Direct Revenue Tax Rate
PAKNECONPRVTKN_A: Add factor:HH. Cons Real
PAKNECONPRVTKN_D: Fix dummy:HH. Cons Real
PAKNECONPRVTKN_X: Fix value:HH. Cons Real
PAKNECONPRVTXN : Implicit LCU defl., Pvt. Cons., 2000 = 1
PAKNYYWBTOTLCN : Total Wage Bill
PAK PANUSATLS : Exchange rate LCU / US$ - Pakistan

```

Careful inspection of the output from the `.frml()` and `eviews()` methods, reveals that in the `.frml()` specification the three special variables have been added to the model formula that are not part of the EViews output. These variables each have the same root mnemonic as the dependent variable **PAKNECONRPVTKN** but have special terminators `_A` `_X` `_D` appended to them.

To exclude an equation, the `_D` variable is set to 1 and the equation simplifies to `PAKNECONRPVTKN=PAKNECONRPVTKN_X` if `_D=0` then the econometric relationship and the add-factor will jointly determine the value of **PAKNECONRPVTKN**.

10.5 The ECM specification

Many of the behavioral equations in World Bank models are written as Error Correction Models (ECMs).

The Error correction specification was developed to deal with two important problems in econometric equations.

1. Many time-series data tend to increase over time. As a result, a regression of one series on another series tends to have good fit even if the two variables are not really connected economically. For example, the price of cookies tends to rise over time because of inflation. Similarly, the quantity of screws produced in the manufacturing sector tends to rise over time because of increased population and, therefore, demand for manufactured goods. Regressing screw production on cookie prices will show a strong but spurious correlation.
2. Purely short run models focus on growth or differences and get around the problem of the spurious correlation arising from regressing two unrelated series that each have a trend. While, these explained the short run deviations, stringing the estimated growth rates together could result in implicit levels that were unstable because they were not anchored to the long-run relationship between variables dictated by underlying economic theory (or empirical behavior).

The co-integration approach to econometrics (`{cite:t} engle_co-integration_1987`) combined with the closely related ECM approach provided a solution to the above problem by providing a mechanism for modeling both the long run relationship and short-run relationships between variables.

The ECM specification used in World Bank models is a single equation approach that follows (Wickens and Breusch (1988)) and is comprised of two parts (the long run relationship, and the short-run relationship), which are estimated simultaneously.

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 and x .

10.5.1 The short run relationship

In its simplest form, a short run relationship between the growth rates of two variables could be written as:

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

or substituting lower case letters for the logged values.

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

10.5.2 The long run equation

The long run relates the level of two (or more) variables. A simplified version of that equation can be written as:

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

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

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

The long run equation in the steady state

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 the long run relationship can be simplified to:

$$y_t = \ln(\alpha) + \beta x_t + 0$$

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

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

Moreover if this expression is multiplied 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

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

The part of the equation between the parenthesis is equal to the lagged error term of the long-run equation (η_{t-1}). In the Long Run its expected value is zero, but at any give instant it could be different from zero. The distance it is from zero at any point in time, reflects the distance that the dependent variable is from its long-run equilibrium value at that moment.

10.5.3 Putting it together

From before we have the short run equation:

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

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

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

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

10.5.4 Lambda, the speed of adjustment

The parameter λ can then be interpreted as the speed of adjustment. It determines what share of the previous period error (distance from equilibrium) is absorbed in the following period. As long as λ is greater than zero and less or equal to one if there are no further disturbances ($\epsilon_t = 0$) the expression multiplied by lambda will slowly decline toward zero. How fast depends on how large or small is λ .

Intuitively, the lagged long-run error-term measures how far the model was from equilibrium one period earlier (at t-1). The ECM term (multiplied by λ) ensures the model will slowly converge to equilibrium – the point at which the long run equation holds exactly – if λ is greater than zero but less than or equal to one. In these conditions during each time period some portion λ of the previous period year's disequilibrium will be absorbed each year. How much is absorbed depends on the size of estimated speed of the adjustment coefficient λ .

An ECM equation can, therefore be broken into two component parts. For the consumption function it will look something like this:

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

More precisely to be convergent λ must be between 0 and 2. If Lambda is greater than 1 but less than two, the error term will oscillate from positive to negative but will slowly converge to zero.

If lambda is greater than 2 (or less than zero), then the long run portion of the equation will cause the disequilibrium to grow each period not diminish.

If lambda is less is greater than zero but less than one, the equation will converge more or less directly at a speed determined by the value of λ .

An illustrative example

Below three ECMs are written out, each with an equilibrium value of 50 and different speeds of adjustment ranging from 0.3, 0.5 and 0.9. The figure and table below illustrate the adjustment to the equilibrium value starting from an initial value of 100 (error of 50) under the three speeds of adjustment.

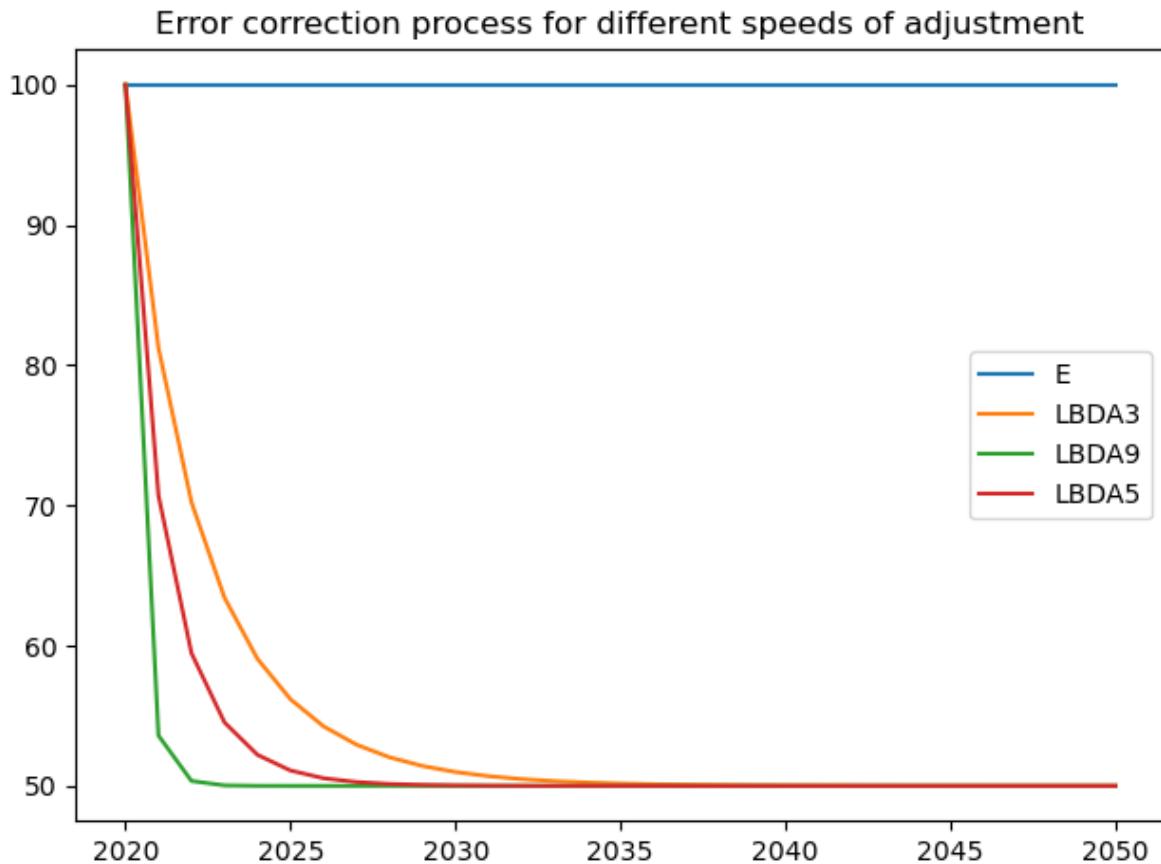
```
import pandas as pd
ECMdf = pd.DataFrame({'E': 100}, index=[v for v in range(2020,2051)])
ECMdf=ECMdf.upd('lbda3 lbda9 lbda5 = 100')
ECMdf=mfcalc('''
<2021 2050> dlog(Lbda3) = -.3 * (log(Lbda3(-1))-log(50))
<2021 2050> dlog(Lbda9) = -.9 * (log(Lbda9(-1))-log(50))
```

(continues on next page)

(continued from previous page)

```
<2021 2050> dlog(Lbda5) = -.5 * (log(Lbda5(-1))-log(50))
' ''')

ECMdf.plot(title="Error correction process for different speeds of adjustment");
```



With a slow speed of adjustment, the equilibrium level of 50 is not achieved until around 2030 (<51>) or 2032 (50.5). With $\lambda=0.5$ the gap is closed in around 5 years (2025=50.5), while with $\lambda=0.9$ it takes just two years (2023=50.3).

Note

Advanced formatting of tables The table below introduces some advanced formatting routines, using the Pandas style property. For more see [here](https://pandas.pydata.org/pandas-docs/stable/user_guide/style.html) (https://pandas.pydata.org/pandas-docs/stable/user_guide/style.html).

Info on python named colors can be found here: https://matplotlib.org/stable/gallery/color/named_colors.html.

```
def color_proximity(val):
    if val > 80:
        color="red"
    elif val > 70:
        color="orangered"
    elif val > 55:
        color="coral"
    elif val > 51:
```

(continues on next page)

(continued from previous page)

```

    color="lightsalmon"
elif val > 50.5:
    color="peachpuff"
else:
    color="white"
return 'background-color: %s' % color
ECMdf.loc[2020:2035,['LBDA3','LBDA5','LBDA9']].style.map(color_proximity) \
.format(precision=2).set_table_attributes('style="font-size: 10px"')

```

| | LBDA3 | LBDA5 | LBDA9 |
|------|--------|--------|--------|
| 2020 | 100.00 | 100.00 | 100.00 |
| 2021 | 81.23 | 70.71 | 53.59 |
| 2022 | 70.22 | 59.48 | 50.35 |
| 2023 | 63.42 | 54.53 | 50.03 |
| 2024 | 59.05 | 52.21 | 50.00 |
| 2025 | 56.18 | 51.09 | 50.00 |
| 2026 | 54.25 | 50.54 | 50.00 |
| 2027 | 52.94 | 50.27 | 50.00 |
| 2028 | 52.04 | 50.14 | 50.00 |
| 2029 | 51.42 | 50.07 | 50.00 |
| 2030 | 50.99 | 50.03 | 50.00 |
| 2031 | 50.69 | 50.02 | 50.00 |
| 2032 | 50.48 | 50.01 | 50.00 |
| 2033 | 50.34 | 50.00 | 50.00 |
| 2034 | 50.24 | 50.00 | 50.00 |
| 2035 | 50.16 | 50.00 | 50.00 |

SCENARIO ANALYSIS

Building and running scenarios is central to working with a macroeconomic model. Scenarios help to quantify the expected impacts of different policies and external events. In so doing they give valuable insights to policy makers concerning potential policies. This chapter introduces users to scenario analysis using World Bank models and ModelFlow. The scenarios presented are relatively simple, but cover all of the different kind of scenarios typically performed and insights into how to get around shortcomings that a specific model may have for a given question.

1 In this chapter - Scenario Analysis

This chapter presents examples of running simulations. Four types of simulation are covered:

- 1) **Permanent Exogenous Shocks:** These involve either *shocking an exogenous variable* (say World oil prices) or *deactivating an equation* and treating its dependent variable as if it were exogenous and shocking it directly.
- 2) **Endogenous Shocks:** In these shocks, a behavioral equation is left active, but its *add-factor is shocked* in order to impact its trajectory. This might be used when an external shock is expected, but the analyst wants the subsequent (and even contemporaneous) behavior of the variable to react to second-round effects that occur in the model (i.e., an increase in investment by one firm could be modelled using an add-factor, but as GDP rises, other firms would normally also want to increase their investment to meet the additional demand. By using the add-factor this second order effect is enabled).
- 3) **Temporary Exogenous Shocks:** Here a behavioral equation can be temporarily deactivated — give the endogenous variable an initial shock by setting its level directly to a specific level for a given period — but then reactivated in subsequent periods so that the variable's trajectory is affected by second- and third-round effects of the initial shock. This differs from the add-factor shock in that there is no endogenous reaction of the shocked variable during the period it is exogenized.
- 4) **Mixed Scenarios:** More complex scenarios that combine one or more shocks.

Similar shocks using each method are presented, allowing the reader to visualize the consequences of choosing one method over another.

11.1 Prepare the python session

A first step in any policy analysis is to prepare your python environment for a new ModelFlow session. Starting with a fresh session (a clean slate as it were) is a critical assure reproducibility by ensuring that each time a scenario or scenarios are run they do so from the same starting point. This is done below starting a new python session, initializing pandas and ModelFlow and the reading a previously saved WBG model read from disk and solving it.

```
# Prepare the notebook for use of ModelFlow
# Jupyter magic command to improve the display of charts in the Notebook
```

(continues on next page)

(continued from previous page)

```
%matplotlib inline
# Import pandas
import pandas as pd
# Import the model class from the modelclass module
from modelclass import model
# functions that improve rendering of ModelFlow outputs
model.widescreen()
model.scroll_off();
```

Note

These are precisely the same commands used to start the previous chapter and form the essential initialization commands of any python session using ModelFlow.

Set some pandas display options.

```
# pandas options to make output more legible:
pd.set_option('display.width', 100)
pd.set_option('display.float_format', '{:.2f}'.format)
```

Following the discussion in the previous chapter a model object mpak is loaded and solved with the result being stored in a new dataframe called bline. The keep option causes a copy of the solved scenario to be stored as “Baseline” within the model object mpak.

```
#Load a saved version of the Pakistan model and solve it,
#saving the results in the model object mpak, and the resulting dataframe in bline
#Replace the path below with the location of the pak.pcim file on your computer
mpak,bline = model.modelload('../models/pak.pcim', \
                           run=True,keep= 'Baseline')
```

Zipped file read: ..\models\pak.pcim

Recall

In addition, to the keep dataframe and the bline dataframe the model object (mpak in this instance) always contains two DataFrames. The .basedf dataframe that contains the initial values for the variables of the model, and the .lastdf contains the results of the most recently executed scenario – in this case lastdf will have the same values as bline.

11.1.1 Checking that the simulated results reproduce the inputs

A critical component of reproducibility is that when a model is solved without changing any inputs (as was the case of the load) the model should return (reproduce) exactly the same data as before¹. The results from the simulation run on loading the model (caused by the Run=True option in the modelload command) can be examined by comparing the values in the basedf and lastdf DataFrames.

Below, the percent difference between the values of the variables for real GDP and Consumer demand in the two dataframes .basedf and lastdf are displayed. They return zero following a simulation where the inputs were

¹ If a model does not reproduce itself when solved, then mathematically it means that the system of equations is not closed (there is no single answer) or data are inconsistent (usually one or more identities do not hold). In such circumstances, either the model will not solve or it will but it will not be possible to have confidence in any of the results generated.

not changed – confirming that the model reproduced the original data.

The .difpctlevel method of the model object display for the variables indicated the contents of the .lastdf (results of most recent simulation) and .basedf (initial simulation results) DataFrames, expressed as the percent change from the original (.basedf). $\frac{\text{lastdf} - \text{basedf}}{\text{basedf}} * 100$ The .df modifier instructs the model object to return the results as a DataFrame (Chapter 14 below has more on report-writing options and techniques).

```
with mpak.set_smpl(2020, 2030):
    print(mpak['PAKNYGDPMKTPKN PAKNECONPRVTKN'].difpctlevel.rename().df)
```

| | Real GDP | HH. Cons | Real |
|------|----------|----------|------|
| 2020 | 0.00 | 0.00 | |
| 2021 | 0.00 | 0.00 | |
| 2022 | 0.00 | 0.00 | |
| 2023 | 0.00 | 0.00 | |
| 2024 | 0.00 | 0.00 | |
| 2025 | 0.00 | 0.00 | |
| 2026 | 0.00 | 0.00 | |
| 2027 | 0.00 | 0.00 | |
| 2028 | 0.00 | 0.00 | |
| 2029 | 0.00 | 0.00 | |
| 2030 | 0.00 | 0.00 | |

11.2 Different kinds of simulations

The ModelFlow package performs 4 different kinds of simulation:

1. A shock to an exogenous variable in the model.
2. An exogenous shock of a behavioral variable, executed by exogenizing the variable (de-activating its equation).
3. An endogenous shock of a behavioral variable, executed by shocking the add factor of the variable.
4. A mixed shock of a behavioral variable, achieved by temporarily exogenizing the variable.

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

As a result, this possibility is not discussed.

11.2.1 A shock to an exogenous variable

A World Bank model will reproduce the same values if inputs (exogenous variables) are not changed (as demonstrated above). In the simulation below, the oil price is changed – increasing by \$25 for the three years between 2025 and 2027 inclusive. As a result, we expect the solution to return different values for the endogenous variables that are sensitive to changes in the price of oil, such as GDP, inflation, consumption and the current account balance for example.

Preparing the data for simulation

The following steps are performed to prepare the simulation:

1. Using the `.upd` method, a new input dataframe `oilshockdf` is created where the oil price is increased by \$25 during 2025, 2026 and 2027.
2. A dataframe `compdf` is assigned the pre-shock and post-shock values of the oil price and the difference between the initial and shocked values.
3. Finally the results are displayed, confirming that the `mfcalc` statement revised the oil price data.

More on the `.upd` method here: *The upd method returns a DataFrame with updated variables*

Even more *upd test*

```
# Use the upd routine to create a new dataframe where $25 is added to the oil price_
# bewteen 2025 and 2027
oilshockdf = mpak.basedf.upd('<2025 2027> WLDFCRUDE_PETRO + 25')
#compdf.drop(axis=0, inplace=True) #snure compdf is empty
# Create new df compdf and initialize it for the period
# 2000-2030 with the values from basedf
compdf=mpak.basedf.loc[2000:2030,['WLDFCRUDE_PETRO']]
compdf = compdf.rename(columns={'WLDFCRUDE_PETRO': 'Original'})
# Add a new series LASTDF with the World Crude price from the shock dataframe
compdf['Shock']=oilshockdf.loc[2000:2030,['WLDFCRUDE_PETRO']]
# Add a final series as the difference between the first two.
compdf['Dif']=compdf['Shock']-compdf['Original']
# Display the new comparison dataframe
compdf.loc[2024:2030]
```

| | Original | Shock | Dif |
|------|----------|--------|-------|
| 2024 | 80.37 | 80.37 | 0.00 |
| 2025 | 85.34 | 110.34 | 25.00 |
| 2026 | 90.61 | 115.61 | 25.00 |
| 2027 | 96.22 | 121.22 | 25.00 |
| 2028 | 102.17 | 102.17 | 0.00 |
| 2029 | 108.48 | 108.48 | 0.00 |
| 2030 | 115.19 | 115.19 | 0.00 |

Running the simulation

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

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

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

Results

ModelFlow tools can be used to visualize the impacts of the shock; as a table; as a chart and, within Jupyter notebook, as an interactive widget.

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

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

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

| WLDFCRUDE_PETRO | |
|-----------------|-------|
| 2020 | 0.00 |
| 2021 | 0.00 |
| 2022 | 0.00 |
| 2023 | 0.00 |
| 2024 | 0.00 |
| 2025 | 25.00 |
| 2026 | 25.00 |
| 2027 | 25.00 |
| 2028 | 0.00 |
| 2029 | 0.00 |
| 2030 | 0.00 |

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

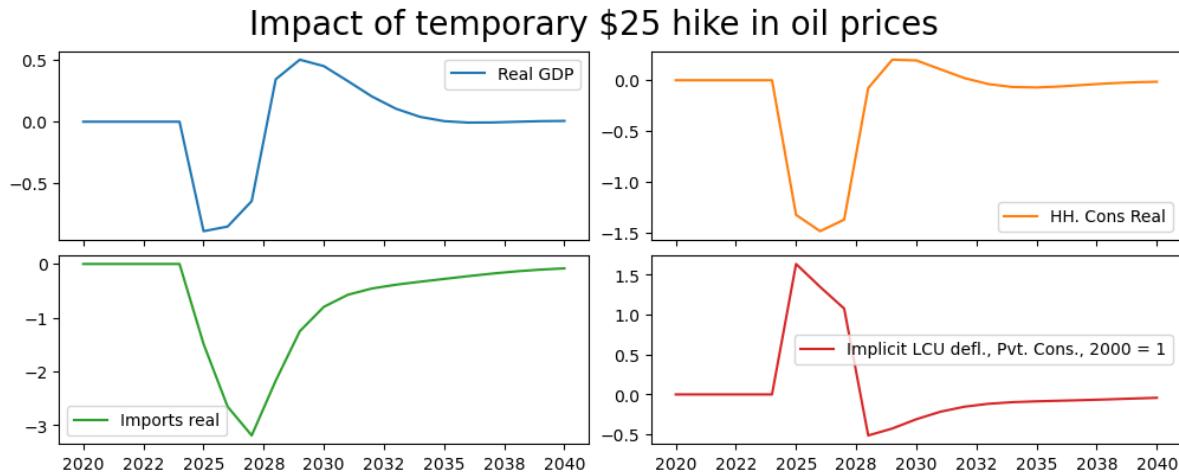
The first variable `PAKNYGDPMKTPKN` is Pakistan's real GDP, the second `PAKNECONPRVTKN` is real consumption, the third is **REAL IMPORTS OF GOODS AND SERVICES** (`PAKNEIMPGNFSKN`) and the final is the level of the Consumer price deflator `PAKNECONPRVTXN`.

The modifier `.difpctlevel` instructs ModelFlow to calculate the difference in the selected variables and express them as a percent of the original level.

$$= \frac{x_{new} - x_{original}}{x_{original}} * 100$$

The modifier `rename()` will replace the variable **name** with the variable **description** in the graphs.

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



```
with mpak.set_smpl(2020,2035):
    print(mpak['PAKNYGDPMKTPKN PAKNECONPRVTKN PAKNEIMPGNFSKN PAKNECONPRVTXN'].
        ↪difpctlevel.dfd
```

| | PAKNYGDPMKTPKN | PAKNECONPRVTKN | PAKNEIMPGNFSKN | PAKNECONPRVTXN |
|------|----------------|----------------|----------------|----------------|
| 2020 | 0.00 | 0.00 | 0.00 | 0.00 |
| 2021 | 0.00 | 0.00 | 0.00 | 0.00 |
| 2022 | 0.00 | 0.00 | 0.00 | 0.00 |
| 2023 | 0.00 | 0.00 | 0.00 | 0.00 |
| 2024 | 0.00 | 0.00 | 0.00 | 0.00 |
| 2025 | -0.89 | -1.32 | -1.49 | 1.64 |
| 2026 | -0.85 | -1.48 | -2.65 | 1.35 |
| 2027 | -0.64 | -1.37 | -3.19 | 1.08 |
| 2028 | 0.34 | -0.08 | -2.17 | -0.51 |
| 2029 | 0.50 | 0.20 | -1.25 | -0.43 |
| 2030 | 0.45 | 0.19 | -0.80 | -0.31 |
| 2031 | 0.33 | 0.11 | -0.57 | -0.22 |
| 2032 | 0.20 | 0.02 | -0.46 | -0.15 |
| 2033 | 0.10 | -0.04 | -0.39 | -0.12 |
| 2034 | 0.04 | -0.07 | -0.33 | -0.10 |
| 2035 | 0.00 | -0.07 | -0.28 | -0.09 |

The graphs show the change in the level as a percent of the previous level. They suggest that a temporary \$25 oil price hike would reduce GDP in the first year by about 0.9 percent, but that the impact would diminish by the third year to -.64 percent, and then turn positive in the fourth year when the temporary price hike was no longer in effect. By the end of the simulation period the net effect on GDP would be zero.

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

The GDP impact is smaller partly because the decline in domestic demand (due to reduced real incomes from high oil prices) reduces imports. Because imports enter into the GDP identity with a negative sign, a reduction in imports actually increase aggregate GDP – or in this case partially offsets the declines coming from reduced consumption (and investment - which is not shown above).

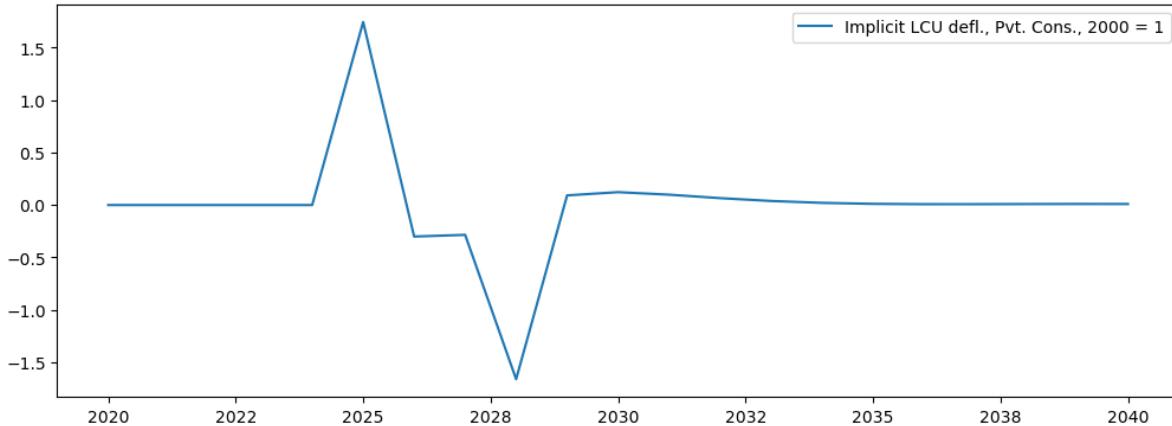
Finally as could be expected, initially prices rise sharply with higher oil prices. However, as the slowdown in growth is felt, inflationary pressures turn negative and the overall impact on the price level declines. When the oil price hike is eliminated, the overall impact on prices turns negative, and is still slowly returning to zero by the end of the simulation period.

Note: The graph and table above shows what is happening to the **price level**. To see the impact on inflation (the rate of growth of prices), a separate graph can be generated using `difpct`, which shows the change in the rate of growth of

variables where the growth rate is expressed as a per cent $\left[\left(\frac{x_{t-1}^{shock}}{x_{t-1}^{baseline}} - 1 \right) - \left(\frac{x_t^{baseline}}{x_{t-1}^{baseline}} - 1 \right) \right] * 100.$

```
mpak['PAKNECONPRVTXN'].difpct.rename().plot()
    title="Change in inflation from a temporary $25 hike in oil prices",
    colrow=1,ysize=4);
```

Change in inflation from a temporary \$25 hike in oil prices



This view, gives a more nuanced result. The inflation rate increases initially by about 1.6 percentage points, but falls compared with the baseline during the 2026-2027 period as the influence of the slowdown in GDP more than offsets the continued inflationary impetus from the lagged increase in oil prices. In 2028, when oil prices drop back to their previous level, an additional dis-inflationary force is generated and a sharp drop in inflation as compared with the baseline ensues. Over time, the boost to demand from lower prices translates into an acceleration in growth and a return of inflation back to its trend rate.

11.2.2 An exogenous shock to a Behavioral variable

Behavioral equations can be de-activated by exogenizing them, either for the entire simulation period, or for a selected sub-period. When exogenized, instead of the equation returning the value returned by its econometrically equation plus the add-factor, it returns the value placed in the variable $_X_t$ (see the discussion in Chapter Ten).

In the following scenario, consumption is exogenized for the entire simulation period.

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

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

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

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

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

$$cons_t = (1 - cons_D) * \left[C'(X_t) \right] + cons_D * cons_X_t$$

When $cons_D = 1$ (as it does in this scenario) the first part of the equation $(1 - cons_D) * C'(X)$ evaluates to zero and consumption is equal to $(1)^* cons_X$. If instead (which would be the normal case) $cons_D$ were set to zero, the equation would simplify to $cons_t = C'(X_t)$ where $C'(X_t)$ is the estimated equation that determines the value of real consumption in the model conditioned on the value of other variables in the model (the X_t of $C'(X_t)$) and the add factor $C_A - t$.

The `.fix()` method also 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 simulation period, the normal behavioral equation is deactivated or exogenized and consumption will just equal its exogenized value: $PAKNECONPRVTKN = PAKNECONPRVTKN_X$.

```
# reset the active sample period to the full model.
mpak.smp1()
#Create a new df equal to the initial one (bline initialized when we loaded the
#model)
# but set real consumption exogenous
Cfixed=mpak.fix(bline,'PAKNECONPRVTKN')
```

The following variables are fixed
`PAKNECONPRVTKN`

Box 4. The steps performed by the `.fix()` method

The fix command above effectively does in one line each of the following lines of code.

```
bline_real=bline.copy() #make a copy of the baseline dataframe
#Create the _X variables if we exogenize the equation
baseline_real = baseline_real.mfcalc('''
PAKNECONPRVTKN_X = PAKNEPRVTKN
''')
#create the _D variable so we can exogenize the equation (set _D=1)-- currently it
#is exogenized
baseline_real = baseline_real.upd('''
<-0 -1>
PAKNECONPRVTKN_D = 1
'''')
```

Preparing the shock

For the moment, the equation is exogenized but the values have been set to the same values as the `.basedf DataFrame` (`bline` and `basedf` have the same values on load), so solving the model will not change anything.

The `.upd()` method can now be used to implement the assumption that real consumption (`PAKNECONPRVTKN`) would be 2.5% stronger. Because the equation has been turned off, it is the `_X` version of the variable that is increased by 2.5 percent.

```
# Bump the _X version of the variable by 2.5% between 2025 and 2040
Cfixed=Cfixed.upd("<2025 2040> PAKNECONPRVTKN_X * 1.025")
```

Performing the simulation

To perform the simulation, the revised CFixed DataFrame is input to the mpak model solve routine, and the model is solved over the period 2020 through 2040.

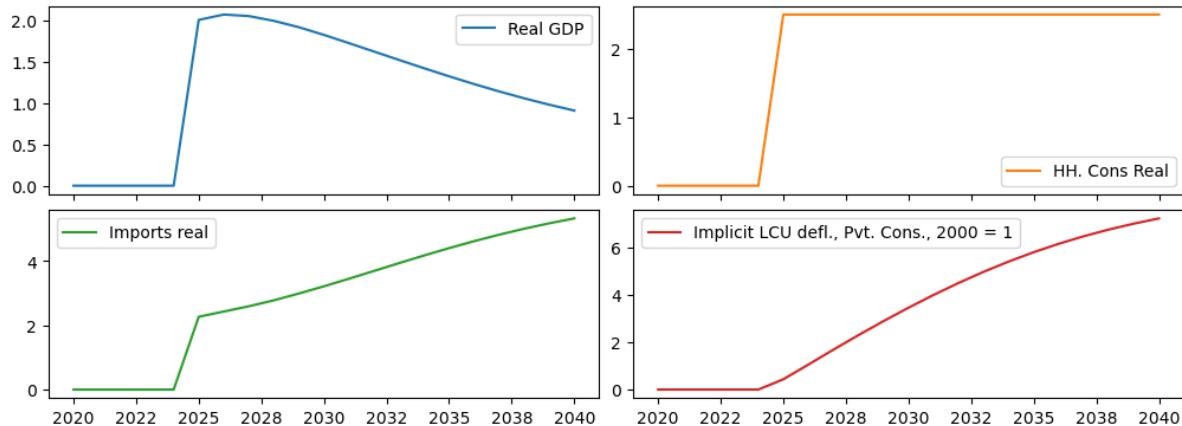
```
CFixedRes = mpak(Cfixed,2020,2040,keep='2.5% increase in C 2025-40 (fix)')

# simulates the model for the period 2020 2040 and gives the name '2.5% increase in C
# 2025-40 to the simulation
CFixedRes = mpak(Cfixed,2020,2040,keep='2.5% increase in C 2025-40')
```

The results can be examined graphically as before.

```
#plots the percent difference (*100) between the lastdf and basedf versions of the
#specified variables
(mpak['PAKNYGDPMKTPKN PAKNECONPRVTKN PAKNEIMPGNFSKN PAKNECONPRVTXN'].
difpctlevel.rename().plot(title="Impact of a permanent 2.5% increase in Consumption
#"));
```

Impact of a permanent 2.5% increase in Consumption



Below results are displayed in tabular form. Note the use of the pandas options with the with clause. This sets the display format of floating point variables to one decimal points. The second with clause temporarily restricts the display to the period 2020 to 2040.

```
with pd.option_context('display.float_format', '{:.1f}%'.format):
    with mpak.set_smp1(2020,2040):
        print(mpak['PAKNYGDPMKTPKN PAKNECONPRVTKN PAKNEIMPGNFSKN'].
difpctlevel.rename().df)
```

| | Real GDP | HH. Cons Real | Imports real |
|------|----------|---------------|--------------|
| 2020 | 0.0% | 0.0% | 0.0% |
| 2021 | 0.0% | 0.0% | 0.0% |
| 2022 | 0.0% | 0.0% | 0.0% |
| 2023 | 0.0% | 0.0% | 0.0% |
| 2024 | 0.0% | 0.0% | 0.0% |
| 2025 | 2.0% | 2.5% | 2.3% |
| 2026 | 2.1% | 2.5% | 2.4% |
| 2027 | 2.1% | 2.5% | 2.6% |
| 2028 | 2.0% | 2.5% | 2.8% |
| 2029 | 1.9% | 2.5% | 3.0% |
| 2030 | 1.8% | 2.5% | 3.2% |

(continues on next page)

(continued from previous page)

| | | | |
|------|------|------|------|
| 2031 | 1.7% | 2.5% | 3.5% |
| 2032 | 1.6% | 2.5% | 3.7% |
| 2033 | 1.5% | 2.5% | 3.9% |
| 2034 | 1.4% | 2.5% | 4.2% |
| 2035 | 1.3% | 2.5% | 4.4% |
| 2036 | 1.2% | 2.5% | 4.6% |
| 2037 | 1.1% | 2.5% | 4.8% |
| 2038 | 1.1% | 2.5% | 5.0% |
| 2039 | 1.0% | 2.5% | 5.2% |
| 2040 | 0.9% | 2.5% | 5.3% |

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

11.2.3 Exogenize a behavioral variable and temporarily shock it

The third method of formulating a scenario involves exogenizing an endogenous variable and shocking its value for a defined period of time. The methodology is the same except the period for which the variable is exogenized is different.

Here the set up is basically the same as before.

```
#reset the active sample period to the full period
mpak.smpl(2020,2040)
# create a copy of the bline DataFrame, but setting the PAKNECONPRVTKN_D variable to
# 1 for the period 2025 through 2027
CTempExogAll=mpak.fix(bline,'PAKNECONPRVTKN')
```

The following variables are fixed
PAKNECONPRVTKN

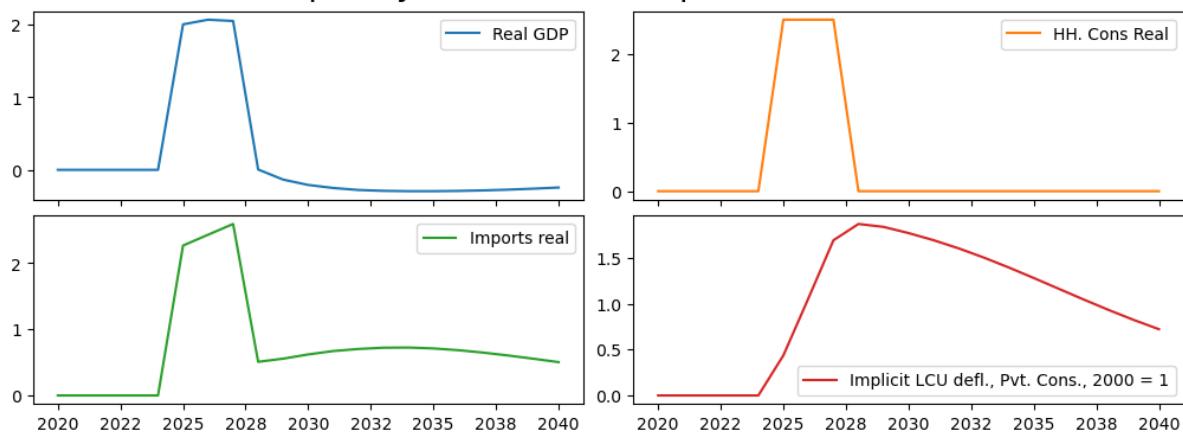
The above .fix() command exogenizes the variable PAKNECONPRVTKN real consumer consumption. The .upd() method in the following line increases the exogenized value of the consumption variable PAKNECONPRVTKN_X by 1.25 percent for three years only, 2025, 2026 and 2027.

```
# multiply the exogenized value of consumption by 2.5% for 2025 through 2027
CTempExogAll=CTempExogAll.upd("<2025 2027> PAKNECONPRVTKN_X * 1.025")
```

Finally the model is solved and selected results displayed as shock-control in percent from the baseline (pre-shock values of the displayed variables).

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

Temporary hike in Consumption 2025-2027



The results are quite different in this scenario. GDP is boosted initially as before but when consumption drops back to its pre-shock level in 2028, GDP and imports decline sharply.

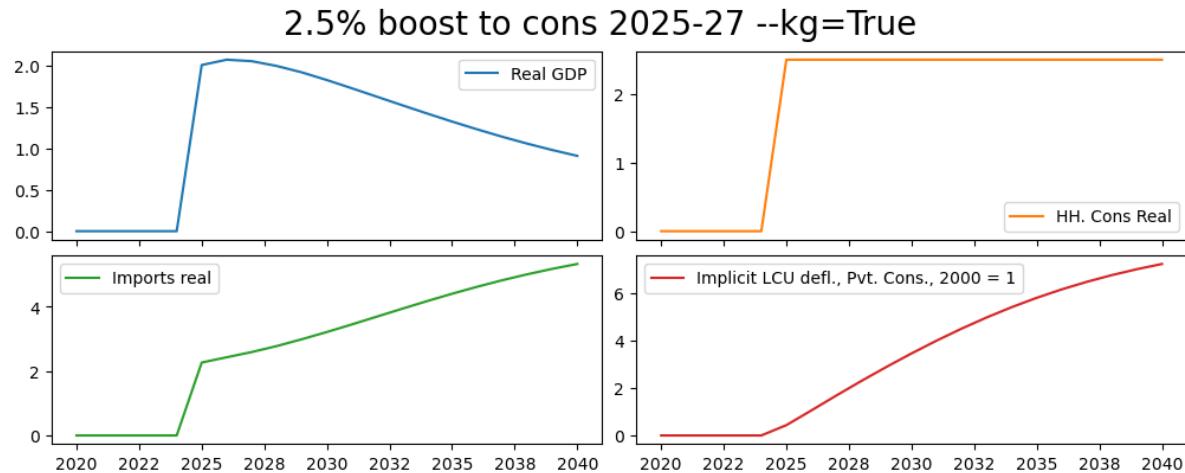
Prices (and inflation) are higher initially but when the economy starts to slow after 2025 prices actually fall (deflation). Although prices are falling, the level of prices remains higher at the end of the simulation than it was in the baseline.

Temporary shock exogenized for the whole period (with KG Option)

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

```
mpak.smpl() # reset the active sample period to the full model.
CTempExogAllKG=mpak.fix(bline,'PAKNECONPRVTKN')
CTempExogAllKG = CTempExogAllKG.upd('''
<2025 2027> PAKNECONPRVTKN_X * 1.025 --kg
'',lprint=0)
#Now we solve the model
CTempXAllResKG = mpak(CTempExogAllKG,2020,2040,keep='2.5% increase in C 2025-27 --_
↪exog whole period --KG=True') # simulates the model
(mpak['PAKNYGDPMKTPKN PAKNECONPRVTKN PAKNEIMPGNFSKN PAKNECONPRVTXN'].difpctlevel.
 rename().plot(title="2.5% boost to cons 2025-27 --kg=True"));
```

The following variables are fixed
PAKNECONPRVTKN



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

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

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

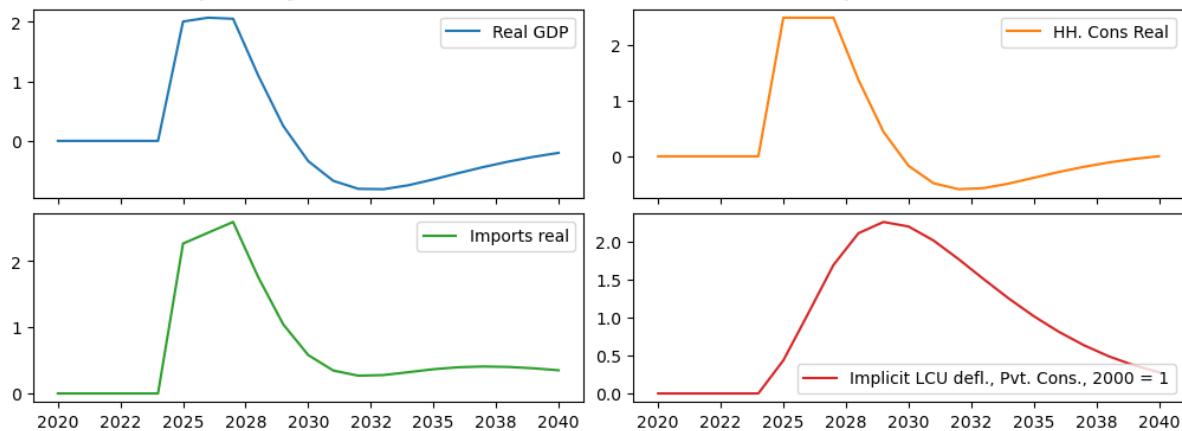
Looking at the maths of the model the consumption equation is effectively split into three.

1. for the period before 2025 $cons_D = 0$ and the consumption equation simplifies to: $cons = C(X)$
2. for the period 2025-2028 it is exogenized ($cons_D = 1$) so it simplifies to: $cons = cons_X$
3. but in the final period 2028-2040 ($cons_D = 0$) and the equation reverts to: $cons = C(X)$

```
# reset the active sample period to the full model.
mpak.smp1()
#Consumption is exogenized only for three years 2025 2026 and 2027
#      PAKNECONPRVTKN_D=1 for 2025,2026, 2027 0 elsewhere.
# NB the 2025,2027 sets the period over which the change is made not specific dates
# -- i.e. 2025, 2026 and 2027 anot just 2025 and 2027
#In subsequent years (2028 onwards) the level of consumption will be determined by_
#the equation
CExogTemp=mpak.fix(bline,'PAKNECONPRVTKN',2025,2027)
#Now increase Consumption by 2.5% over the period 2025-2027
CExogTemp = CExogTemp.upd('<2025 2027> PAKNECONPRVTKN_X * 1.025',lprint=0)
#Solve the model by passing it the revised DataFrame
CExogTempRes = mpak(CExogTemp,2020,2040,keep='2.5% increase in C 2025-27 --_
#temporarily exogenized') # simulates the model
#display the impulse response functions of the specified variables
(mpak['PAKNYGDPMKTPKN PAKNECONPRVTKN PAKNEIMPGNFSKN PAKNECONPRVTXN'].difpctlevel.
rename()).plot(title="Temporary 2.5% boost to cons 2025-27 - equation active"));
```

The following variables are fixed
PAKNECONPRVTKN

Temporary 2.5% boost to cons 2025-27 - equation active



These results have important differences compared with the previous. The most obvious is visible in looking at the graph for Consumption. Rather than reverting immediately to its earlier pre-shock level, it falls more gradually and actually overshoots (falls below its earlier level), before returning slowly to its pre-shock level. That is because unlike in the previous shocks, its path after 2027 is being determined endogenously and reacting to changes elsewhere in the model, notably changes in prices, wages and government spending as well as the lagged level of consumption.

```
print('Consumption base and shock levels\r\n');
print('\r\nReal values in 2030');
print(f'Base value: {bline.loc[2028,"PAKNECONPRVTKN"]:.0f}'.ljust(15,' ') + f'Shocked value: {CExogTempRes.loc[2028,"PAKNECONPRVTKN"]:.0f}'.ljust(15,' '))
print(f'Percent difference: {round(100*((CExogTempRes.loc[2030,"PAKNECONPRVTKN"]-bline.loc[2028,"PAKNECONPRVTKN"])/bline.loc[2028,"PAKNECONPRVTKN"])):.2f}' )
print('\r\n\r\nReal values in 2040');
print(f'Base value: {bline.loc[2040,"PAKNECONPRVTKN"]:.0f}'.ljust(15,' ') + f'Shocked value: {CExogTempRes.loc[2040,"PAKNECONPRVTKN"]:.0f}'.ljust(15,' '))
print(f'Percent difference: {round(100*((CExogTempRes.loc[2040,"PAKNECONPRVTKN"]-bline.loc[2040,"PAKNECONPRVTKN"])/bline.loc[2040,"PAKNECONPRVTKN"])):.2f}' )
```

```
Consumption base and shock levels
Real values in 2030
Base value: 27,241,278.           Shocked value: 27,616,949.
Percent difference: 5.36
Real values in 2040
Base value: 38,692,815.           Shocked value: 38,693,167.
Percent difference: 0.0
```

11.2.5 Simulation with Add factors

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

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

In some ways they are very similar to a temporary exogenous shock. Both ways of producing the shock allow the shocked variable to respond endogenously in the period after the shock. The main difference between the two approaches is in an endogenous shock (add-factor shock), the equation remains active throughout, including during the period the variable is being shocked.

The intuition here for our previous consumption example might be that animal spirits cause households to increase consumption by 2.5 percent all things equal. However, all things are not equal – GDP is higher employment demand is higher which would boost consumption even more; but inflation is also higher which would reduce real incomes and suppress consumption. The net effect will balance these three (and other) factors out even in the shock period.

Box 5. Endogenous (Add-factor) shocks versus temporarily exogenized shocks

- **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 behavior is expected to change.
 - It also makes sense when actions of one part of an aggregate is likely to impact behavior 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.
 - * The final simulation level of the shocked variable during the period of the shock will be equal to the original level plus the shock, plus whatever endogenous additional changes in the shocked variable arise because the conditioning variables (the X_t in the equation) change.

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.
 - the final simulation level of the shocked variable during the period of the shock will be exactly equal to the original level plus the shock

Simulating the impact of a planned investment

The following simulation uses the add-factor to simulate the impact of a 3 year investment program beginning in 2025 of 1 percent of GDP per year. This might reflect a specific large scale plant that is being constructed due to a deal reached by the government with a foreign manufacturer. The add-factor approach is chosen because the additional investment is likely to increase demand for the products of other firms, which is likely to incite them to add to their investments as well – both after the shock as in previous examples **but also during the period that investment is being shocked**.

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 equal to the desired shock expressed as a percent of pre-shock GDP.

- **Level** then the add-factor will be a direct addition to the level of the dependent variable
- Convert the economic shock into the units of the add-factor
- 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 the shock was identified as a 1 percent of GDP increase in private-sector investment. A first step would be to determine the variable(s) that need to be shocked — private investment. To do this we can query the variable dictionary, in this case by listing all variables where `invest` is part of the variable description.

Note

Variable selection More on how to use wildcards to select variables can be found at [Variable selection with wildcharts](#)

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

| | |
|-----------------------|--------------------------------------|
| PAKNEGdifgovkn | : Pub investment real |
| PAKNEGdifprvkn | : Prvt. Investment real |
| PAKNEGdifprvkn_A | : Add factor:Prvt. Investment real |
| PAKNEGdifprvkn_D | : Fix dummy:Prvt. Investment real |
| PAKNEGdifprvkn_FITTED | : Fitted value:Prvt. Investment real |
| PAKNEGdifprvkn_X | : Fix value:Prvt. Investment real |
| PAKNEGdiftotkn | : Investment real |

Querying for all variables that “invest” in their descriptor gives us the mnemonic for the private investment variable `PAKNEGdifprvkn`.

Identify the functional form(s)

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

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

Use the `.eviews` command to identify the functional forms of the equation to be shocked.

```
mpak['PAKNEGdifprvkn'].eviews
```

```

PAKNEGDIFFPRVKN :
(PAKNEGDIFFPRVKN/PAKNEGDIKSTKNN( - 1)) = 0.00212272413966296 + 0.
↳ 970234989019907 * (PAKNEGDIFFPRVKN( - 1)/PAKNEGDIKSTKNN( - 2)) + (1 - 0.
↳ 970234989019907) * (DLOG(PAKNYGDPPOTLKN) + PAKDEPR) + 0.
↳ 0525240494260597*DLOG(PAKNEKRTTOTLCN/PAKNYGDPFCSTXN)

```

In this case the equation is written as a share of the capital stock in the preceding period **PAKNEGDIKSTKNN(-1)**.

Calculate the size of the required add factor shock

The shock to be executed is 1.0 percent of GDP.

It is assumed that the money will be spent in one year on private investment.

The private investment equation is written as a share of the capital stock lagged one period. Therefore, the add-factor needs to be shocked by adding 1 percent of GDP to private investment in 2028 divided by the capital stock in 2027.

```

#Create a DataFrame AFShock that is equal to the baseline
AFShock=bline
#Display the level of the AF
print("Pre shock levels")
AFShock.loc[2025:2030,['PAKNEGDIFFPRVKN_A','PAKNEGDIFFPRVKN','PAKNEGDIKSTKNN']]
#print (AFShock.loc[2025:2030,'PAKNEGDIFFPRVKN']/AFShock.loc[2025:2030,'PAKNYGDPMKTPKN
↳']*100)

```

Pre shock levels

| | PAKNEGDIFFPRVKN_A | PAKNEGDIFFPRVKN | PAKNEGDIKSTKNN |
|------|-------------------|-----------------|----------------|
| 2025 | -0.00 | 1602853.65 | 47303916.21 |
| 2026 | -0.00 | 1581104.20 | 48148785.83 |
| 2027 | -0.00 | 1569541.20 | 49009798.90 |
| 2028 | -0.00 | 1569140.97 | 49898686.55 |
| 2029 | -0.00 | 1580576.85 | 50826938.84 |
| 2030 | -0.00 | 1604394.55 | 51805897.58 |

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

```

AFShock=AFShock.mfcalc("""
<2028 2028> PAKNEGDIFFPRVKN_A = PAKNEGDIFFPRVKN_A + (.01*PAKNYGDPMKTPKN/PAKNEGDIKSTKNN(-
↳ 1))
""");
print("Shocked AF levels")
AFShock.loc[2025:2030,'PAKNEGDIFFPRVKN_A']

```

Shocked AF levels

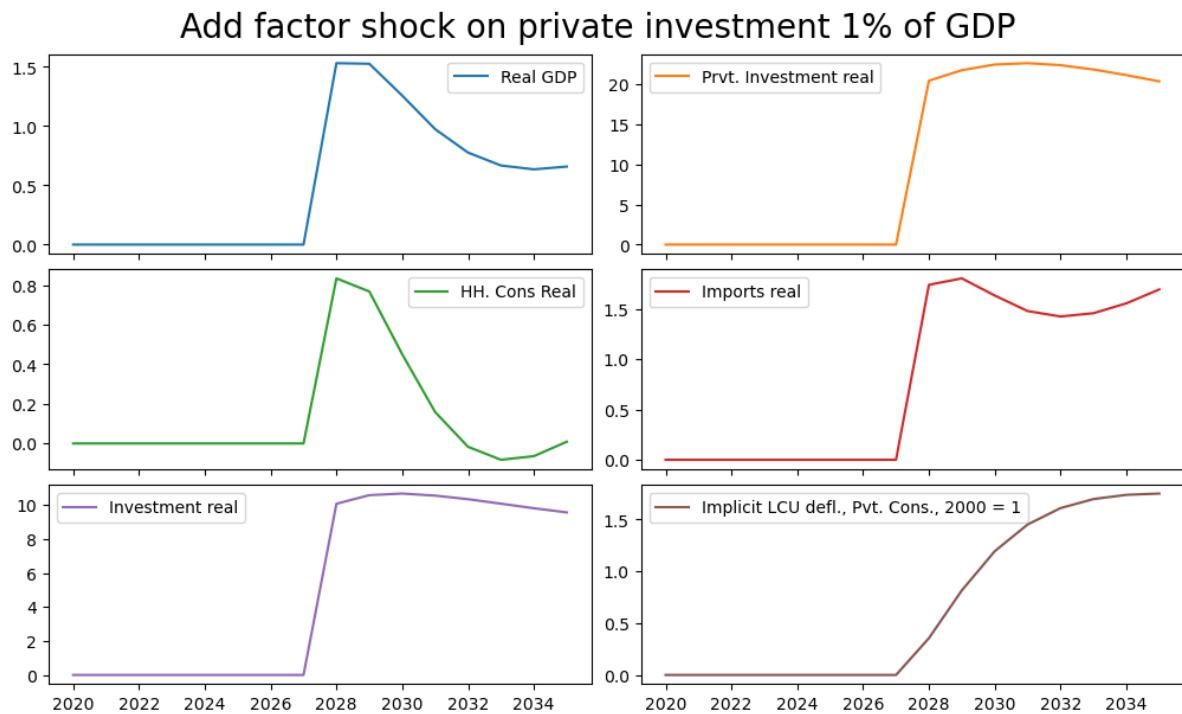
| | |
|------|-------|
| 2025 | -0.00 |
| 2026 | -0.00 |
| 2027 | -0.00 |
| 2028 | 0.01 |
| 2029 | -0.00 |
| 2030 | -0.00 |

Name: PAKNEGDIFFPRVKN_A, dtype: float64

Run the shock

The shock is executed by submitting the revised dataframe to the model object, and solving the model over the period 2020 through 2040.

```
# the simulation is done until 2050, for later use of the kept values
AFShockRes = mpak(AFSHOCK, 2020, 2050, keep='1% of GDP increase in FDI and private_
→investment (AF shock)')
mpak.smpl(2020, 2035)
(mpak[ 'PAKNYGDPMKTPKN PAKNEGdifprvkn PAKNECONPRVTKN PAKNEIMPGNFSKN PAKNEGdiftotkn_
→PAKNECONPRVTXN' ].difpctlevel.rename()).plot(title="Add factor shock on private investment 1% of GDP"));
```



The above graphs are expressed as a percent of the baseline value of each value. Because private investment in the baseline is only 5 percent of GDP, the 1% of GDP shock, expressed as a percent of private investment is much larger (about 20 times larger).

Below to double-check the calculations, two variables `IFPRVOLD_ORIG` and `IFTOT_ORIG` are created that reflect **pre-shock** private and total investment as a share of **pre-shock** GDP. Two additional variables `IFPRV_SHOCK` and `IFTOT_SHOCK` are also created as the shocked values of private and total investment as a percent of the original GDP. The difference between the two sets of variables is the increase in fixed private and fixed total investment as a percent of the same denominator (the original level of GDP).

The ex-post change in private investment and of total investment in 2028 is 1.06 and 1.11 percent respectively, 0.6 and 1.1 percentage points larger than the actual shock 1 percent of GDP shock to the addfactor.

This difference represents the endogenous reaction of other investors in the same time period to the changed circumstances. It is precisely to capture this effect that an endogenous or add-factor shock is employed.

```

AFShockRes['GDPOLD']=bline['PAKNYGDPMKTPKN']
AFShockRes['IFPRVOLD']=bline['PAKNEGDIFFPRVKN']
AFShockRes['IFTOTOLD']=bline['PAKNEGDIFFTOTKN']
AFShockRes=AFShockRes.mfcalc('''
    IFPRV_SHOCK = PAKNEGDIFFPRVKN / GDPOLD*100
    IFTOT_SHOCK = PAKNEGDIFFTOTKN / GDPOLD*100
    IFPRV_Orig = IFPRVOLD / GDPOLD*100
    IFTOT_Orig = IFTOTOLD / GDPOLD*100
    IFPRV_IMPACT = IFPRV_SHOCK - IFPRV_Orig
    IFTOT_IMPACT = IFTOT_SHOCK - IFTOT_Orig
    ''')
print(round(AFSHockRes.loc[2025:2030, ['IFPRV_ORIG', 'IFPRV_SHOCK',
    'IFTOT_ORIG', 'IFTOT_SHOCK',
    'IFPRV_IMPACT', 'IFTOT_IMPACT']], 2))

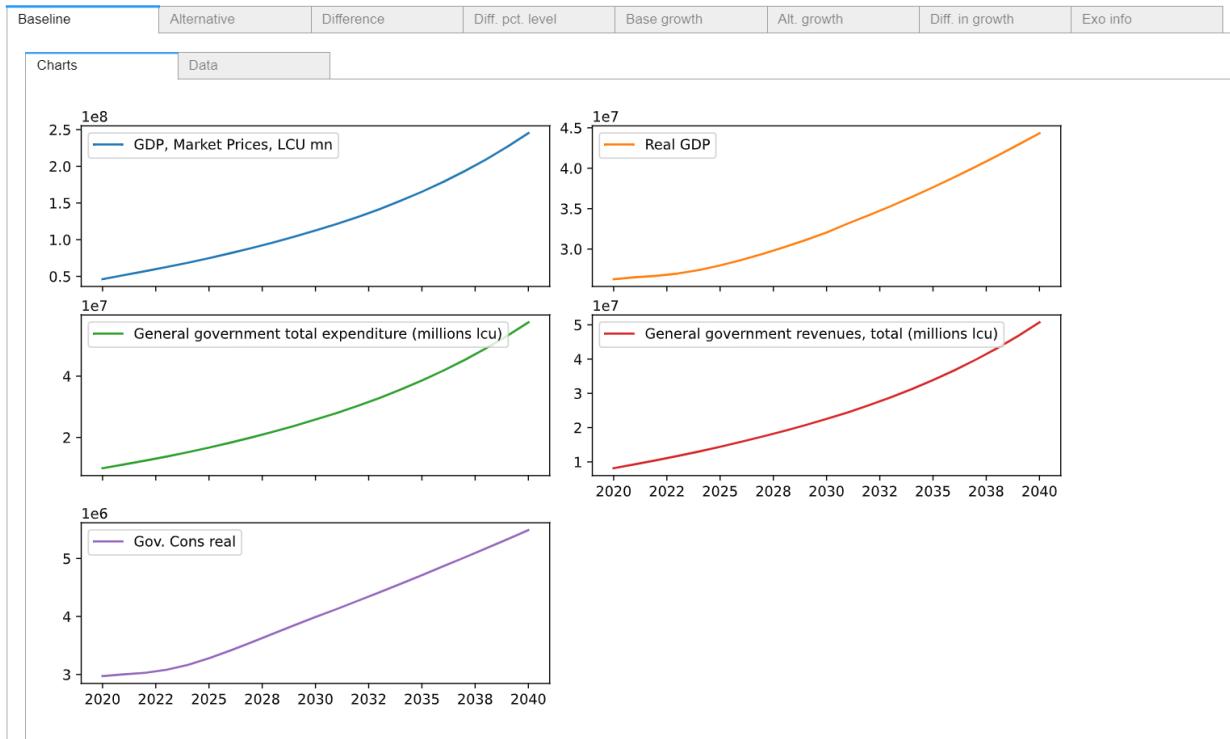
```

| | IFPRV_ORIG | IFPRV_SHOCK | IFTOT_ORIG | IFTOT_SHOCK | IFPRV_IMPACT | IFTOT_IMPACT |
|------|------------|-------------|------------|-------------|--------------|--------------|
| 2025 | 5.73 | 5.73 | 11.31 | 11.31 | 0.00 | 0.00 |
| 2026 | 5.52 | 5.52 | 11.21 | 11.21 | 0.00 | 0.00 |
| 2027 | 5.34 | 5.34 | 11.12 | 11.12 | 0.00 | 0.00 |
| 2028 | 5.19 | 6.25 | 11.05 | 12.16 | 1.06 | 1.11 |
| 2029 | 5.08 | 6.19 | 11.01 | 12.17 | 1.10 | 1.16 |
| 2030 | 5.01 | 6.13 | 10.99 | 12.16 | 1.12 | 1.17 |

11.3 The results visualization widget view

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

```
display(mpak['PAKNYGDPMKTPCN PAKNYGDPMKTPKN PAKGGEXPTOTLCN PAKGGREVTOTLCN_
→PAKNECONGOVTKN'])
```



11.4 Save simulation results to a pcim file for later exploration

The next chapter explores the different results generated during these simulations. Rather than re-run them, the model object `mpak` (and the simulation results that are stored by the `keep` command used above) are saved to a local file for retrieval in the next chapter.

NB: The `keep=True` instructs ModelFlow to save the results from any kept solutions as well.

```
help(mpak.modeldump)
```

```
Help on method modeldump in module modelclass:  
modeldump(file_path='', keep=False, zip_file=True) method of modelclass.model_
    ↪instance
```

```
mpak.modeldump(r'../models/mpakw.pcim', keep=True)
```

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

```
dict_keys(['Baseline', '$25 increase in oil prices 2025-27', '2.5% increase in C_2025-40', '2.5% increase in C 2025-27 -- exog whole period', '2.5% increase in C_2025-27 -- exog whole period --KG=True', '2.5% increase in C 2025-27 -- temporarily exogenized', '1% of GDP increase in FDI and private investment (AF_shock)'])
```

MORE COMPLEX SCENARIOS

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

The particular scenario to be examined is the introduction of a Carbon Tax. The model used, and the example presented are both taken from the model of Pakistan presented in [Burns et al.\(2021\)](#).

1 In this chapter - More Complex Scenarios

This chapter presents some more complex scenarios, and illustrates how to develop a script that introduces changes to the model.

A complex scenario is developed as deficiencies in initial scenarios are unearthed. Scenarios presented include:

- 1) The introduction of a simple carbon tax (a simple shock of an exogenous variable) three exogenous variables in this case.
- 2) Upon examining results, it is recognized that the initial nominal shocks loses impact over time because of inflation. Therefore, the shock is re-estimated as an ex-ante real shock (i.e. the initial shock is up-scaled over time in line with inflation to keep its real value constant).
- 3) This example is judged imperfect because it does not account for the in-scenario inflation impacts. In a third scenario, the model equations are adjusted so that the ex-post inflation rate is used to maintain the real-value of the Carbon tax.

The chapter also illustrates how to modify the description of variables in a scenario, and illustrates various techniques for visualizing scenario results.

12.1 Load a pre-existing model, data and descriptions

After initializing a ModelFlow pandas session in the usual way, the Pakistan model, which is comprised of the model object, its estimated equations and the data is loaded. The pcim file was created by the World Bank from a slightly modified version of the original EViews model used in the paper ([Burns et al.,2021](#)).

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

```
Zipped file read: ..\models\pak.pcim
```

12.2 The policy problem

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

The Pakistan model contains three carbon tax variables:

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

As discussed in earlier chapters the meaning of the mnemonics can be retrieved from the model object `mpak` using the `.des` method and a wild-card search.

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

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

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

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

```
PAKCCEMISCO2TKN : Total Carbon emissions (tons)
PAKGGREVCO2CER : Carbon tax on coal (USD/t)
PAKGGREVCO2GER : Carbon tax on gas (USD/t)
PAKGGREVCO2OER : Carbon tax on oil (USD/t)
```

Techniques to query the model object for the meaning of mnemonics or the mnemonics associated with economic concepts are discussed in more detail here.¹.

12.3 Add variable descriptions

A ModelFlow model imported from EViews will inherit the variable descriptors coming from Eviews. The variable descriptors are stored in a dictionary named: `.var_description`. Not all EViews variables will necessarily have a description so additional definitions (descriptions) may need to be provided.

Below we define a python dictionary in the same format as the dictionary `var_description` that is contained in the model object `mpak`. Each dictionary entry is comprised of a key (the variables mnemonic) and a value (the description of the variable).

¹ Techniques to identify mnemocs that correspond to economic ideas are explored here: [Variable selection with wildcharts](#)

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

These new definitions can be merged with the existing description by using the `|` operator.

The command

```
mpak.var_description = mpak.var_description | extra_description
```

Sets `mpak.var_description` to a merge between the contents of the dictionary: `mpak.var_description` (its current content) and the dictionary: `extra_description`

Following execution, the `.var_description` will be changed to contain both its old values and those added in the `extra_description` dictionary defined above.

```
mpak.var_description = mpak.var_description | extra_description
```

Several ModelFlow methods take advantage of this dictionary to provide more reader-friendly descriptions of variables – typically through the `rename` option. For those methods that define it, if `rename` is set to True, the method will substitute the description for the variable name in any outputs.

Variables with descriptions can also be selected for by using the `mpak['!*subtext*']` syntax, where `subtext` is some text that appears in the variable descriptor.

12.4 Simulating the impact of imposing a carbon price

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

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

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

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

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

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

12.4.1 Solve the model

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

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

Note

This simulation is a shock on an exogenous variable (the first kind of shock discussed in the previous chapter), although in this case the shock is applied to three exogenous variables simultaneously, whereas in earlier examples only one variable was shocked.

Examining the results

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

The `bline`, `.basedf` and the original `.lastdf` Dataframes were created when the model was initially loaded and solved. The `resultsdf` database and a revised `.lastdf` were generated when the model was solved for the new carbon prices.

Note

The standard dataframes are part of the ModelFlow object and managed by it.

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

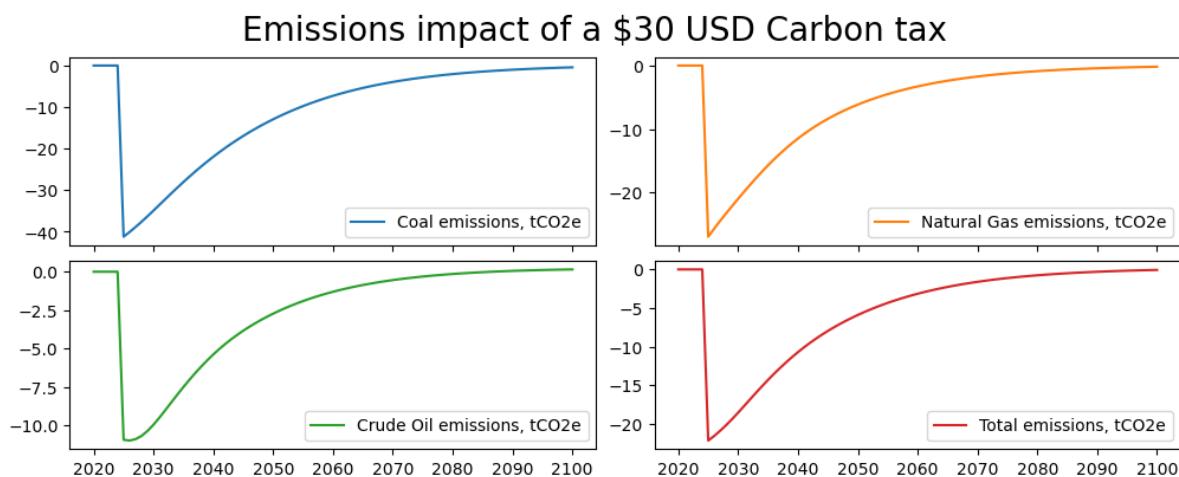
The command below shows the results of the simulation on the four emissions variables in the model expressed as a percent deviation from the level of the baseline (the `.difpctlevel` operator below), and where the mnemonics have been replaced by their descriptions using the `.rename` option.

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

| | Coal emissions, tCO ₂ e | Natural Gas emissions, tCO ₂ e | \ |
|------|---|---|---|
| 2023 | 0.00 | 0.00 | |
| 2024 | 0.00 | 0.00 | |
| 2025 | -41.19 | -26.99 | |
| 2026 | -40.06 | -25.72 | |
| 2027 | -38.85 | -24.48 | |
| 2028 | -37.59 | -23.30 | |
| 2029 | -36.26 | -22.14 | |
| 2030 | -34.89 | -21.01 | |
| | Crude Oil emissions, tCO ₂ e | Total emissions, tCO ₂ e | |
| 2023 | 0.00 | 0.00 | |
| 2024 | 0.00 | 0.00 | |
| 2025 | -10.93 | -22.17 | |
| 2026 | -10.98 | -21.56 | |
| 2027 | -10.89 | -20.89 | |
| 2028 | -10.68 | -20.17 | |
| 2029 | -10.35 | -19.38 | |
| 2030 | -9.95 | -18.55 | |

The impact of the imposition of the carbon tax in the model is relatively quick, resulting in an overall decline in emissions of 22.2% in the first year, with coal emissions (coal is a relatively carbon intensive source of energy so harder hit by the carbon tax) recording the biggest hit at -41.2 percent.

```
mpak['PAKCEMISCO2?KN'].difpctlevel.rename().plot(title="Emissions impact of a $30 USD Carbon tax");
```



Abstracting from the fact that the impact is occurring too quickly (it would take time for the substitution towards alternative sources of power to occur), the fact that impacts are fading with time suggests an error in the specification of the shock.

Indeed, high domestic inflation means that the real price change of the \$30 nominal increase in the Carbon price is declining over time – suggesting that the scenario needs tweaking.

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

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

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

Finally the model is re-solved.

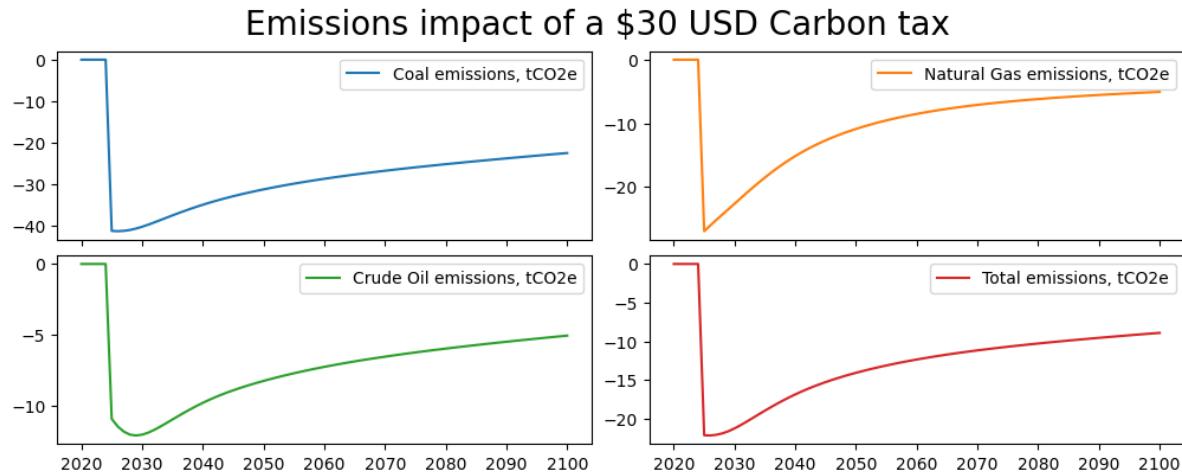
```
CT30realdf = bline.copy()
CT30realdf=CT30realdf.upd("<2025 2025> PAKGGREVCO2CER PAKGGREVCO2OER PAKGGREVCO2GER ="
→30")
#NB: Variables used
#   PAKNECONPRVTXN is the consumer price deflator
#   PAKPANUSATLS id the USD exchange rate
CT30realdf=CT30realdf.mfcalc(''
<2026 2100> PAKGGREVCO2CER = PAKGGREVCO2CER(-
→1) * (PAKNECONPRVTXN*PAKPANUSATLS) / (PAKNECONPRVTXN(-1)*PAKPANUSATLS(-1))
PAKGGREVCO2OER = PAKGGREVCO2OER(-
→1) * (PAKNECONPRVTXN*PAKPANUSATLS) / (PAKNECONPRVTXN(-1)*PAKPANUSATLS(-1))
PAKGGREVCO2GER = PAKGGREVCO2CER(-
→1) * (PAKNECONPRVTXN*PAKPANUSATLS) / (PAKNECONPRVTXN(-1)*PAKPANUSATLS(-1))
'')
CT30realdf.loc[2023:2030,'PAKGGREVCO2CER']
resultsdf = mpak(CT30realdf,2020,2100,keep="Ex ante Real $30USD Carbon tax") #_
→simulates the model
```

The above code first sets the Carbon prices to 30USD and then grows them at the same rate as inflation. Below we see that with inflation of 30% per annum the domestic carbon price is rising rapidly.

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

| | Carbon tax on coal (USD/t) | Carbon tax on gas (USD/t) |
|------|----------------------------|---------------------------|
| 2023 | -5.55 | -41.00 |
| 2024 | -5.55 | -41.00 |
| 2025 | 30.00 | 30.00 |
| 2026 | 31.83 | 31.83 |
| 2027 | 33.64 | 33.64 |
| 2028 | 35.45 | 35.45 |
| 2029 | 37.26 | 37.26 |
| 2030 | 39.09 | 39.09 |
| | Carbon tax on oil (USD/t) | |
| 2023 | -8.71 | |
| 2024 | -8.71 | |
| 2025 | 30.00 | |
| 2026 | 31.83 | |
| 2027 | 33.64 | |
| 2028 | 35.45 | |
| 2029 | 37.26 | |
| 2030 | 39.09 | |

```
mpak['PAKCCEMISCO2?KN'].difpctlevel.rename().plot(title="Emissions impact of a $30_
→USD Carbon tax");
```



These results are better, but still there is an erosion of the effect of the tax.

On introspection, this is likely due to the fact that the carbon tax itself is inflationary. As a result, prices probably rose to a higher level than supposed by the ex ante calculation.

To deal with this, a different approach is needed. Rather than maintaining the carbon price as an exogenous variable, instead it should be made an endogenous variable by changing the model and adding equations for all of the carbon tax variables.

Before doing so lets save the current version of the model for further work later.

```
mpak.modeldump('..\models\pakCarbonTaxScenarios.pcim')
```

12.6 Changing the model – modifying and or adding equations

To endogenize the carbon price, an equation for each carbon price has to be added to the model. This can be done with the `.equpdate()` method.

```
#Reload original model and data
mpak1,bline = model.modelload('..\models\pak.pcim',alfa=0.7,run=1,keep="Baseline")
#Create a new model object mpakreal and new baseline dataframe-- baselinereal
#The nominal carbon taxes (expressed in USD) are now endogenous
#and increase with domestic inflation and the exchange rate
mpakreal,blinereal = mpak1.equpdate('''
<fixable> PAKGGREVC02CER = PAKGGREVCO2CER(-1) * (PAKNYGDPMKTPXN*PAKPANUSATLS) /_
↳(PAKNYGDPMKTPXN(-1)*PAKPANUSATLS(-1))
<fixable> PAKGGREVC02OER = PAKGGREVCO2OER(-1) * (PAKNYGDPMKTPXN*PAKPANUSATLS) /_
↳(PAKNYGDPMKTPXN(-1)*PAKPANUSATLS(-1))
<fixable> PAKGGREVC02GER = PAKGGREVCO2GER(-1) * (PAKNYGDPMKTPXN*PAKPANUSATLS) /_
↳(PAKNYGDPMKTPXN(-1)*PAKPANUSATLS(-1))
'',add_add_factor=False, calc_add=False,newname='Pak model, with real Carbon price_
↳equations')
```

```
Zipped file read: ..\models\pak.pcim
```

```
The model:"PAK" got new equations, new model name is:"Pak model, with real Carbon_
↳price equations"
New equation for For PAKGGREVCO2CER
```

(continues on next page)

(continued from previous page)

```

Old frml  :new endogeneous variable
New frml  :FRML <fixable> PAKGGREVC02CER = (PAKGGREVC02CER (-
    ↵1) * (PAKNYGDPMKTPXN*PAKPANUSATLS) / (PAKNYGDPMKTPXN (-1) *PAKPANUSATLS (-1))) * (1-
    ↵PAKGGREVC02CER_D) + PAKGGREVC02CER_X*PAKGGREVC02CER_D$
Adjust calc:No frml for adjustment calc
New equation for For PAKGGREVC02OER
Old frml  :new endogeneous variable
New frml  :FRML <fixable> PAKGGREVC02OER = (PAKGGREVC02OER (-
    ↵1) * (PAKNYGDPMKTPXN*PAKPANUSATLS) / (PAKNYGDPMKTPXN (-1) *PAKPANUSATLS (-1))) * (1-
    ↵PAKGGREVC02OER_D) + PAKGGREVC02OER_X*PAKGGREVC02OER_D$
Adjust calc:No frml for adjustment calc
New equation for For PAKGGREVC02GER
Old frml  :new endogeneous variable
New frml  :FRML <fixable> PAKGGREVC02GER = (PAKGGREVC02GER (-
    ↵1) * (PAKNYGDPMKTPXN*PAKPANUSATLS) / (PAKNYGDPMKTPXN (-1) *PAKPANUSATLS (-1))) * (1-
    ↵PAKGGREVC02GER_D) + PAKGGREVC02GER_X*PAKGGREVC02GER_D$
Adjust calc:No frml for adjustment calc

```

As written, the `.equupdate()` command creates a new model, which is a copy of the existing model with three new equations.

Each equation grows the nominal rate of the carbon tax at the same rate as *ex post* inflation (PAKNECONPRVTXN) converted into USD via the exchange rate PAKPANUSATLS. The equations are introduced as exogenizable equations (as distinct from an identity which must always hold) by adding the `<fixable>` prefix to each equation. The equations are not estimated, so no add-factors are included in the equations.

The output for the `.equupdate()` reports the actual formulae included in the model.

```

New equation for For PAKGGREVC02CER
Old frml  :new endogeneous variable
New frml  :FRML <fixable> PAKGGREVC02CER = (PAKGGREVC02CER (-
    ↵1) * (PAKNECONPRVTXN*PAKPANUSATLS) / (PAKNECONPRVTXN (-1) *PAKPANUSATLS (-1))) * (1-
    ↵PAKGGREVC02CER_D) + PAKGGREVC02CER_X*PAKGGREVC02CER_D$
Adjust calc:No frml for adjustment calc

```

Note that because the equations are to be fixable, an `_X` and `_D` variable are added to the specified equations. Combined they effectively split each equation into two:

1. the specified equations when `_D` equals zero
2. equal to `_X` when the `_D` equals one.

The newly created model is given the name mpakreal and is given a text description.

Following the addition of the equations, the new variables (`_D` and `_X`) must be initialized. The `_X` variables are made equal to the current values of the various tax rates, while the `_D` is set to 1 everywhere – effectively turning the equation off and re-creating the same situation as the initial model where the tax rates are fully exogenous.

As with the mpak model the variable descriptions need to be updated.

```
mpakreal.var_description = mpakreal.var_description | extra_description
```

```
#Exogenizes the newly added equations and sets the dummy =1 and the _x to the current_
↪value of the dependent variable
bline_real=mpakreal.fix(blinereal,'PAKGGREVC02CER PAKGGREVC02GER PAKGGREVC02OER')
```

The following variables are fixed
PAKGGREVC02CER

(continues on next page)

(continued from previous page)

```
PAKGGREVC02GER
PAKGGREVC02OER
```

Finally the new model is solved, the result is kept in a new baseline and a quick check ensures that the model did indeed reproduce the data that it was originally fed, including the initial Carbon Tax levels.

```
#Solve the model for the new baseline
res = mpakreal(bline_real, 2021, 2100, alfa=0.5, keep='Baseline - adjusted model')
mpakreal['PAKNYGDPMKTPKN PAKNECONPRVTXN PAKGGBALOVR PAKGGREVC02CER PAKCCEMISCO2TKN'].
    ↪difpctlevel.rename().df
```

| | GDP Implicit LCU defl., Pvt. Cons., 2000 = 1 \ | |
|------|---|------|
| 2021 | 0.00 | 0.00 |
| 2022 | 0.00 | 0.00 |
| 2023 | 0.00 | 0.00 |
| 2024 | 0.00 | 0.00 |
| 2025 | 0.00 | 0.00 |
| ... | ... | ... |
| 2096 | 0.00 | 0.00 |
| 2097 | 0.00 | 0.00 |
| 2098 | 0.00 | 0.00 |
| 2099 | 0.00 | 0.00 |
| 2100 | 0.00 | 0.00 |
| | Carbon tax on coal (USD/t) Total emissions, tCO2e | |
| 2021 | -0.00 | 0.00 |
| 2022 | -0.00 | 0.00 |
| 2023 | -0.00 | 0.00 |
| 2024 | -0.00 | 0.00 |
| 2025 | -0.00 | 0.00 |
| ... | ... | ... |
| 2096 | -0.00 | 0.00 |
| 2097 | -0.00 | 0.00 |
| 2098 | -0.00 | 0.00 |
| 2099 | -0.00 | 0.00 |
| 2100 | -0.00 | 0.00 |

[80 rows x 4 columns]

12.6.1 Solving the revised model

With the new model generated, it can now be solved with the real tax rate endogenized in the forecast period. This involves three steps.

1. Set the nominal tax rate to 30 in 2025
2. Now Endogenize the equation for the rest of the period
3. Solve the model.

```
scenario_real_CTax = bline_real.upd('''
<2025 2025>
PAKGGREVC02CER_X PAKGGREVC02GER_X PAKGGREVC02OER_X = 30 # Sets the exogenous value to_
↪30 in 2025
<2026 2100 >
PAKGGREVC02CER_D PAKGGREVC02GER_D PAKGGREVC02OER_D = 0 # Endogenizes the new_
↪equations for the rest of time so that the real-rate stays at 30USD
```

(continues on next page)

(continued from previous page)

```
' ''')
= mpakreal(scenario_real_CTax, 2021, 2100, alfa=0.5, keep='Real model real tax = 30 at
→2025 prices and exchange rates')
```

Initially the Carbon tax comes in at 30 but gradually its rate in USD rises in line with inflation such that it reaches \$1358 by 2100.

```
mpakreal['PAKGGREVCO2?ER PAKNECONPRVTXN PAKNYGDPMKTPXN'].rename().df
```

| | Carbon tax on coal (USD/t) | Carbon tax on gas (USD/t) | \ |
|-----------------------|---|--|---|
| 2021 | -5.55 | -41.00 | |
| 2022 | -5.55 | -41.00 | |
| 2023 | -5.55 | -41.00 | |
| 2024 | -5.55 | -41.00 | |
| 2025 | 30.00 | 30.00 | |
| ... | ... | ... | |
| 2096 | 1098.50 | 1098.50 | |
| 2097 | 1158.34 | 1158.34 | |
| 2098 | 1221.44 | 1221.44 | |
| 2099 | 1287.97 | 1287.97 | |
| 2100 | 1358.11 | 1358.11 | |
| | Carbon tax on oil (USD/t) | Implicit LCU defl., Pvt. Cons., 2000 = 1 | \ |
| 2021 | -8.71 | 1.82 | |
| 2022 | -8.71 | 1.98 | |
| 2023 | -8.71 | 2.14 | |
| 2024 | -8.71 | 2.30 | |
| 2025 | 30.00 | 2.51 | |
| ... | ... | ... | |
| 2096 | 1098.50 | 106.62 | |
| 2097 | 1158.34 | 112.69 | |
| 2098 | 1221.44 | 119.11 | |
| 2099 | 1287.97 | 125.89 | |
| 2100 | 1358.11 | 133.06 | |
| | GDP, Marker Prices, LCU Price defl., 2000 = 1 | | |
| 2021 | | 1.95 | |
| 2022 | | 2.14 | |
| 2023 | | 2.32 | |
| 2024 | | 2.50 | |
| 2025 | | 2.72 | |
| ... | ... | | |
| 2096 | | 112.39 | |
| 2097 | | 118.75 | |
| 2098 | | 125.47 | |
| 2099 | | 132.58 | |
| 2100 | | 140.08 | |
| [80 rows x 5 columns] | | | |

This seemingly very high level is just a reflection of the 75 years of inflation that compounded require a much higher nominal Carbon tax rate to have the same relative price effect. The cumulative effect of inflation in the range of 5.5 real per annum causes the price level to increase 74 times (7400 percent increase 133/1.8 from fourth data column in the above table).

The table below shows the same data but in growth rate terms – indicating that the nominal Carbon tax rate is gradually rising each year in line domestic inflation adjusted for the exchange rate – i.e. it is constant in real terms.

```
mpakreal['PAKGGREVCO2?ER PAKPANUSATLS PAKNYGDPMKTPXN'].pct.rename().df
```

```

Carbon tax on coal (USD/t) Carbon tax on gas (USD/t) \
2021          0.00          0.00
2022          0.00          0.00
2023          0.00          0.00
2024          0.00          0.00
2025        -640.56        -173.17
...
2096         5.45         5.45
2097         5.45         5.45
2098         5.45         5.45
2099         5.45         5.45
2100         5.45         5.45
Carbon tax on oil (USD/t) Exchange rate LCU / US$ - Pakistan \
2021          0.00        -0.16
2022          0.00        -0.16
2023          0.00        -0.14
2024          0.00        -0.12
2025       -444.41        -0.30
...
2096         5.45        -0.20
2097         5.45        -0.20
2098         5.45        -0.20
2099         5.45        -0.20
2100         5.45        -0.20
GDP, Marker Prices, LCU Price defl., 2000 = 1
2021          10.69
2022           9.76
2023           8.71
2024           7.76
2025           8.90
...
2096          5.66
2097          5.66
2098          5.66
2099          5.66
2100          5.66
[80 rows x 5 columns]

```

12.6.2 Results

The results from the simulation with the Carbon Tax rate endogenized so as to maintain its real value over time, are broadly consistent with the results from the ex ante real scenario performed above.

```
mpakreal['PAKCCEMISCO2?KN'].difpctlevel.rename().df
```

```

Coal emissions, tCO2e Natural Gas emissions, tCO2e \
2021          0.00          0.00
2022          0.00          0.00
2023          0.00          0.00
2024          0.00          0.00
2025        -41.19        -26.99
...
2096       -24.23       -10.33
2097       -24.09       -10.25
2098       -23.95       -10.18

```

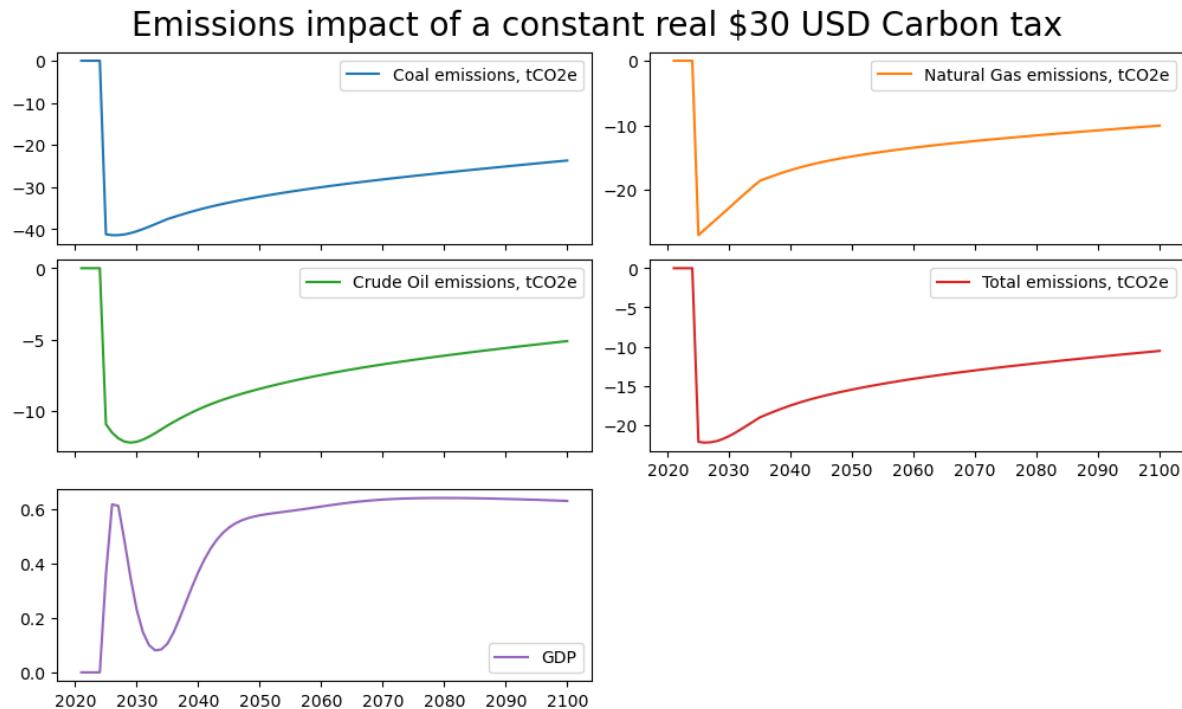
(continues on next page)

(continued from previous page)

| | | |
|----------------------------|------------------------|--------|
| 2099 | -23.81 | -10.11 |
| 2100 | -23.67 | -10.04 |
| Crude Oil emissions, tCO2e | Total emissions, tCO2e | |
| 2021 | 0.00 | 0.00 |
| 2022 | 0.00 | 0.00 |
| 2023 | 0.00 | 0.00 |
| 2024 | 0.00 | 0.00 |
| 2025 | -10.93 | -22.17 |
| ... | ... | ... |
| 2096 | -5.30 | -10.85 |
| 2097 | -5.25 | -10.78 |
| 2098 | -5.20 | -10.70 |
| 2099 | -5.15 | -10.63 |
| 2100 | -5.11 | -10.55 |

[80 rows x 4 columns]

```
(mpakreal['PAKCCEMISCO2?KN PAKNYGDPMKTPKN'].difpctlevel.  
rename().plot(title="Emissions impact of a constant real $30 USD Carbon tax"));
```



The modified model, which preserves the real value of the carbon tax has a permanent and substantial negative effect on emissions. The impact is not at an unchanging level, reflecting in part adaptation within the economy. Of particular import is what is being done with the revenues from the Carbon tax. As the structure of GDP (shifts to less carbon intensive activities) carbon tax revenues fall even as the Carbon Tax rate rises.

Note, the percent deviation of emissions tends to decrease over time in this scenario principally because the level of activity (GDP) is higher with the Carbon Tax than without it. As a result, emissions are higher than they would have been had GDP remained unchanged. For the Pakistan model, the positive impact of the Carbon tax is mainly explained by co-benefits, notably reduced reliance on more distortionary taxes such as payroll taxes and because of reductions in informality and due to health and productivity benefits from lower pollution levels (see [Burns et al. \(2021\)](#) for more details how these effects operate in the Pakistan model).

A SIMULATION THAT TARGETS A SPECIFIC OUTCOME

Normally, when working with a model, the trajectory of the exogenous variables determine the trajectory of the endogenous (left hand side) variables. However, sometimes it can be useful to reverse the causality in the model, to find a trajectory for some exogenous variables that will result in certain endogenous variables achieving specific values.

A simple example might be where a policy maker wants to know what level of Carbon Tax, if implemented in a budget neutral fashion, would be needed to achieve a specific level of emissions. As specified, the analytical question is to achieve two targets:

1. A specific CO^2 emissions trajectory
2. An unchanged fiscal deficit

In this Chapter - Targeting Specific Outcomes

This chapter illustrates how to target specific outcomes using ModelFlow. In the example, a specific outcome for carbon emissions is introduced and instruments (carbon taxes) are specified for achieving the target.

The chapter introduces a ModelFlow routine that searches for the values of the instruments (the carbon tax) that achieve the desired target level of carbon emissions (the target).

A second example illustrates a scenario with two targets (one a specific trajectory for emissions as above) and the second the stipulation that the policy be introduced in a budget neutral manner (a second target of an unchanged budget deficit) to be achieved by re-cycling Carbon tax revenues as transfers (a second instrument) to households.

In addition to these examples, techniques for fine-tuning the process in instances where the default parametrization of the system fails to find a result are also presented.

13.1 Targeting in ModelFlow

Targeting in ModelFlow requires that there be **at least as many instruments as there are targets**. So in the above example two instruments would be required.

The instrument to achieve the emissions could be a Carbon tax (applied uniformly on emissions from coal, gas and crude oil). The instrument to achieve an unchanged fiscal deficit, could be government spending or some form of revenue, say taxes on labor.

To illustrate targeting, the climate-aware model of Pakistan (Burns *et al.* (2021)) is used.

To run a targeting solution, the following main steps must be undertaken

1. Initialize a ModelFlow python session.
2. Load the model and data

3. Create a baseline
4. Define instrument variables (one instrument can consist of several variables).
5. Define a dataframe with the trajectory of the target variables.
6. Solve the problem using the `.invert` method.
7. Visualize the results

13.2 Load a model, data and descriptions

Following the initialization of the python session, the model is loaded and a model object mpak declared. The file pak.pcim contains the model object for the Pakistan model described in Burns (2021), including all of the equations data and variables.

```
mpak,initial = model.modelload('..\\models\\pak.pcim')
```

```
Zipped file read: ..\\models\\pak.pcim
```

```
mpak.var_description = mpak.var_description | {  
    'PAKCCEMISCO2TKN' : 'Pakistan Total Carbon emissions (tons)'  
}
```

13.3 Solve the model to create a baseline

Next the model is solved, using the initial dataframe that was generated on the modelload.

In this instance, the model is solved from 2022 through 2100. The option `ljit=True` tells the model object to compile the model. Model compilation takes time, but once a model is compiled it will solve faster. When a model will be solved multiple times, the additional time required to compile the model, can be made up by the faster execution time each time the model is solved. Targeting requires the model to be solved many times, which makes the additional overhead of compilation worthwhile.

```
baseline = mpak(initial,2022,2100,alfa=0.7,silent=1,ljit=True)  
mpak.basedf = baseline.copy()
```

Box 6. Compilation of a model

Python is an interpreted language, so solving can be improved by compiling the solving routines to machine code. The [Numba package](https://numba.pydata.org/) (<https://numba.pydata.org/>) can help overcome this by translating Python code into machine code, significantly speeding up computation. The extent of this speed improvement depends on the complexity of the model and the computational resources required.

When solving a model, you can enable Numba compilation with by setting the option `l jit=True`. This setting cause the model to be compiled. The first time it is used for a model, the compilation will take some time. The compiled code is cached in the `modelsource` subfolder. As a result, subsequent runs of the model — even across different Python sessions — will use the precompiled model and solve much faster than the uncompiled versions.

Note: If there are issues with the solution, error messages may not be visible when the model is compiled. To troubleshoot, set `l jit` to `False` and rerun the model.

If you recreate a model with different logic but retain the same model name, consider clearing the `model-source` subfolder to remove any cached compiled code that could interfere with the new model version.

13.4 Target CO₂ emission - a very simple example

To illustrate the process for solving the model in targeting mode, an initial very simple example is set up, where **only emissions** are targeted (one target), and the objective is to have emissions grow at an annual rate of 1 percent.

13.4.1 Define targets

Having loaded the model, created a baseline and designated the target (steps 1-3 from above), the next step is to create the target variable(s).

This is done below by defining two series. The first, `target_before_simple`, is set equal to the value of the total emissions variable `PAKCCEMISCO2TKN` in the baseline. The second, `target_simple`, is the target for the same variable.

For this example we assume policy makers are looking for a carbon price that will restrict emissions growth to 1 percent per annum. The `target_simple` variable is therefore set equal to `target_before_simple` through 2024 and then, using the `.upd()` method, grown slowly at 1 percent per year afterwards.

```
# Create dataframe with only the target variable (for memory)
target_before_simple = baseline.loc[2024:2100, ['PAKCCEMISCO2TKN']]
# create a target as a copy of the before variable through 2024 and then
# having it grow by 1 percent per annum from 2025 onwards.
target_simple = target_before_simple.upd(f'<2025 2100> PAKCCEMISCO2TKN =growth 1')
```

13.4.2 Define instruments

As noted above, for each target there must be one instrument. However, **one instrument** can consist of **several instrument variables**. Below the instrument is defined as three instrument variables: the carbon tax rate on each of Oil, Natural Gas and Coal. A list `instruments_simple` is created comprised of the mnemonics of each of the three variables. The three variables together are considered one instrument.

```
instruments_simple = [['PAKGGREVC02CER', 'PAKGGREVC02GER', 'PAKGGREVC02OER']]
```

13.4.3 Finding the values for the instruments that achieve the desired target

To find the value for the instrument(s) that achieves the target of a 1 percent growth in emissions, the `ModelFlow.invert()` method is used: passing it the baseline database, the target `dataframe` that we defined above and the list of the instruments to use.

The `.invert()` method will solve the model once for each of the 77 years of the active sample period, and will solve multiple times for each year until it finds a set of instrument values that result in the desired targets. The dataframe containing the solution to the targeting problem is returned by the `.invert()` method and also stored in the `.lastdf` internal dataframe.

Invert options

The `.invert()` method has many options. The first three define the problem, the remainder influence how the solver operates. Finding the right options for a given problem is not always straightforward. The following table provides some hints on how to work with these options to optimize solution speed for a given problem.

| Option | Parameters | Explanation |
|---------------|-------------------|---|
| databank | name of dataframe | Dictates the initial conditions of the model (same as in a normal solve) |
| targets | list | A list of the variables to be targeted |
| instruments | list | A list of the variables that will be used as instruments to achieve the targets. Individual instruments may have multiple variables (as in the example above). Instrument list may contain Impulse parameters specific to the instrument (see below). |
| silent | bool | True: Suppress detailed outputs; False: Show detailed outputs (one line per iteration per year) |
| defaultimpuls | float | Determines the size of the change in instruments as the model searches for answers. Choose a value relative to the size of the actual series being modified. |
| defaultconv | float | Specifies the amount by which targeted variables may deviate from the target value and still be considered a solution. Should reflect the size of the target. |
| delay | Integer | Causes the instrument value changed to be lagged N periods from the target period being solved; For WBG models this should almost always be 0 |
| varimpulse | bool | True: Sets the initial change in the instrument for the future to the same as in the most recently solved period. This will greatly speed solves where instruments are expected to evolve smoothly. |
| nonlin | integer | N: Jacobian will be updated after N iterations without a solution; Most WBG models are near-linear so setting to 0 will solve faster. If the solution fails, try non-linear=5 |
| maxiter | integer | Maximum number of Newton iterations; If passed model will fail with a non-convergence error. If model does not converge try with nonlin=False |
| progressbar | boolean | default=False Determines whether or not a progress bar is displayed |

Multiple instruments for one target

As mentioned, while there must be at least one instrument for each target it is possible to have more than one variable in a given instrument. Moreover, it is possible to assign specific impulse defaults to different instruments, or different weights for different variables in a multi-variable instrument.

Below are specific illustrations of how the instrument list can be specified.

| Type | Instruments | Explanation |
|--|---|--|
| Single Target; Single Instrument | ['myvar'] | The instrument list includes only one variable, default impulse |
| 3 Targets; 3 Instruments, each with 1 variable | ['myvar1', 'myvar2', 'Myvar3'] | All variables get the default impulse |
| 2 Targets; 2 Instruments; different impulse | [('myvar1',0.7), ('myvar2',2000)] | The first instrument takes an impulse value of 0.7 (presumably because its values are relatively small). The second takes a much larger impulse value of 2000, reflecting its larger scale. |
| 2 Targets; 2 variables for first instrument; 1 for second | [['myvar1', 'myvar2'], 'myvar3')] | The first instrument takes two variables: myvar1 and myvar2; the second instrument has just one variable: myvar3 |
| 1 Target; 1 Instrument with 3 variables; different impulse | [[(‘myvar1’,50), (‘myvar2’,25), (‘myvar3’,10)]] | Three instrument variables, each is assigned an impulse/weight, such that in finding values to achieve the target, ‘myvar’ will be perturbed twice as much as myvar2 and 5 times as much as myvar3 |
| 1 Target; 1 Instruments with 3 variables; smaller impulse | [('myvar1',0.5), ('myvar2',0.25), ('myvar3',0.10)]] | Again three instrument variables, each is assigned an impulse/weight, such that in finding values to achieve the target, ‘myvar1’ will be perturbed twice as much as myvar2 and 5 times as much as MyVar3. NB this example will generate the same results as above, because although the impulse values have changed, the relative size of the impulse are the same |

Note

The final section of this chapter explains in more detail the solution algorithm of the `.invert()` method and the meaning of the various options of the method.

13.4.4 Solving for the instruments to reach the targets

Below is the actual call to invert used for this example.

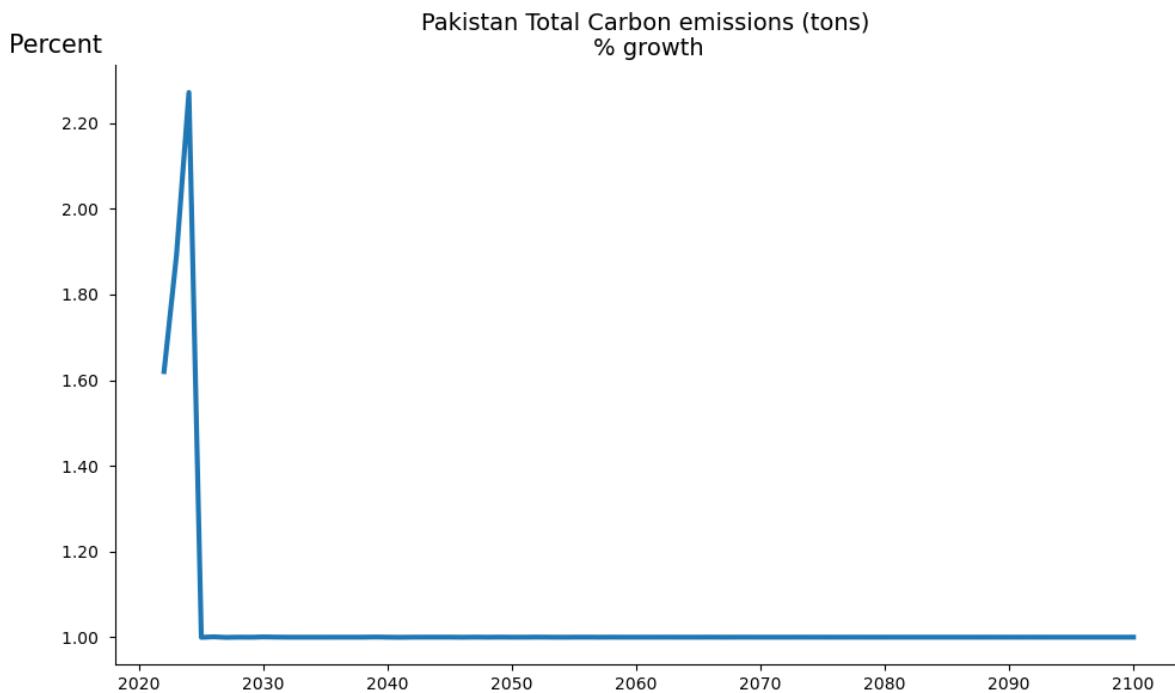
```
simple = mpak.invert(baseline,      # Invert calls the target instrument device
                     ↪
                     targets = target_simple,
                     instruments=instruments_simple,
                     #invert options
                     defaultimpuls=20,      # The default impulse instrument variables
                     defaultconv=2000.0,     # Convergence criteria for targets
                     varimpulse=True,        # Changes in instruments in each iteration
                     ↪
                     nonlin=3,               # are carried over to future iterations
                     ↪
                     silent=True,             # If no convergence after 3 iteration
                     ↪
                     recalculat=jacobian matrix
                     ↪
                     silent=False,            # Don't show iteration output
                     ↪
                     # (try False to show the results)
                     maxiter = 3000,
                     progressbar = True)
```

13.4.5 Display result

Once the simulation is complete the results are, as usual, stored in the mpak model object in the .lastdf dataframe. Results can be inspected either by using tables or graphically.

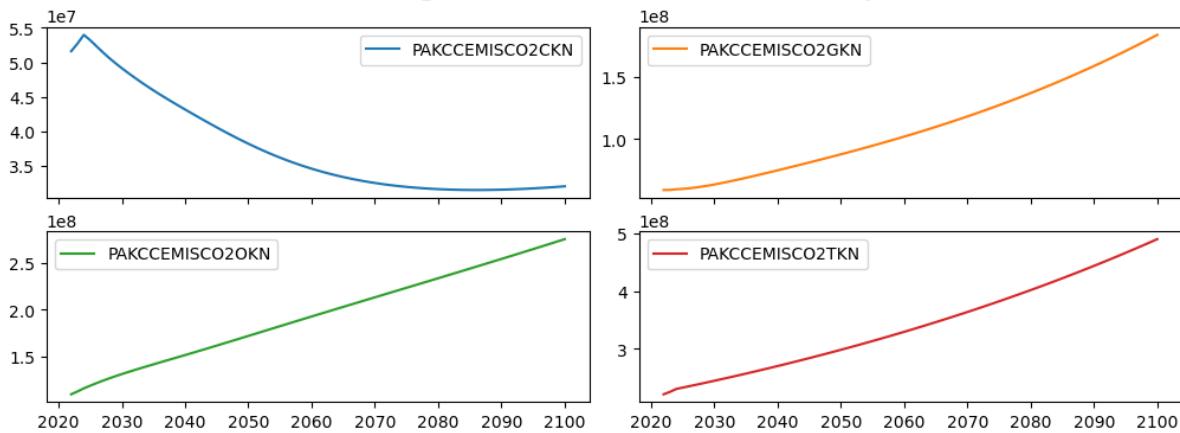
Below the .plot() method with option datatype='growth' is used to compare the total emissions and the carbon taxes from the solution set and the initial baseline, first in growth rates, then as a percent deviation from baseline option datatype='difpctlevel'. The option base_last=True ensures that the results from the most recent simulation (the lastdf DataFrame) are displayed along side those from the basedf DataFrame. If multiple series are specified each series will be displayed on a separate figure.

```
mpak.plot('PAKCCEMISCO2TKN',
          datatype='growth',
          legend=True,
          base_last=True).show
```



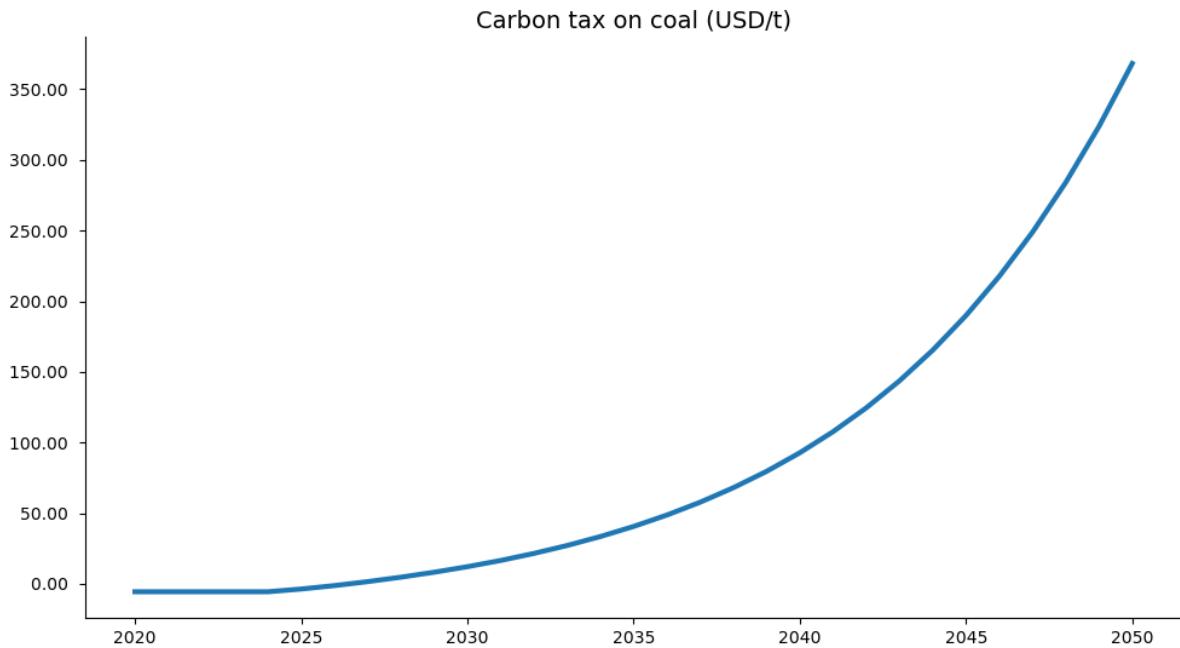
```
mpak['PAKCCEMISCO2?KN'].plot(
    title="Emissions: slow growth of emissions to 1% per annum",
    datatype='difpctlevel',
    showfig=True);
```

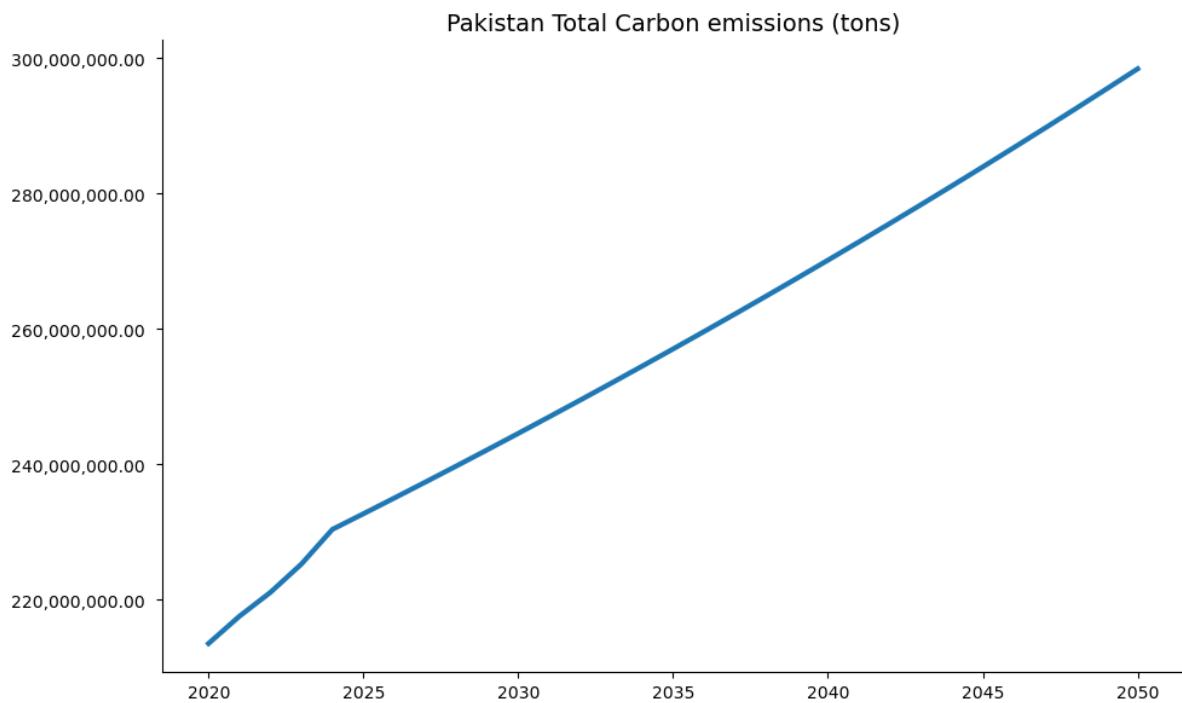
Emissions: slow growth of emissions to 1% per annum



```
with mpak.set_smpl(2020,2050):      # change if you want another timeframe
    fig1=mpak.plot('PAKCCEMISCO2TKN',
                    datatype='level',
                    legend=True,
                    base_last=True)
    fig2=mpak.plot('PAKGREVCO2CER',
                    datatype='level',
                    title=f'Pakistan, tax rate required to achieve slower emissions growth',
                    base_last=True,
                    legend=True)
    combo=(fig2+fig1)
    combo.show
```

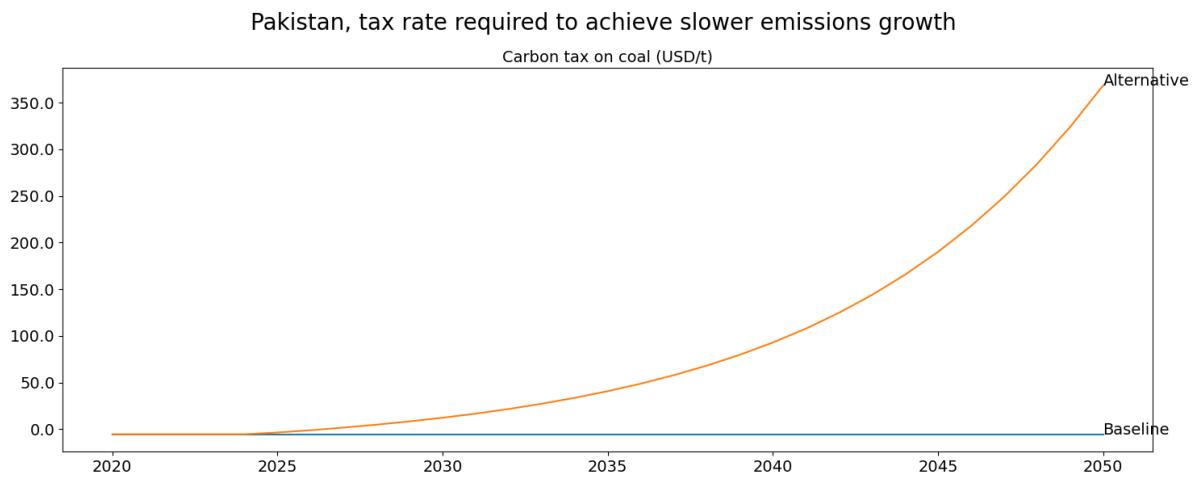
Pakistan, tax rate required to achieve slower emissions growth



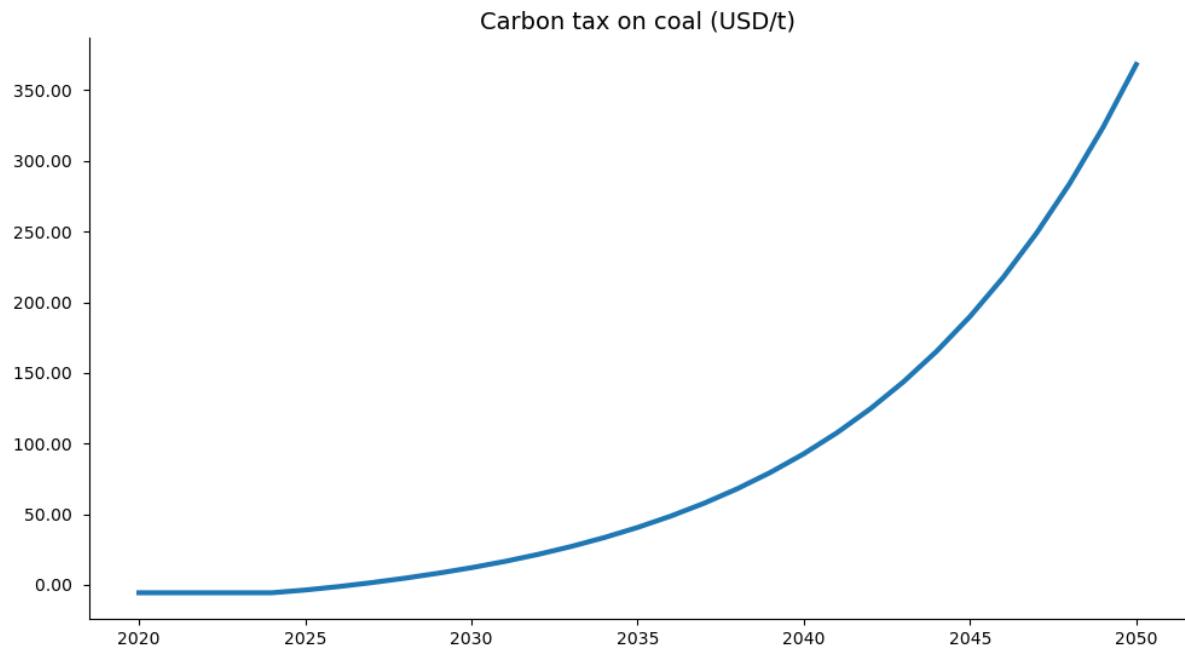


```
with mpak.set_smpl(2020,2050):      # change if you want another timeframe
    fig = mpak[f'PAKCCEMISCO2TKN'].plot_alt(title='Pakistan CO2 emission')
    fig2 = mpak[f'PAKGGREVCO2CER'].plot_alt(
        title=f'Pakistan, tax rate required to achieve slower emissions growth');

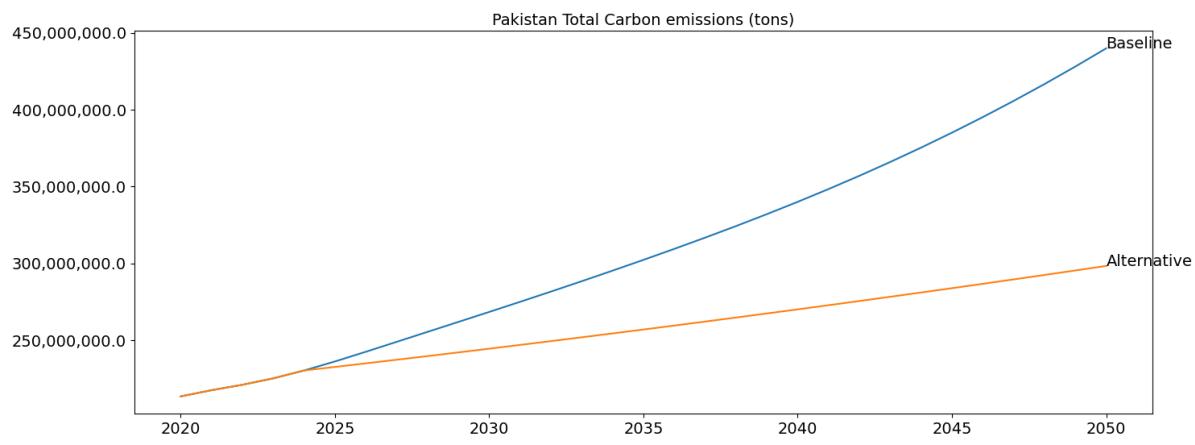
fig1
fig2
```



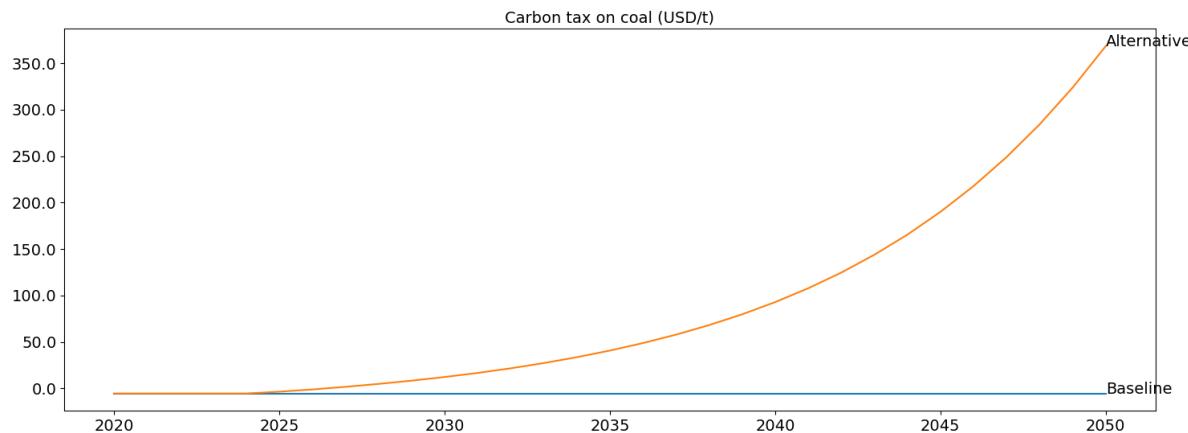
Pakistan, tax rate required to achieve slower emissions growth



Pakistan CO2 emission



Pakistan, tax rate required to achieve slower emissions growth



13.5 Targeting carbon emissions in a budget neutral manner

In this example, a more complex targeting exercise is conducted.

In this instance two targets are identified:

1. An unchanged fiscal deficit
2. A 40 percent decline in overall emissions

This requires at least two instruments:

1. The carbon emissions target will be determined by 3 instrument variables (the carbon taxes on each of coal, oil and natural gas)
2. The fiscal balance target will be met by one instrument Government spending on goods and services, implying that revenues from the Carbon Taxes will be used to increase government services.

Burns *et al.* (2021), using the same model explores, the macroeconomic consequences implications of alternative uses of the revenues from the Carbon Tax.

Note

there is no technical restriction on what instruments to choose, However, if instruments are chosen that have little influence on the targets, ModelFlow is unlikely to find values for the instruments that achieve the desired levels of the target variables.

13.5.1 Define target trajectory for CO2 emission.

The objective is to reduce Carbon emissions by 40% (as compared with the baseline) by the year 2050 and hold them constant in level terms afterwards.

The two variables `reduction_percent`, which reflects by how much emissions are to decline, and `achieved_by`, which represents by when the reduction should be achieved, are used to define the objectives series in a flexible way.

Using variables like this to express the constraint may be a bit more complicated. However, in the long run it may be easier as it allows the same code to be used to explore how different emission targets and different years in which the target should be fulfilled might affect the results.

```
reduction_percent = 40 # Input the desired reduction in percent.
achieved_by      = 2050
```

If the target is to achieve a 40 percent reduction in emissions by 2050, the pathway toward that objective can be described as a rate of growth of emissions that leads us to a 40 percent lower level by 2050.

That growth rate can be calculated by:

1. Calculating the level of emissions to be reached in the target year as $PAKCCEMISCO2TKN_{2050} \cdot (1 - 40/100)$
2. Calculate the growth rate of the target variable needed to reach that level in 2050= $\left(\frac{PAKCCEMISCO2TKN_{2050} \cdot (1 - 40/100)}{PAKCCEMISCO2TKN_{2024}} \right)^{\frac{1}{2050 - 2024}} - 1$

Once the target is defined the model can then calculate the values of carbon taxes necessary to reach those levels.

Below the target growth rate is calculated.

```
bau_emissions_final = baseline.loc[achieved_by, 'PAKCCEMISCO2TKN'] #baseline emissions
# in 2050
bau_emissions_2024 = baseline.loc[2024, 'PAKCCEMISCO2TKN'] #baseline emissions
# in 2024
target_emissions_final = bau_emissions_final*(1-reduction_percent/100) #target
# emissions in 2050
#growth rate needed between 2024 and 2050 to reach the target emissioons level in 2050
target_growth_rate = (target_emissions_final/bau_emissions_2024)**(1/(achieved_by-
>2024))-1
bau_growth_rate = (bau_emissions_final/bau_emissions_2024)**(1/(achieved_by-
>2024))-1
```

Below a quick routine to display the parameters and objectives.

```
print(f"Baseline Emissions in {achieved_by} : {bau_emissions_final:13,.0f} tons")
print(f"Target Emissions in {achieved_by} : {target_emissions_final:13,.0f} tons")
print(f"Business as usual growth rate in percent : {bau_growth_rate:13,.1%}")
print(f"Target growth rate in percent : {target_growth_rate:13,.1%}")
```

| | | |
|--|---|------------------|
| Baseline Emissions in 2050 | : | 439,930,561 tons |
| Target Emissions in 2050 | : | 263,958,337 tons |
| Business as usual growth rate in percent | : | 2.5% |
| Target growth rate in percent | : | 0.5% |

13.5.2 Create a dataframe with the target emissions

To prepare the simulation, a dataframe needs to be prepared with the target variable(s) set to the desired growth path, calculated above.

The target dataframe will contain as many variables as there are targets, at this stage just one.

Initially the target variable is set to the values of the original data in the baseline, then it is set to grow at the growth rate calculated above between 2024 and 2050 to achieve the 40 percent reduction in emissions and then it is held constant at this level.

```
# Create dataframe with only the target variable (data defined from 2024 onwards)
target_before = baseline.loc[2024:,[ 'PAKCCEMISCO2TKN']]
# create a target dataframe with a projection of the target variable
target = target_before.upd(f'<2025 {achieved_by}> PAKCCEMISCO2TKN =growth {100*target_-
>growth_rate}')
target = target.upd(f'<{min(2100,achieved_by+1)} {2100}> PAKCCEMISCO2TKN = {target_-
>emissions_final}')
#target.loc[:2055]
```

13.5.3 Create target for government deficit

In this example, there is a second target – to maintain the government deficit unchanged. As the objective is to hold the deficit constant as a share of GDP (at the levels in the baseline), the target for this variable will just take the same values as the government balance variable (expressed as a percent of GDP) PAKGGBALOVRLCN_ in the baseline.

```
#add to the target dataframe the GG balance variable from 2022 through 2100
target.loc[:, 'PAKGGBALOVRLCN_'] = baseline.loc[2022:2100, 'PAKGGBALOVRLCN_']
```

The target dataframe now holds two Series, defined over the period 2024 through 2100.

```
target
```

| | PAKCCEMISCO2TKN | PAKGGBALOVRLCN_ |
|-----------------------|-----------------|-----------------|
| 2024 | 2.303703e+08 | -3.131896 |
| 2025 | 2.315794e+08 | -3.045253 |
| 2026 | 2.327948e+08 | -2.984815 |
| 2027 | 2.340166e+08 | -2.945600 |
| 2028 | 2.352449e+08 | -2.921505 |
| ... | ... | ... |
| 2096 | 2.639583e+08 | -2.540508 |
| 2097 | 2.639583e+08 | -2.540567 |
| 2098 | 2.639583e+08 | -2.540636 |
| 2099 | 2.639583e+08 | -2.540713 |
| 2100 | 2.639583e+08 | -2.540798 |
| [77 rows x 2 columns] | | |

13.6 Define instruments

The instruments to achieve these targets are the **3 carbon taxes** and **government spending on goods and services**. To find the mnemonics for these variables a search is done over the descriptions of the variables, first over the carbon tax:

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

| | |
|-----------------|--|
| PAKCCEMISCO2TKN | : Pakistan Total Carbon emissions (tons) |
| PAKGREVC02CER | : Carbon tax on coal (USD/t) |
| PAKGREVC02GER | : Carbon tax on gas (USD/t) |
| PAKGREVC02OER | : Carbon tax on oil (USD/t) |

A separate search is done to identify the government spending variable to be used as an instrument. It makes sense to use a variable which has a fairly direct impact on the government deficit.

```
mpak['!*government*expenditure*goods*'].des
```

| | |
|----------------------|---|
| PAKGEXPGNFSCN | : General government expenditure on goods and services ↵(millions lcu) |
| PAKGEXPGNFSCN_A | : Add factor:General government expenditure on goods and ↵services (millions lcu) |
| PAKGEXPGNFSCN_D | : Fix dummy:General government expenditure on goods and ↵services (millions lcu) |
| PAKGEXPGNFSCN_FITTED | : Fitted value:General government expenditure on goods and ↵services (millions lcu) |
| PAKGEXPGNFSCN_X | : Fix value:General government expenditure on goods and ↵services (millions lcu) |

For the purposes of this simulation the PAKGGEXPGNFSCN_A variable (the add-factor for the government spending on goods and services equation) is selected. The add-factor is chosen so that the underlying equation remains active during the simulation.

Then a list called instruments is populated with two lists:

- the first is a list of variables for the first instrument (in this case the three carbon taxes)
- the second a list of one instruments for the second instrument (just one variable the add-factor on government spending on goods and services)

```
instruments = [['PAKGGREVCO2CER', 'PAKGGREVCO2GER', 'PAKGGREVCO2OER'],
               ['PAKGGEXPGNFSCN_A']]
```

13.7 Solve the two-target targeting problem:

Having defined the dataframes for the target values and the instrument variables, the model can be solved.

Note

The %%time command at the beginning of the cell below instructs Jupyter Notebook to keep track of how long it takes for the cell to execute and displays the result.

```
%%time
unweighted= mpak.invert(baseline,
                         targets = target.loc[:, :], #our targets defined above
                         instruments=instruments, # our instruments defined above
                         defaultimpuls=20,          # The default impulse value for the
                         ↪instruments
                         defaultconv=2000.0,         # Convergergence criteria for targets ( a
                         ↪relatively large number)
                         varimpulse=True,           # Change in instruments after each
                         ↪iteration are carried over to the future
                         nonlin=5,                  # If no convergence in 15 iteration
                         ↪recalculate jacobi
                         silent=True,                # Don't show iteration output
                         delay=False,
                         maxiter = 75,
                         progressbar = True)
```

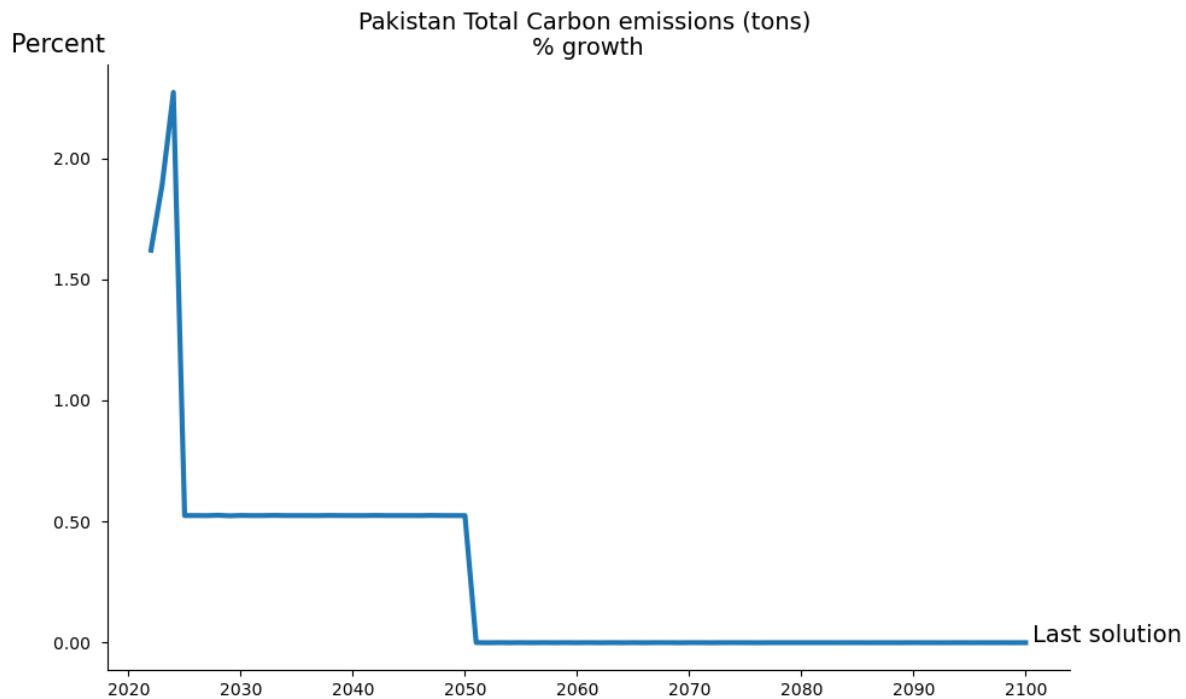
Finding instruments : 0% | 0/77

CPU times: total: 9.06 s
 Wall time: 9.13 s

13.7.1 Results

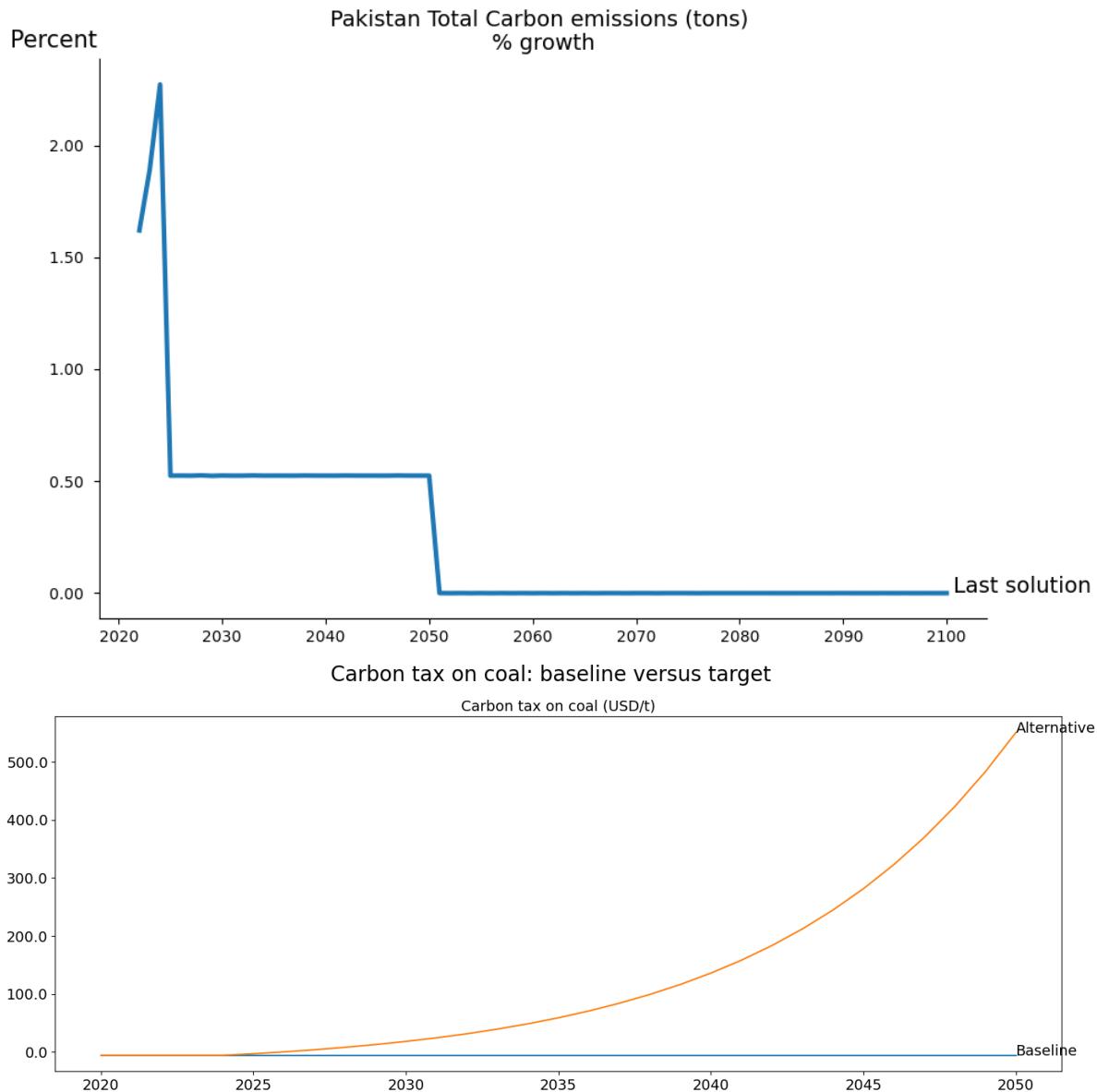
The following two chart illustrate that the objective of slowing emissions growth was achieved, with the growth rate in the targeted series equal to 1 percent.

```
mpak.plot('PAKCCEMISCO2TKN', base_last=True, legend=False).show
```



Below the carbon tax that was required to achieve the desired emissions result.

```
with mpak.set_smpl(2020, 2050):
    mpak['PAKGGREVCO2CER'].plot_alt(title='Carbon tax on coal: baseline versus target
→')
```



```
fig2.figs.keys()
```

```
dict_keys(['Pakistan Total Carbon emissions (tons), growth'])
```

13.8 Weighting the instruments

When using multiple instruments for a single target, the modeler may want to privilege changes in one instrument over another by specifying weights to attach to each. In the example below, specific weights are attached to the instruments, instructing the solver to place twice as much emphasis on adjusting the carbon tax on coal emissions (as compared with the other two carbon taxes). The actual number applied to the weights is not important as it is the relative weights that play a role. Thus here the weights 50,25,25 would have precisely the same effect as 2,1,1.

```

new_instruments =[['PAKGGREVCO2CER', 50],
                  ('PAKGGREVCO2GER', 25),
                  ('PAKGGREVCO2OER', 25)],
                  'PAKGGEEXPGNFSCN_A']

weighted = mpak.invert(baseline,      # Invert calls the target instrument device
                        ↪
                        targets = target.loc[:, :, ],
                        instruments=new_instruments,
                        defaultimpuls=20, # The default impulse instrument variables
                        defaultconv=2000.0, # Convergergence criteria for targets
                        varimpulse=True, # Changes in instruments in each iteration are
                        ↪carried over to the future
                        nonlin=5,        # If no convergence in 15 iteration recalculate jacobi
                        silent=True,     # Don't show iteration output
                        delay=0,
                        maxiter = 50,
                        progressbar = True)

```

Finding instruments : 0% | 0 / 77

With a weight twice as large on the coal carbon tax instrument, the carbon tax on coal rises to a level twice as fast as that of the other carbon taxes.

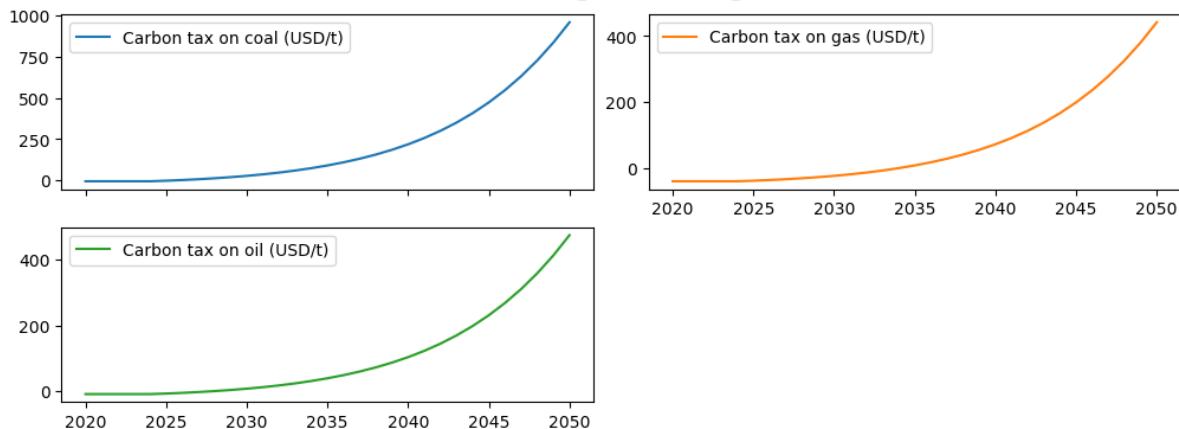
```

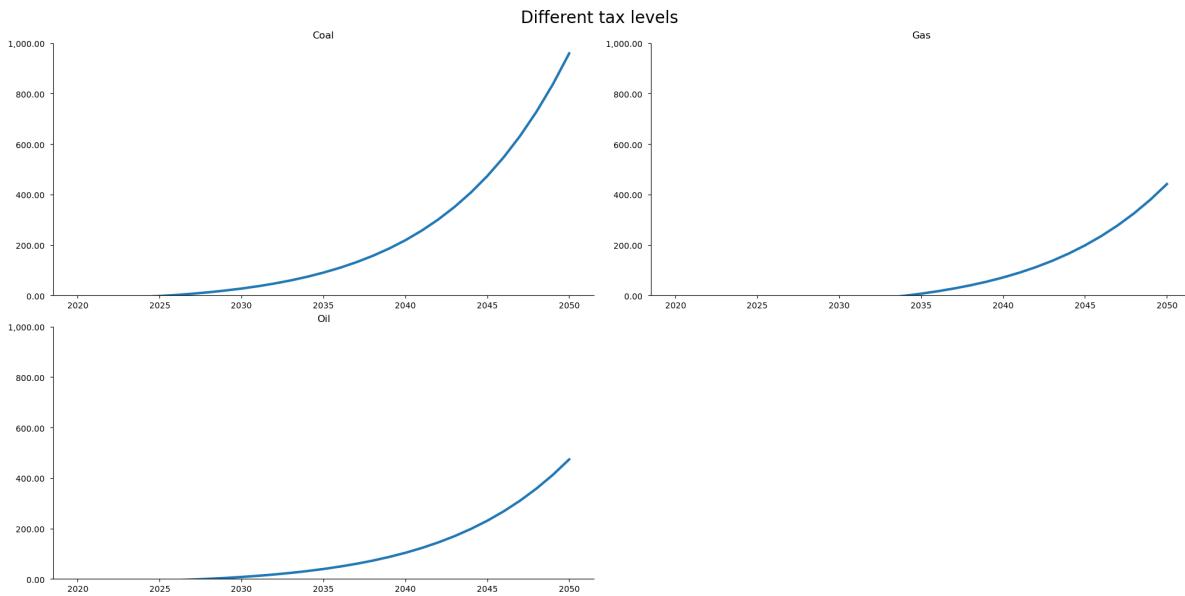
with mpak.set_smpl(2020,2050):    # change if you want another timeframe
    fig1 = mpak['PAKGGREVCO2CER PAKGGREVCO2GER PAKGGREVCO2OER'].rename().plot(
        title='Carbon taxes in weighted-target scenario')

with mpak.set_smpl(2020,2050):    # change if you want another timeframe
    fig = mpak.plot('PAKGGREVCO2CER PAKGGREVCO2GER PAKGGREVCO2OER',
                     datatype='level',base_last=True,name='fignewstyle',
                     mul=1,samefig=True,title='Different tax levels')
    fig.figs['fignewstyle'].axes[0].set_title('Coal')
    fig.figs['fignewstyle'].axes[1].set_title('Gas')
    fig.figs['fignewstyle'].axes[2].set_title('Oil')
    fig.figs['fignewstyle'].axes[0].set_ylim(0, 1000)
    fig.figs['fignewstyle'].axes[1].set_ylim(0, 1000)
    fig.figs['fignewstyle'].axes[2].set_ylim(0, 1000)
    fig.show

```

Carbon taxes in weighted-target scenario





Instead of putting different weights on the carbon taxes, an alternative might have been to add more instruments to the budget balance target (say direct and indirect taxes), with the weights equal to each tax type's share in total revenues. Set up this way, the scenario would maintain budget balance neutrality by using the revenues from the carbon taxes to reduce other (perhaps more distorting taxes).

Box 7. Targeting background

The concept of targets and instruments in economic modeling was introduced by Tinbergen (1967)

When solving a targeting problem it can be thought as follows:

Take a generic system of equations (a model): $\mathbf{y}_t = \mathbf{F}(\mathbf{x}_t)$

Where, \mathbf{x}_t are all predetermined variables - lagged endogenous and exogenous variables.

A condensed model (\mathbf{G}) can be defined comprised of a few endogenous variables ($\bar{\mathbf{y}}_t$) – the targets and a few a few exogenous variables($\bar{\mathbf{x}}_t$) – the instrument variables.

In this model, the remaining predetermined variables are fixed. Thus this model can be expressed as $\bar{\mathbf{y}}_t = \mathbf{G}(\bar{\mathbf{x}}_t)$.

In some models the result depends on the level of exogenous variables with a lag. For instance in a disease spreading model, the *number of infected* on a day depends on the *probability of transmission* some days before. If the *probability of transmission* is the instrument and the *number of infected* is the target. Therefor it can be useful to allow a **delay**, when finding the instruments. In this case we want to look at $\mathbf{y}_t = \mathbf{F}(\mathbf{x}_{t-delay})$

Inverting \mathbf{G} , gives a model where instruments are a functions of targets: $\bar{\mathbf{x}}_{t-delay} = \mathbf{G}^{-1}(\bar{\mathbf{y}}_t)$.

In other words, the inverted model is solved for the value of the instruments that gives the desired level for the targets: $\mathbf{G}^{-1}(\bar{\mathbf{y}}_t)$

For most models $\bar{\mathbf{x}}_{t-delay} = \mathbf{G}^{-1}(\bar{\mathbf{y}}_t)$ does not have a nicely closed-form solution. However it can be solved numerically – in ModelFlow this is done using the **Newton–Raphson** method.

So $\bar{\mathbf{x}}_{t-delay} = \mathbf{G}^{-1}(\bar{\mathbf{y}}_t^*)$ will be found using :

for $k = 1$ to convergence

$$\bar{\mathbf{x}}_{t-delay,end}^k = \bar{\mathbf{x}}_{t-delay,end}^{k-1} + \mathbf{J}_t^{-1} \times (\bar{\mathbf{y}}_t^* - \bar{\mathbf{y}}_t^{k-1})$$

$$\bar{\mathbf{y}}_t^k = \mathbf{G}(\bar{\mathbf{x}}_{t-delay}^k)$$

convergence: $|\bar{\mathbf{y}}_t^* - \bar{\mathbf{y}}_t| \leq \epsilon$

ModelFlow uses numerical differentiation, to find the Jacobian of the inverted matrix because it is simple and fast.

$$\mathbf{J}_t = \frac{\partial \mathbf{G}}{\partial \bar{\mathbf{x}}_{t-delay}}$$

$$\mathbf{J}_t \approx \frac{\Delta \mathbf{G}}{\Delta \bar{\mathbf{x}}_{t-delay}}$$

Mechanically that requires the model should be solved once for each instrument with a given delta applied to the targets. Recording the impact on each of the targets from the $\Delta x_{t-delay}^{instrument}$ gives and estimate of \mathbf{J}_t

In order for \mathbf{J}_t to be invertible there has to be **the same number of targets and instruments**.

However, each instrument can be a basket of exogenous variable an they can have different impulse Δ .

$$\Delta x^{instrument=i} = \begin{bmatrix} \Delta x^{instrument=i, variable=1} \\ \Delta x^{instrument=i, variable=2} \\ \Delta x^{instrument=i, variable=3} \\ \vdots \\ \Delta x^{instrument=i, variable=n} \end{bmatrix} \quad (13.1)$$

When an instrument changes the variables will change and the change will be in the proportions defined by their impulse.

Notice that the level of $\bar{\mathbf{x}}$ is updated (by $\mathbf{J}_t^{-1} \times (\bar{\mathbf{y}}_t^* - \bar{\mathbf{y}}_t^{k-1})$) in all periods from $t - delay$ to end , where end is the last timeframe in the dataframe. This is useful for many applications, where the instruments are level variable (i.e. not change variables).

This is the default behavior. It can be changed.

13.9 Tuning the target input to get a result

Models implemented in ModelFlow can be very different, and the targeting routine `.invert()` is fairly general. In many cases, targeting will not work out-of-the-box, its options will have to be tweaked to fit the problem at hand.

13.9.1 Targetting options

The invert options that affect the speed and accuracy of a solution are:

- `defaultimpuls`
- `defaultconv`
- `nonlin`
- `maxiter`
- `varimpulse`

13.9.2 defaultimpuls – set the size of the delta used when calculating the Jacobian

The impulse variable determines the size of the delta that is used to calculate the jacobian matrix. If it is too small or too large the resulting jacobian will solve only very slowly. Typically the impulse should be scaled in relation to the magnitude of the instrument it is to impact.

If a large impulse is used for a small variable (or a small impulse for a large variable) $\mathbf{x} + \Delta\bar{\mathbf{x}}_{t-delay}$ the model may become unsolvable.

Separate impulse values can be set for each instrument. This is done when setting the instruments (see discussion below).

13.9.3 nonlin – an integer - set to the number of iterations to attempt before recalculating the Jacobian

If the model is nonlinear it makes sense to re-estimate the jacobian matrix \mathbf{J}_t frequently. The `nonlin=a` number option allows the user to set the number of iterations the solver should allow without finding a solution before calculating a new jacobian.

If:

- `nonlin=0` the jacobian will not be updated (default) – implicitly indicates the model should be treated as if linear.
- `nonlin=<a number>` the same jacobi matrix will be updated after `<a number>` iterations. `-nonlin=3, 5` and `10` are all reasonable options in cases where model non-linearity requires the recalculation of he Jacobian.

13.9.4 Convergence

The targeting is stopped when all target variables converge. The convergence criteria should reflect the size of the target variables. Too large and the solution may not actually reflect a close approximation of the target, too small and the model may take a very long-time to solve.

- `defaultconv=<a number>`

13.9.5 Maximum number of iterations

- `maxiter=<a number>`

This option determines the maximum number of iterations that the model should run in trying to find a solution. Reasonable initial numbers may be between 50 and 100. If a model takes more than 100 iterations, there may be an issue. Potentially the chosen instruments do not have much impact on the target variables, or the model is relatively non-linear. Try setting `nonlin=10` to see if recalculating the Jacobian allows the model to solve.

13.10 Definition of Instruments

As noted above there must be at least one instrument for each target. Instruments are passed as a python list.

Each element in the list is an instrument.

- An element can be:
 - a variable name
 - a tuple with a variable name and an associated impulse Δ

- an inner list which defines which contains:
 - * a list of variable names. Each element in the inner list is an instrument variable
 - * a list of tuples each tuple contains a variable name and the associated impulse Δ .

The Δ variable(s) is (are) used in the numerical differentiation. Also if one instrument contains several variables, the proportion of each variable will be determined by the relative Δ variable.

CHAPTER
FOURTEEN

REPORT WRITING AND SCENARIO RESULTS

ModelFlow, is built on the back of and inherits the functionalities of standard pandas routines and other python libraries like [matplotlib](https://matplotlib.org/) (<https://matplotlib.org/>), [seaborn](https://seaborn.pydata.org/) (<https://seaborn.pydata.org/>), [plotly](https://plotly.com/) (<https://plotly.com/>), and [bokeh](https://bokeh.org/) (<https://bokeh.org/>). These libraries are well-integrated with DataFrames, and offer a wide-range of visualization capabilities. As has been done throughout this manual, they can be used to look at data, and compare simulation results among other things.

In this chapter - Report Writing

This chapter focuses on tools and techniques for examining the data in a model and the results of simulations. Many of these techniques have been illustrated elsewhere, but are brought together in one place in this chapter to facilitate retrieval later. The chapter includes a discussion of techniques for creating comprehensive reports based on simulations and analyzes performed in ModelFlow.

Examples in the chapter:

- Demonstrate how to generate structured tables, charts, and text outputs directly from simulation results.
- Introduces ModelFlow's reporting classes (`.table()`, `.plot()`, `.text()`) and how they can be used to produce reports that combine both tabular, graphical and textual results.
- Issues dealt with include how to generate tables that display only selected time periods
- How to present data in landscape and portrait form
- Presenting graphs either on their own or in groups
- using the `.reports` method to store a set of tables and or graphs as a template that then can be applied to different scenario results yielding standardized reports

Outputs from all of these methods can be rendered as interactive widgets, html, pdf or various bitmap forms.

ModelFlow users are free to employ any Python-based library for visualization purposes. For specific tasks, ModelFlow also includes some specialized procedures derived from the above packages that may be of interest. These routines are tailored to utilize ModelFlow internal data structures, like the `.lastdf`, `.basedf` and `keep` DataFrames as well as metadata in the model object such as variable descriptions. Moreover, ModelFlow reporting routines include specific transformations (like growth rates) and scenario comparison routines that are useful in the analysis of macroeconomic model results.

The following box summarizes the four different kinds of report-writing objects incorporated into the ModelFlow package.

Box 8. ModelFlow report writing routines

ModelFlow augments standard Python routines with four report-writing classes:

1. **table**: a class that represents data from the `.lastdf` and `.basedf` in tabular form.
2. **plot**: a class that represents data from the `.lastdf` and `.basedf` and kept dataframes in graphical form.
3. **text**: a class for text-based tables, which can be specified using plain text, LaTeX, or HTML, or any combination of the three.
4. **report**: a container class that can be comprised of an arbitrary number of tables and plots in any order.

14.1 Preparing a ModelFlow Python environment

To begin, a solution file that was saved at the end of the previous chapter is loaded, using the by now familiar `modelload` method. Because the simulations performed in the previous chapter used the `keep` option, and, because the `mpakwScenarios.pcim` file was saved with the `keep` option set to true, the `modelload` method gives the current session access to all the results generated in the previous chapter.

```
mpak, _ = model.modelload(r'..\models\mpakw.pcim', run=1)
_ = mpak.smpl(2025, 2029)
```

Zipped file read: ..\models\mpakw.pcim

Box 9. Where ModelFlow Stores Results

When a model is solved (simulated), the result is returned as a `DataFrame`.

To facilitate reporting, the resulting `DataFrame` is also stored as the `.lastdf` property of the model object. The `.lastdf` property is overwritten every time the model is solved. To preserve the results for future reference, the `keep='Some Solution Name'` option can be used. This will store a copy of the `DataFrame` in a dictionary called `keep_solutions` where the key will be the text descriptor given in the `keep` option – in this case ‘Some Solution Name’.

```
result = mpak(dataframe_with_experiment, keep='Some Solution Name')
```

The `.basedf` `DataFrame` contains the baseline values. It is set during the first simulation in a session, either when a model is loaded with `run=True` or when the model is simulated for the first time. But it can also be reset manually if desired `mpak.basedf=mydataframe`.

If a model has a lot of variables and there are a lot of scenarios it can be useful to limit the number of variables in the stored `DataFrame`. This is done by specifying: `keep_variables=<a string with list of variables including wildcards>`

```
result = mpak(dataframe_with_experiment, keep='some text', keep_variables = '*NYGDPMKTPKN
*NECONPRVTKN')
```

Will only keep variables matching the wildcard expressions: `*NYGDPMKTPKN *NECONPRVTKN`

To reset `.keep_solutions` to an empty dictionary type: `{modelobject}.keep_solutions = {}`

Below, in order to have meaningful data for the following report-writing examples, the `.basedf` and `.lastdf`

DataFrames of mpak are pre-populated with the “Baseline” and “1% of GDP increase in FDI and private investment (AF shock)” results from the kept scenarios of the previous chapter.

```
mpak.basedf = mpak.keep_solutions['Baseline']
mpak.lastdf = mpak.keep_solutions['1% of GDP increase in FDI and private investment_
↪(AF shock)']
mpak.smpl(2025,2029);
```

Recall

The .keep_solutions dictionary can be interrogated to list all of the solutions (the keys) that were performed and stored in the previous chapter and the dataframes that were generated when the simulations were performed (the values) in the dictionary.

14.2 The .table() class

The .table() method is a constructor for the ModelFlow class DisplayVarTableDef. When called, it creates a table object from the data series passed to it. By default, it represents the data as growth rates and draws them from the .lastdf dataframe unless .basedf is requested specifically.

If a comparison display option is chosen (see below), the comparison will be between the values for the selected variables from the .lastdf and the .basedf DataFrames.

14.2.1 Create a .table() object

The following generates a simple table object:

```
tab = mpak.tab(name='My_first_table', pat=' *NYGDPMKTPKN *NECONPRVTKN *NEGDIFTOTKN_
↪*NEEXPGNFSKN *NEIMPGNFSKN', title='GDP components', foot='Source: World Bank ')
```

Arguments used:

- **pattern** (pat) specifies the variables to be displayed. Here the * wildcard is used, allowing this pattern to be used for models for any World Bank model that conforms to the Bank’s standard naming conventions
- **name** is an internal identifier used to identify output from this table object when different tables are producing output.
- **title** specifies text to be used as a title for the Table
- **foot** specifies text to be placed in a footer for the table

As written the command creates an object called tab. tab can be called subsequently to display or manipulate the table in different ways.

Below the same table is generated using a variable to represent the pattern of variables to be included in the table.

```
pat= ' *NYGDPMKTPKN *NECONPRVTKN *NEGDIFTOTKN *NEEXPGNFSKN *NEIMPGNFSKN '
tab = mpak.table(name='My_first_table', pat=pat, title='GDP components', foot='Source:_
↪World Bank ')
```

14.2.2 Rendering table objects

Table objects can be rendered as html, text and pdf objects. The below table indicates the methods associated with each.

Display options for table objects

| command | output |
|-----------------------------|--|
| None (just the object name) | In Jupyter Notebook displays the table in its html format. |
| display(tab) | Displays the table in html format. |
| .show | Returns a simple text version of the table. |
| .pdf | Returns a nicely formatted table in pdf format. |

.tab() example output

When the table object in the last line of a cell in Jupyter Notebooks it will cause the content of the object to be rendered in html.

```
tab
```

| | 2025 | 2026 | 2027 | 2028 | 2029 |
|------------------------|------|------|------|-------|------|
| --- Percent growth --- | | | | | |
| Real GDP | 2.08 | 2.42 | 2.64 | 4.36 | 2.91 |
| HH. Cons Real | 2.03 | 2.32 | 2.48 | 3.45 | 2.62 |
| Investment real | 1.10 | 1.46 | 1.82 | 12.46 | 2.98 |
| Exports real | 4.37 | 4.20 | 4.04 | 3.88 | 3.72 |
| Imports real | 3.10 | 3.10 | 3.02 | 4.68 | 2.84 |

Source: World Bank

The display(tab) method produces exactly the same output

And it can be used in any line in a jupyter cell

The Table .show method

The .show() method renders the table in pure text form, without any html or pdf formatting.

```
tab.show
```

| GDP components | 2025 | 2026 | 2027 | 2028 | 2029 |
|-----------------|------------------------|------|------|-------|------|
| | --- Percent growth --- | | | | |
| Real GDP | 2.08 | 2.42 | 2.64 | 4.36 | 2.91 |
| HH. Cons Real | 2.03 | 2.32 | 2.48 | 3.45 | 2.62 |
| Investment real | 1.10 | 1.46 | 1.82 | 12.46 | 2.98 |
| Exports real | 4.37 | 4.20 | 4.04 | 3.88 | 3.72 |
| Imports real | 3.10 | 3.10 | 3.02 | 4.68 | 2.84 |

Source: World Bank

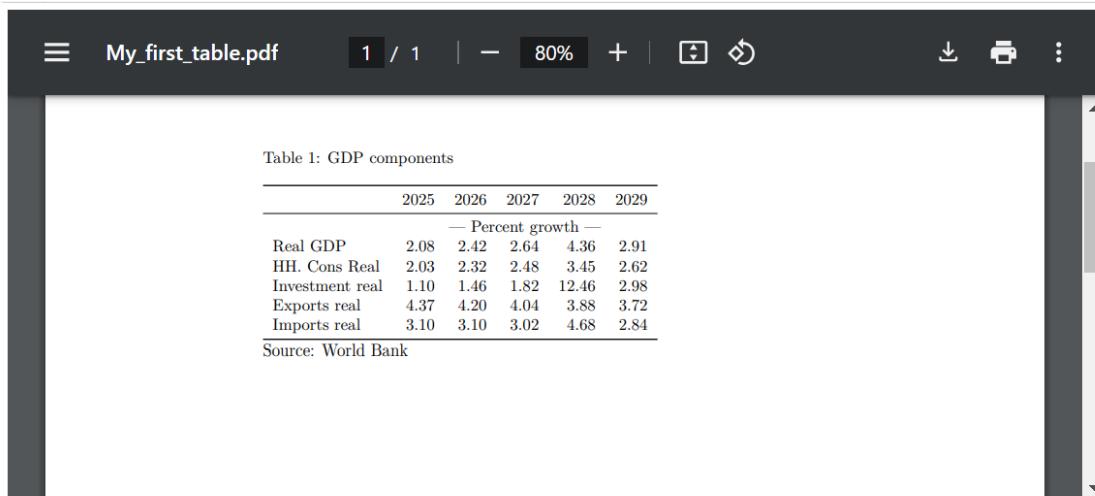
The .pdf method causes the table to be rendered to a pdf using latex.

In Jupyter Notebook the rendered file will be displayed within the Jupyter Notebook. Alternatively if the .pdf method is called with the option (pdfopen=True) then the pdf will be opened in a separate browser. In either case, if the table is too large to appear on the pdf page, the scrollbar of the pdf viewer can be used to view the extra columns or rows.

```
tab.pdf()
```

⚠ Warning

PDF functionality does not work in Google Colab out of the box. Google Colab does not include an installation of latex by default. As a result, when running the PDF routines on Colab they will not render.



Box 10. PDFs and latex outputs

PDF files are generated using Latex as an intermediary. As a result, Latex must be installed on the machine for this option to work. For this publication the `Miktex` package has been used, and routines have also been tested with the `Tex Live` package. More on installing the `Miktex` package can be found here: <https://miktex.org/download>. When installing Miktex allow automatic download of latex packages.

On a Mac, MacTex can be used. It can be found here: <https://www.tug.org/mactex/mactex-download.html>

ModelFlow writes the intermediate latex commands from which the pdf is generated to a sub-directory entitled `latex` one level below the current work directory. The filename will be set to the `table_name` parameter specified when the `Table` object was declared. Thus, the table called “`myTable`” will be saved to `latex/myTable/MyTable.pdf`. The intermediate latex source code is placed in `latex/myTable/myTable.tex`.

In case of errors or if the user want to enhance the output this file can be edited manually and the pdf re-generated manually using TexWorks which is part of the Miktex instalation.

14.2.3 Table options

Data transformations - Growth Rates, Levels, etc.

When the `Table` object is invoked, it defaults to displaying the growth rates for specified variables. However, the constructor supports a range of transformations. Which specification is used is determined by the `datatype` argument, which can take the following values:

Table: Data Types and Their Meanings

| datatype | Meaning |
|---|--|
| <code>growth</code> | This is the default setting. Growth rate in percent in the most recent dataset (<code>.lastdf</code>). |
| <code>level</code> | Values in <code>.lastdf</code> . |
| <code>gdppct</code> | Percentage of GDP in <code>.lastdf</code> . |
| <code>qoq_ar</code> | Quarterly growth annualized. - Quarterly models only. |
| Difference views | |
| <code>difgrowth</code> | Change (Δ) in growth rates (<code>.lastdf</code> less <code>basedf</code>). |
| <code>diflevel</code> | Change (Δ) in values (<code>.lastdf</code> less <code>basedf</code>). |
| <code>difgdppct</code> | Change (Δ) in the percentage of GDP from (<code>.lastdf</code> less <code>basedf</code>). |
| <code>difqoq_ar</code> | change in the annualized quarterly growth (<code>.lastdf</code> less <code>basedf</code>) |
| <code>difpctlevel</code> | Percentage change (Δ) in values (<code>.lastdf</code> less <code>basedf</code>). |
| Values from <code>.basedf</code> | |
| <code>basegrowth</code> | Growth rate in percent in <code>.basedf</code> . |
| <code>baselevel</code> | Level of the data in <code>.basedf</code> |
| <code>basegdppct</code> | Percentage of GDP in <code>.basedf</code> . |
| <code>baseqoq_ar</code> | Quarterly growth in <code>.basedf</code> |

transpose option

When set, `transpose=True` (default is False), the table will be rendered so that series appear as columns (with dates progressing downward on the page). In this mode, many time-periods can be displayed, but the number of series will be limited by the width of the paper/screen.

```
with mpak.set_smpl(2022, 2027):
    tab_t = mpak.table(name='A_transposed_table', pat=pat, title='GDP components',
                        foot='Source: World Bank ', transpose = True)
tab_t.show
```

| GDP components | | | | | |
|----------------|------------------------|----------|-----------------|--------------|--------------|
| | Real GDP | HH. Cons | Real Investment | Exports real | Imports real |
| | --- Percent growth --- | | | | |
| 2022 | 0.66 | 0.80 | 0.70 | 4.71 | 3.00 |
| 2023 | 1.04 | 1.09 | 0.63 | 4.66 | 2.86 |
| 2024 | 1.60 | 1.60 | 0.79 | 4.53 | 2.99 |
| 2025 | 2.08 | 2.03 | 1.10 | 4.37 | 3.10 |
| 2026 | 2.42 | 2.32 | 1.46 | 4.20 | 3.10 |
| 2027 | 2.64 | 2.48 | 1.82 | 4.04 | 3.02 |

Source: World Bank

The dec## option: limits the decimals displayed

In the example below, only one decimal of the generated table is displayed. Note this table uses the datatype option "pctgap", which displays the results as a percent of GDP.

⚠ Warning

The option `datatype='gdppct'` is only well-defined for variables that follow World Bank naming conventions and end either (CD,CN, KD or KN). The `gdppct` routine uses these endings to determine whether to calculate the percent of GDP as a percent of nominal GDP either in local currency (CN) or USD (CD) or real GDP in local currency (KN) or real USD (KD).

```
pat_gov = '*GGBALOVRLCN *GGDBTTOTLCN *BNCABFUNDCD'
tab_gov = mpak.table(pat_gov,datatype='gdppct',title='In percent of GDP ',dec=1)
tab_gov.show
```

| In percent of GDP | | | | | |
|---|------------------------|------|------|------|------|
| | 2025 | 2026 | 2027 | 2028 | 2029 |
| | --- Percent of GDP --- | | | | |
| General Government Revenue, Deficit, LCU mn | -3.0 | -3.0 | -2.9 | -2.8 | -2.8 |
| General government gross debt millions lcu | 57.0 | 57.3 | 57.7 | 57.1 | 57.5 |
| Current Account Balance, US\$ mn | -3.6 | -3.5 | -3.4 | -3.5 | -3.4 |

14.2.4 Handling Table Overruns (too much data)

It is possible to specify a table that is too large for conventional display systems. For example, a table of data taken from a model that covers 100 years could, if the dates run vertically, span more than 1 page, or one that covers many data series might overrun the right hand side of a computer screen, pdf file or printed page. Displaying such tables in Jupyter Notebooks or other computer-based displays is straightforward because Jupyter Notebook will allow the user to scroll content that does not fit on the page. However, in LaTeX documents or printed materials, tables that overrun the page can become impossible to read.

To address this issue, several strategies can be employed:

- **Drop Middle Columns:** This involves removing several columns from the middle of the table that may not be critical to the reader's understanding, helping to condense the table into a more manageable size.
- **Display Tables in Chunks:** Specify a rule for breaking the table into multiple smaller tables, each covering a shorter time span. This not only makes each segment easier to fit on a page but also can help readers digest the information in smaller, more focused increments.
- **Display a Slice of Data:** Rather than showing the entire timeline, you can display a specific segment or 'slice' of the data that is most relevant to the analysis at hand. This approach focuses the reader's attention and avoids overwhelming them with too much information at once.

These methods can significantly enhance the readability and presentation of large tables when rendered on media that are less flexible than interactive digital platforms.

Default behavior for dealing with large tables

Below a long table is generated, which runs beyond the edges of the monitor on most computer displays. In Jupyter Notebook or other IDEs elements the standard display options [table name or display(tablename) – both of which produce html] will generate outputs that that cannot render within one screen width but can scrolled to.

```
with mpak.set_smpl(2026,2050):
    tab_large = mpak.table(pat,title='GDP components',name='large_table',dec=1)
```

```
display(tab_large)
```

| GDP components | | 2026 | 2027 | 2028 | 2029 | 2030 | 2031 | 2032 | 2033 | 2034 | 2035 | 2036 | 2037 | 2038 | 2039 | 2040 | 2041 | 2042 | 2043 | 2044 | 2045 | 2046 | 2047 | 2048 | 2049 | 2050 |
|------------------------|--|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|
| --- Percent growth --- | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Real GDP | | 2.4 | 2.6 | 4.4 | 2.0 | 2.8 | 2.8 | 3.0 | 3.2 | 3.3 | 3.4 | 3.4 | 3.4 | 3.4 | 3.4 | 3.4 | 3.4 | 3.4 | 3.4 | 3.4 | 3.4 | 3.3 | 3.3 | 3.3 | 3.3 | |
| HH. Cons Real | | 2.3 | 2.5 | 3.4 | 2.6 | 2.5 | 2.6 | 2.8 | 2.9 | 3.0 | 3.1 | 3.2 | 3.2 | 3.2 | 3.2 | 3.2 | 3.2 | 3.2 | 3.2 | 3.2 | 3.2 | 3.2 | 3.2 | 3.2 | 3.2 | 3.2 |
| Investment real | | 1.5 | 1.8 | 12.5 | 3.0 | 2.9 | 3.0 | 3.2 | 3.4 | 3.6 | 3.8 | 4.0 | 4.1 | 4.2 | 4.2 | 4.3 | 4.3 | 4.4 | 4.4 | 4.3 | 4.3 | 4.3 | 4.3 | 4.3 | 4.3 | 4.3 |
| Exports real | | 4.2 | 4.0 | 3.9 | 3.7 | 3.6 | 3.5 | 3.4 | 3.3 | 3.3 | 3.2 | 3.1 | 3.1 | 3.0 | 3.0 | 2.9 | 2.9 | 2.8 | 2.8 | 2.7 | 2.7 | 2.7 | 2.7 | 2.7 | 2.7 | 2.6 |
| Imports real | | 3.1 | 3.0 | 4.7 | 2.8 | 2.5 | 2.5 | 2.6 | 2.6 | 2.6 | 2.6 | 2.6 | 2.6 | 2.6 | 2.6 | 2.6 | 2.6 | 2.7 | 2.7 | 2.7 | 2.8 | 2.8 | 2.9 | 2.9 | 2.9 | 2.9 |

The .show and .pdf methods automatically truncates the width of the table to fit on screen. By default it will show the beginning and ending data columns leaving out the intermediate columns.

```
tab_large.show
```

| GDP components | | 2026 | 2027 | 2028 | ... | 2048 | 2049 | 2050 |
|------------------------|--|------|------|------|-----|------|------|------|
| --- Percent growth --- | | | | | | | | |
| Real GDP | | 2.4 | 2.6 | 4.4 | ... | 3.3 | 3.3 | 3.3 |
| HH. Cons Real | | 2.3 | 2.5 | 3.4 | ... | 3.2 | 3.2 | 3.2 |
| Investment real | | 1.5 | 1.8 | 12.5 | ... | 4.3 | 4.3 | 4.3 |
| Exports real | | 4.2 | 4.0 | 3.9 | ... | 2.7 | 2.7 | 2.7 |
| Imports real | | 3.1 | 3.0 | 4.7 | ... | 2.8 | 2.8 | 2.9 |

```
tab_large.pdf()
```

Table 1: GDP components

| | 2026 | 2027 | 2028 | 2048 | 2049 | 2050 |
|--------------------|------|------|------|------|------|------|
| — Percent growth — | | | | | | |
| Real GDP | 2.4 | 2.6 | 4.4 | 3.3 | 3.3 | 3.3 |
| HH. Cons Real | 2.3 | 2.5 | 3.4 | 3.1 | 3.2 | 3.2 |
| Investment real | 1.5 | 1.8 | 12.5 | 4.3 | 4.3 | 4.3 |
| Exports real | 4.2 | 4.0 | 3.9 | 2.7 | 2.7 | 2.6 |
| Imports real | 3.1 | 3.0 | 4.7 | 2.8 | 2.8 | 2.8 |

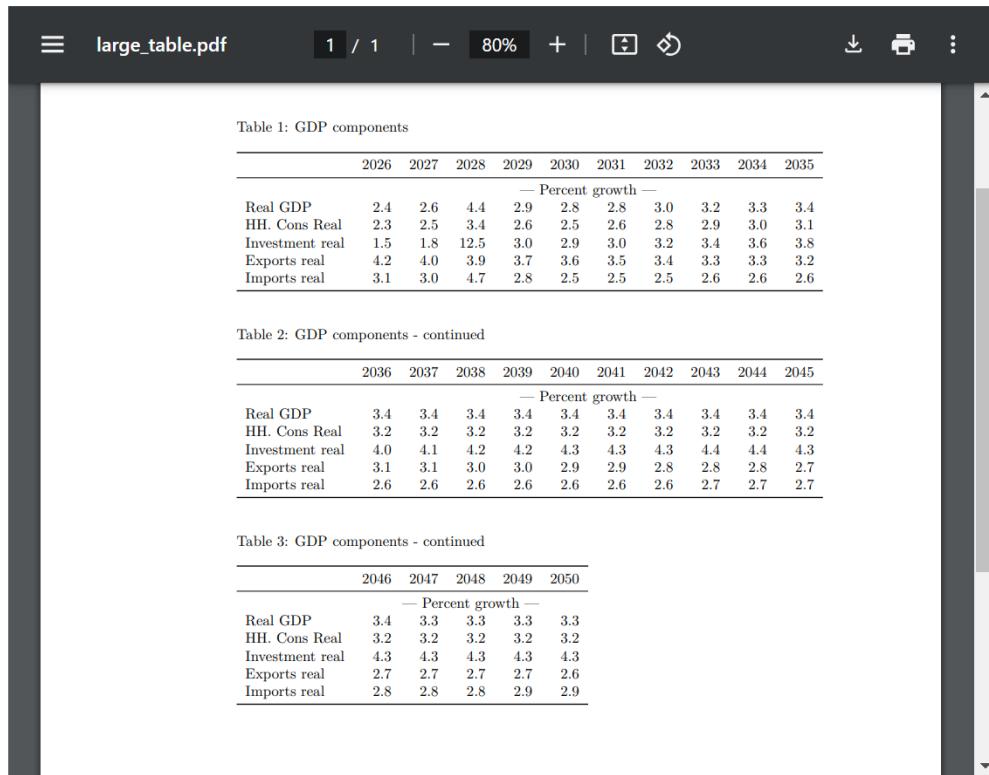
Customizing the display of “too large” tables

These default behaviors can be modified using various `.table` options (see table), notably the options (`chunk_size=`, `time-slice=`, `transpose=`, `max_cols=`, `last_cols=`) which determine how excess data is treated.

`chunk_size=` to split pdf tables.

If the option `chunk_size` is set. The pdf table will be split into sub tables each with `chunk_size` columns (except, perhaps, the final table).

```
tab_large.set_options(chunk_size=10).pdf(height = '700px')
```



large_table.pdf

Table 1: GDP components

| | 2026 | 2027 | 2028 | 2029 | 2030 | 2031 | 2032 | 2033 | 2034 | 2035 |
|-----------------|--------------------|------|------|------|------|------|------|------|------|------|
| | — Percent growth — | | | | | | | | | |
| Real GDP | 2.4 | 2.6 | 4.4 | 2.9 | 2.8 | 2.8 | 3.0 | 3.2 | 3.2 | 3.3 |
| HH. Cons Real | 2.3 | 2.5 | 3.4 | 2.6 | 2.5 | 2.6 | 2.8 | 2.9 | 3.0 | 3.1 |
| Investment real | 1.5 | 1.8 | 12.5 | 3.0 | 2.9 | 3.0 | 3.2 | 3.4 | 3.6 | 3.8 |
| Exports real | 4.2 | 4.0 | 3.9 | 3.7 | 3.6 | 3.5 | 3.4 | 3.3 | 3.3 | 3.2 |
| Imports real | 3.1 | 3.0 | 4.7 | 2.8 | 2.5 | 2.5 | 2.5 | 2.6 | 2.6 | 2.6 |

Table 2: GDP components - continued

| | 2036 | 2037 | 2038 | 2039 | 2040 | 2041 | 2042 | 2043 | 2044 | 2045 |
|-----------------|--------------------|------|------|------|------|------|------|------|------|------|
| | — Percent growth — | | | | | | | | | |
| Real GDP | 3.4 | 3.4 | 3.4 | 3.4 | 3.4 | 3.4 | 3.4 | 3.4 | 3.4 | 3.4 |
| HH. Cons Real | 3.2 | 3.2 | 3.2 | 3.2 | 3.2 | 3.2 | 3.2 | 3.2 | 3.2 | 3.2 |
| Investment real | 4.0 | 4.1 | 4.2 | 4.2 | 4.3 | 4.3 | 4.3 | 4.4 | 4.4 | 4.3 |
| Exports real | 3.1 | 3.1 | 3.0 | 3.0 | 2.9 | 2.9 | 2.8 | 2.8 | 2.8 | 2.7 |
| Imports real | 2.6 | 2.6 | 2.6 | 2.6 | 2.6 | 2.6 | 2.6 | 2.7 | 2.7 | 2.7 |

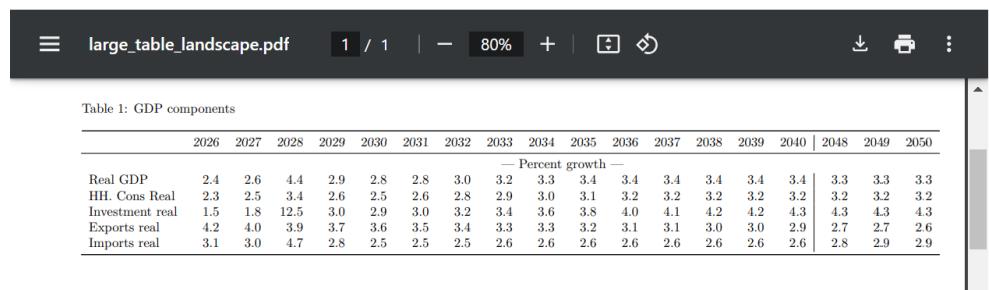
Table 3: GDP components - continued

| | 2046 | 2047 | 2048 | 2049 | 2050 |
|-----------------|--------------------|------|------|------|------|
| | — Percent growth — | | | | |
| Real GDP | 3.4 | 3.3 | 3.3 | 3.3 | 3.3 |
| HH. Cons Real | 3.2 | 3.2 | 3.2 | 3.2 | 3.2 |
| Investment real | 4.3 | 4.3 | 4.3 | 4.3 | 4.3 |
| Exports real | 2.7 | 2.7 | 2.7 | 2.7 | 2.6 |
| Imports real | 2.8 | 2.8 | 2.8 | 2.9 | 2.9 |

`landscape=` to show a pdf in landscape mode. .

If the option `landscape=` is set. The pdf table will be shown in landscape mode. In this case the `max_cols` option should also be used, as the default will only be 6.

```
tab_large.set_options(landscape=True, max_cols=18, name='large_table_landscape').pdf()
```



large_table_landscape.pdf

Table 1: GDP components

| | 2026 | 2027 | 2028 | 2029 | 2030 | 2031 | 2032 | 2033 | 2034 | 2035 | 2036 | 2037 | 2038 | 2039 | 2040 | 2048 | 2049 | 2050 |
|-----------------|--------------------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|
| | — Percent growth — | | | | | | | | | | | | | | | | | |
| Real GDP | 2.4 | 2.6 | 4.4 | 2.9 | 2.8 | 2.8 | 3.0 | 3.2 | 3.3 | 3.4 | 3.4 | 3.4 | 3.4 | 3.4 | 3.4 | 3.3 | 3.3 | 3.3 |
| HH. Cons Real | 2.3 | 2.5 | 3.4 | 2.6 | 2.5 | 2.6 | 2.8 | 2.9 | 3.0 | 3.1 | 3.2 | 3.2 | 3.2 | 3.2 | 3.2 | 3.2 | 3.2 | 3.2 |
| Investment real | 1.5 | 1.8 | 12.5 | 3.0 | 2.9 | 3.0 | 3.2 | 3.4 | 3.6 | 3.8 | 4.0 | 4.1 | 4.2 | 4.2 | 4.3 | 4.3 | 4.3 | 4.3 |
| Exports real | 4.2 | 4.0 | 3.9 | 3.7 | 3.6 | 3.5 | 3.4 | 3.3 | 3.3 | 3.2 | 3.1 | 3.0 | 3.0 | 2.9 | 2.7 | 2.7 | 2.6 | 2.6 |
| Imports real | 3.1 | 3.0 | 4.7 | 2.8 | 2.5 | 2.5 | 2.6 | 2.6 | 2.6 | 2.6 | 2.6 | 2.6 | 2.6 | 2.6 | 2.6 | 2.8 | 2.9 | 2.9 |

The `timeslice =` option allows the user to select which years to render

The `timeslice` option gives the user full control over which columns (years or quarters) will be displayed.

Beware, `timeslice` doesn't care if years are out of order.

In the example below the years are out of order, and the resulting output reflects what was asked for.

```
tab_large.set_options(timeslice=[2026, 2040, 2028 , 2029, 2050]).show
```

| GDP components | 2026 | 2040 | 2028 | 2029 | 2050 |
|------------------------|------|------|------|------|------|
| --- Percent growth --- | | | | | |
| Real GDP | 2.4 | 3.4 | 4.4 | 2.9 | 3.3 |
| HH. Cons Real | 2.3 | 3.2 | 3.4 | 2.6 | 3.2 |
| Investment real | 1.5 | 4.3 | 12.5 | 3.0 | 4.3 |
| Exports real | 4.2 | 2.9 | 3.9 | 3.7 | 2.6 |
| Imports real | 3.1 | 2.6 | 4.7 | 2.8 | 2.9 |

In this example, a new table is generated with more conventional timeslices.

```
with mpak.set_smpl(2026, 2050):
    tab2 = mpak.table(pat,title='GDP components', name='long_table_2',timeslice=[2026,
→2030, 2035 , 2040, 2045, 2050]);

tab2.show
```

| GDP components | 2026 | 2030 | 2035 | 2040 | 2045 | 2050 |
|------------------------|------|------|------|------|------|------|
| --- Percent growth --- | | | | | | |
| Real GDP | 2.42 | 2.75 | 3.37 | 3.41 | 3.36 | 3.33 |
| HH. Cons Real | 2.32 | 2.46 | 3.12 | 3.17 | 3.18 | 3.20 |
| Investment real | 1.46 | 2.92 | 3.80 | 4.29 | 4.34 | 4.25 |
| Exports real | 4.20 | 3.59 | 3.19 | 2.93 | 2.75 | 2.64 |
| Imports real | 3.10 | 2.51 | 2.59 | 2.58 | 2.72 | 2.88 |

14.2.5 Other `.table` options

The `table` object has two kinds of options:

- 1) so-called **line options** that can only be set when the basic table is created with the `.table()` call.
- 2) so-called **table options** can be reset using the `set_options` method.

Line Options These options are set on table creation.

| Argument | Default Value | Description |
|----------|--|---|
| pat | '#Headline' (defined in the ModelFlow class) | variable names to be displayed, may include wildcard specifications |
| datatype | 'growth' | Defines the data transformation displays (cf. next table) |
| mul | 1.0 | Multiplier of values before display |
| dec | 2 | set the decimal places to display |
| col_desc | as shown below | A centered Description of columns (non transposed table) |
| heading | " | A centered text above columns (non transposed table) |
| rename | True | If True use descriptions else variable names |

Table Options These options can be revised after a table has been generated.

| Argument | Default Value | Description |
|--------------------|---------------|--|
| name | " | Name for this display. |
| custom_description | { } | Custom description, override default descriptions. |
| title | " | Title. |
| foot | " | Footer. |
| transpose | False | If True, transposes the table |
| chunk_size | 0 | Specifies the number of columns per chunk in tables. |
| timeslice | [] | Specifies the time slice for data display. |
| max_cols | 6 | Maximum columns when displayed as string. |
| last_cols | 3 | In Latex, the number of last columns in a display slice. |
| smpl | (",") | set or re-set the smpl for the table |

Some table examples

Displaying data as the difference in levels. The datatype=diflevel option causes the difference between the level of the selected variables in the .lastdf and .basedf databases to be displayed.

```
mpak.table(pat,datatype = 'diflevel').show
```

| Table | 2025 | 2026 | 2027 | 2028 | 2029 |
|-----------------------|-------|-------|-------|------------|------------|
| --- Impact, Level --- | | | | | |
| Real GDP | 0.01 | 0.01 | 0.01 | 462,356.71 | 474,302.09 |
| HH. Cons Real | 0.00 | 0.01 | 0.01 | 227,277.15 | 214,784.60 |
| Investment real | 0.00 | 0.01 | 0.01 | 336,198.25 | 361,924.14 |
| Exports real | -0.00 | -0.00 | -0.00 | -530.52 | -2,078.90 |
| Imports real | 0.00 | 0.01 | 0.01 | 139,487.97 | 148,662.16 |

The col_desc option causes the default descriptions of variables to be overwritten

The user may change these default descriptors using by setting the `col_desc` option directly. In the example below, the default text displayed in the previous table is revised to something that is more accurate for this particular example.

```
mpak.table(pat, datatype = 'diflevel',
           col_desc    = 'Delta, LCU, base year 2000'
           ).show
```

| Table | 2025 | 2026 | 2027 | 2028 | 2029 |
|-----------------|------------------------------------|-------|-------|------------|------------|
| | --- Delta, LCU, base year 2000 --- | | | | |
| Real GDP | 0.01 | 0.01 | 0.01 | 462,356.71 | 474,302.09 |
| HH. Cons Real | 0.00 | 0.01 | 0.01 | 227,277.15 | 214,784.60 |
| Investment real | 0.00 | 0.01 | 0.01 | 336,198.25 | 361,924.14 |
| Exports real | -0.00 | -0.00 | -0.00 | -530.52 | -2,078.90 |
| Imports real | 0.00 | 0.01 | 0.01 | 139,487.97 | 148,662.16 |

If no description is required set `col_desc=' '` That is a single blank.

```
mpak.table(pat, datatype = 'diflevel',
           col_desc    = ' '
           ).show
```

| Table | 2025 | 2026 | 2027 | 2028 | 2029 |
|-----------------|-------|-------|-------|------------|------------|
| Real GDP | 0.01 | 0.01 | 0.01 | 462,356.71 | 474,302.09 |
| HH. Cons Real | 0.00 | 0.01 | 0.01 | 227,277.15 | 214,784.60 |
| Investment real | 0.00 | 0.01 | 0.01 | 336,198.25 | 361,924.14 |
| Exports real | -0.00 | -0.00 | -0.00 | -530.52 | -2,078.90 |
| Imports real | 0.00 | 0.01 | 0.01 | 139,487.97 | 148,662.16 |

14.3 Complex tables

In the real world more complex tables are often called for, either including a mixture of data display type (some series in levels, others as a percent of GDP) or ones that mix data and text in ways the simple table object does not deal with easily.

A simple solution is to produce and then join two tables, which can be done with the `|` operator. More complex solutions would involve the `report` object which allows different table, text and plot elements to be joined in a single report (see the discussion at the end of this chapter).

14.3.1 The | operator joins non-transposed tables

The `|` operator allows two or more table objects (vertical not transposed) to be combined.

This allows one table expressed in levels (or growth rates) to be combined with another table of data expressed as a percent of GDP. The more complex conjoined table can therefore mix a number of different data representations.

When constructing more complex tables, the heading parameter (`heading=some text`) can be used to delineate different sections within the combined table. NB: The option `heading` is case sensitive, `Heading=` will do nothing.

Note

Non-transposed tables can be joined with transposed tables and figures using the reports functionality described at the end of this chapter.

Below a simple table with a header.

```
test=mpak.table(heading=r'A test header',pat="PAKNYGDPMKTPKN PAKNYGDPPOTLKN")
test.show
```

| | 2025 | 2026 | 2027 | 2028 | 2029 |
|--------------------------------|------|------|------|------|------|
| A test header | | | | | |
| --- Percent growth --- | | | | | |
| Real GDP | 2.08 | 2.42 | 2.64 | 4.36 | 2.91 |
| Potential Output, constant LCU | 2.83 | 2.86 | 2.88 | 2.94 | 3.03 |

In this example, a more complex table is created as a combination of the earlier generated tables `tab` and `tab_gov`, each using different display types.

```
(tab | tab_gov).show
```

| In percent of GDP | 2025 | 2026 | 2027 | 2028 | 2029 |
|---|------|------|------|-------|------|
| --- Percent growth --- | | | | | |
| Real GDP | 2.08 | 2.42 | 2.64 | 4.36 | 2.91 |
| HH. Cons Real | 2.03 | 2.32 | 2.48 | 3.45 | 2.62 |
| Investment real | 1.10 | 1.46 | 1.82 | 12.46 | 2.98 |
| Exports real | 4.37 | 4.20 | 4.04 | 3.88 | 3.72 |
| Imports real | 3.10 | 3.10 | 3.02 | 4.68 | 2.84 |
| --- Percent of GDP --- | | | | | |
| General Government Revenue, Deficit, LCU mn | -3.0 | -3.0 | -2.9 | -2.8 | -2.8 |
| General government gross debt millions lcu | 57.0 | 57.3 | 57.7 | 57.1 | 57.5 |
| Current Account Balance, US\$ mn | -3.6 | -3.5 | -3.4 | -3.5 | -3.4 |
| Source: World Bank | | | | | |

Below several different tables are joined to create quite a complex table.

`tab_total` below is generated as the of several sub tables and illustrates the use of the heading command and the display in a single object of a relatively complex table.

```
pat= '*NYGDPMKTPKN *NECONPRVTKN *NEGDIFTOTKN *NEEXPGNF SKN *NEIMPGNFSKN'
pat_gov = '*GGBALOVRLCN *GGDBTTOTLCN *BNCABFUND_CD'
tab_na_base = mpak.table(pat, datatype='basegrowth',
                           heading=r'Baseline')
tab_gov_base = mpak.table(pat_gov,datatype='basegdppct')
tab_na = mpak.table(pat, datatype='growth',
                           heading=r'Alternative')
tab_gov = mpak.table(pat_gov,datatype='gdppct')
tab_na_dif = mpak.table(pat,datatype='difgrowth',
                           heading=r'Impact')
tab_gov_dif = mpak.table(pat_gov,datatype='difgdppct')
# Use the / to concatenate the tables
tab_total = tab_na_base | tab_gov_base | tab_na | tab_gov | tab_na_dif | tab_gov_dif
```

`tab_total.show`

Table

| | 2025 | 2026 | 2027 | 2028 | 2029 |
|---|-------|-------|-------|-------|-------|
| Baseline | | | | | |
| Real GDP | 2.08 | 2.42 | 2.64 | 2.79 | 2.91 |
| HH. Cons Real | 2.03 | 2.32 | 2.48 | 2.59 | 2.69 |
| Investment real | 1.10 | 1.46 | 1.82 | 2.17 | 2.51 |
| Exports real | 4.37 | 4.20 | 4.04 | 3.90 | 3.77 |
| Imports real | 3.10 | 3.10 | 3.02 | 2.89 | 2.78 |
| --- Baseline Percent of GDP --- | | | | | |
| General Government Revenue, Deficit, LCU mn | -3.05 | -2.98 | -2.95 | -2.92 | -2.91 |
| General government gross debt millions lcu | 57.02 | 57.26 | 57.69 | 58.26 | 58.94 |
| Current Account Balance, US\$ mn | -3.55 | -3.46 | -3.40 | -3.37 | -3.36 |
| Alternative | | | | | |
| --- Percent growth --- | | | | | |
| Real GDP | 2.08 | 2.42 | 2.64 | 4.36 | 2.91 |
| HH. Cons Real | 2.03 | 2.32 | 2.48 | 3.45 | 2.62 |
| Investment real | 1.10 | 1.46 | 1.82 | 12.46 | 2.98 |
| Exports real | 4.37 | 4.20 | 4.04 | 3.88 | 3.72 |
| Imports real | 3.10 | 3.10 | 3.02 | 4.68 | 2.84 |
| --- Percent of GDP --- | | | | | |
| General Government Revenue, Deficit, LCU mn | -3.05 | -2.98 | -2.95 | -2.81 | -2.84 |
| General government gross debt millions lcu | 57.02 | 57.26 | 57.69 | 57.11 | 57.55 |
| Current Account Balance, US\$ mn | -3.55 | -3.46 | -3.40 | -3.48 | -3.43 |
| Impact | | | | | |
| --- Impact, Percent growth --- | | | | | |
| Real GDP | 0.00 | 0.00 | 0.00 | 1.57 | -0.00 |
| HH. Cons Real | 0.00 | 0.00 | 0.00 | 0.86 | -0.07 |
| Investment real | 0.00 | 0.00 | 0.00 | 10.29 | 0.47 |
| Exports real | -0.00 | -0.00 | -0.00 | -0.02 | -0.05 |
| Imports real | 0.00 | 0.00 | 0.00 | 1.79 | 0.06 |
| --- Impact, Percent of GDP --- | | | | | |
| General Government Revenue, Deficit, LCU mn | 0.00 | 0.00 | 0.00 | 0.11 | 0.07 |
| General government gross debt millions lcu | -0.00 | -0.00 | -0.00 | -1.15 | -1.39 |
| Current Account Balance, US\$ mn | 0.00 | -0.00 | -0.00 | -0.11 | -0.06 |

Below the rendering of the same table is revised by using the `set_options(smpl=(2025, 2026))` modifier on the command line.

```
#  
tab_total.set_options(smpl=(2026,2030)).show
```

| Table | 2026 | 2027 | 2028 | 2029 | 2030 |
|---|-------|-------|-------|-------|-------|
| Baseline | | | | | |
| Real GDP | 2.42 | 2.64 | 2.79 | 2.91 | 3.03 |
| HH. Cons Real | 2.32 | 2.48 | 2.59 | 2.69 | 2.78 |
| Investment real | 1.46 | 1.82 | 2.17 | 2.51 | 2.84 |
| Exports real | 4.20 | 4.04 | 3.90 | 3.77 | 3.65 |
| Imports real | 3.10 | 3.02 | 2.89 | 2.78 | 2.69 |
| --- Baseline Percent of GDP --- | | | | | |
| General Government Revenue, Deficit, LCU mn | -2.98 | -2.95 | -2.92 | -2.91 | -2.90 |
| General government gross debt millions lcu | 57.26 | 57.69 | 58.26 | 58.94 | 59.69 |
| Current Account Balance, US\$ mn | -3.46 | -3.40 | -3.37 | -3.36 | -3.37 |
| Alternative | | | | | |
| --- Percent growth --- | | | | | |
| Real GDP | 2.42 | 2.64 | 4.36 | 2.91 | 2.75 |
| HH. Cons Real | 2.32 | 2.48 | 3.45 | 2.62 | 2.46 |
| Investment real | 1.46 | 1.82 | 12.46 | 2.98 | 2.92 |
| Exports real | 4.20 | 4.04 | 3.88 | 3.72 | 3.59 |
| Imports real | 3.10 | 3.02 | 4.68 | 2.84 | 2.51 |
| --- Percent of GDP --- | | | | | |
| General Government Revenue, Deficit, LCU mn | -2.98 | -2.95 | -2.81 | -2.84 | -2.85 |
| General government gross debt millions lcu | 57.26 | 57.69 | 57.11 | 57.55 | 58.27 |
| Current Account Balance, US\$ mn | -3.46 | -3.40 | -3.48 | -3.43 | -3.38 |
| Impact | | | | | |
| --- Impact, Percent growth --- | | | | | |
| Real GDP | 0.00 | 0.00 | 1.57 | -0.00 | -0.27 |
| HH. Cons Real | 0.00 | 0.00 | 0.86 | -0.07 | -0.32 |
| Investment real | 0.00 | 0.00 | 10.29 | 0.47 | 0.09 |
| Exports real | -0.00 | -0.00 | -0.02 | -0.05 | -0.06 |
| Imports real | 0.00 | 0.00 | 1.79 | 0.06 | -0.17 |
| --- Impact, Percent of GDP --- | | | | | |
| General Government Revenue, Deficit, LCU mn | 0.00 | 0.00 | 0.11 | 0.07 | 0.04 |
| General government gross debt millions lcu | -0.00 | -0.00 | -1.15 | -1.39 | -1.42 |
| Current Account Balance, US\$ mn | -0.00 | -0.00 | -0.11 | -0.06 | -0.01 |

14.4 The `.plot()` class

The `.plot()` class can be used to display results graphically.

The ModelFlow class `.plot` extends the standard python charting libraries, making them aware of the ModelFlow databases (including `kept` DataFrames) and provides functionality to display data and results in ways commonly used by macroeconomic modelers.

The class's constructor is the `.plot` method which returns an object of type `DisplayVarFigDef`.

The constructor for `.plot` takes four arguments:

1. **pattern** (`pat_plot` in the example below) which represents a space delimited string containing the mnemonics of the variables (which must be part of the model object to be charted). As with standard ModelFlow search specifications, either mnemonic names, or descriptions can be specified in wildcard specifications.

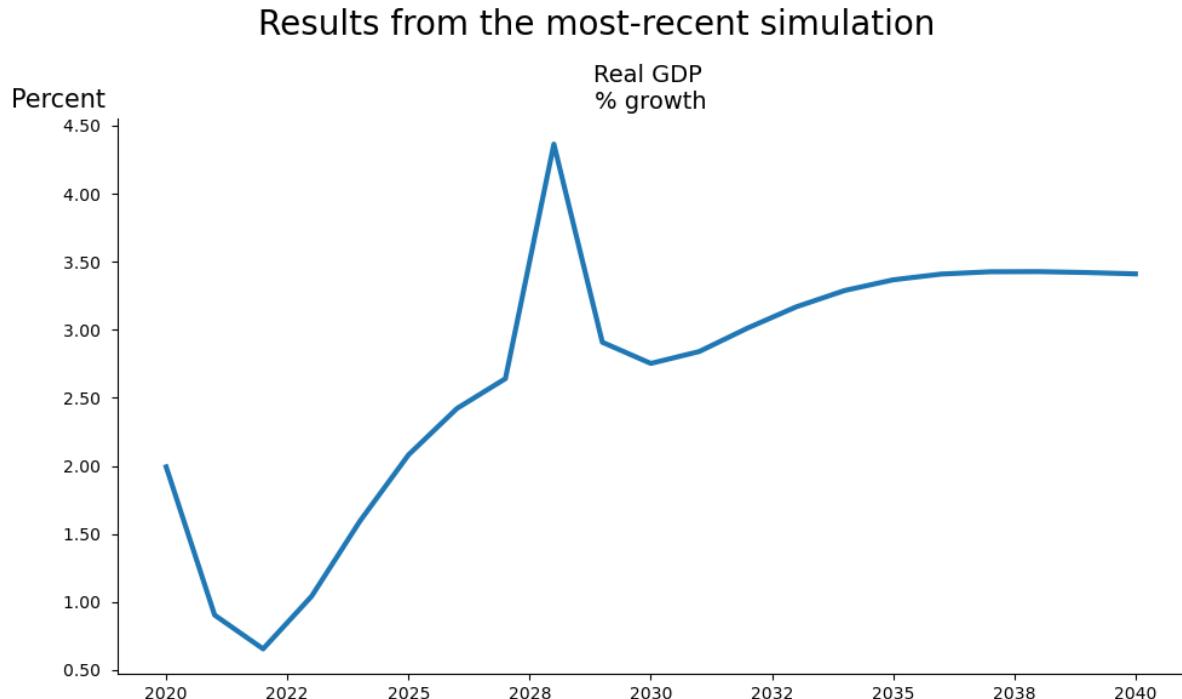
2. **plot_name** an identifier for the plot object, used when storing charts that are part of a plot.
3. **title** A title that will be displayed with any charts that are generated.
4. **scenarios**, an optional string concatenating with the | symbol the text names of scenarios to be plotted. If left blank (the default) plot will display data from the `.lastdf` DataFrame or compare the `.lastdf` results with the data from the `.basedf` DataFrame.

Below the sample period for the model object `mpak` is set, the pattern that will determine which variables are to be displayed is defined, and a plot object `fig` is instantiated using the variables in `pat_plot`, with the name 'My_first_plot' and with the displayed title "Results from the most-recent simulation".

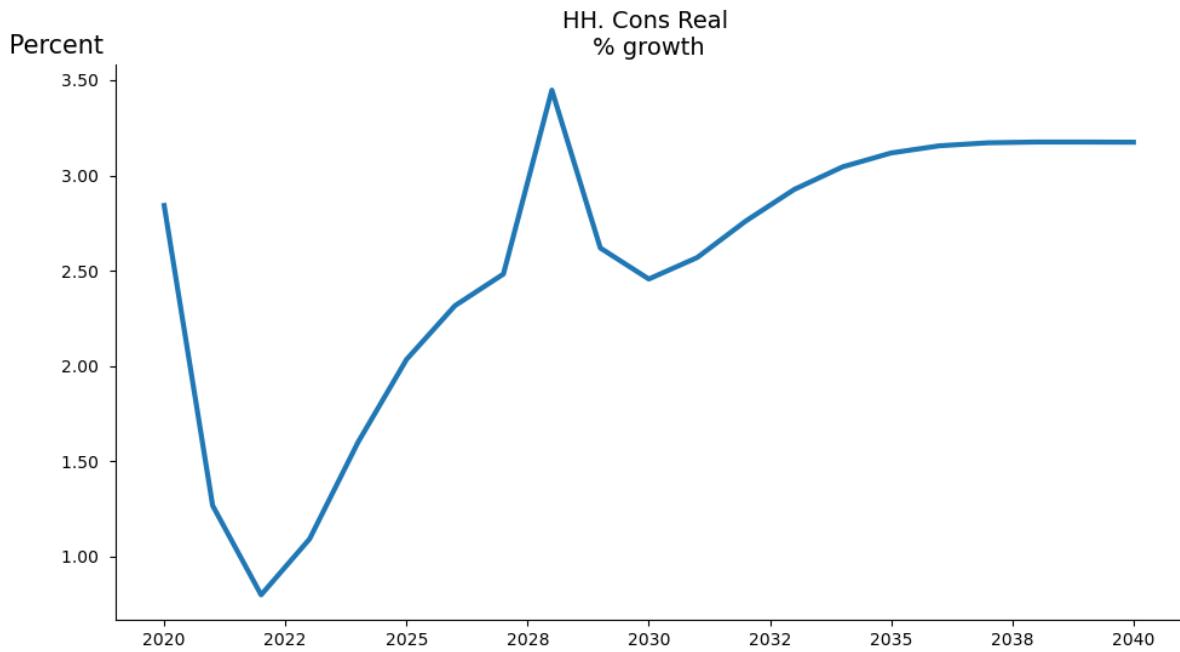
```
mpak.smpl(2020,2040)
pat_plot= '*NYGDPMKT_PKN *NECONPRVT_KN '
fig = mpak.plot(pat_plot, name='My_first_plot', title="Results from the most-recent simulation")
```

The object is an instance of the class `DisplayVarFigDef`. Its construction does not cause the figure to be displayed. Unless the `.show` method is called explicitly as below.

```
fig.show;
```



Results from the most-recent simulation



14.4.1 Rendering a plot object

Like the table object, the plot object has several methods for rendering its figure(s).

These include:

- the name of the object – displays an html widget of the object
- `display(object)` will display a html widget of the object.
- `object.show` will show all the charts sequentially
- `object.pdf()`

14.4.2 The `display(fig)` / name of fig command

The first two options: the name of the figure itself executed as a command; and `display(nameoffigure)` generate exactly the same results.

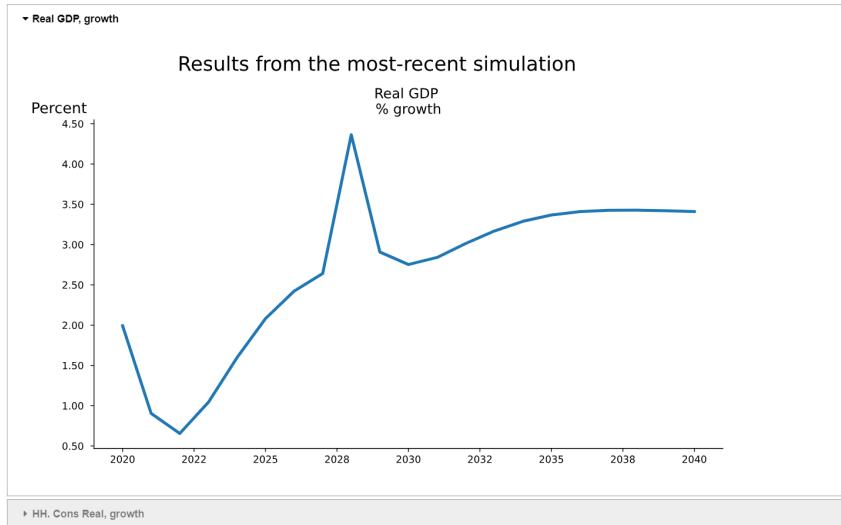
Under Jupyter Notebook, these two methods generate an “accordion” widget that allows the user to flip through the various charts (possibly just one) that comprise the plot object.

Different charts in the object can be visualized by selecting the triangle beside the variable name (found on the lefthand side of the screen). Each chart within the plot object has its own triangle beside its name. Selecting one, will fold-up any other figures that may be open and unfold and display the chart associated with the selected variable.

The below command generates the widget comprised of two charts (if more series had been indicated when the figure was instantiated, more widget items would be displayed), one for real GDP growth and the second for real household consumption growth. By default the first is displayed, it can be minimized by clicking on the triangle beside its name. The second (household consumption) will appear under it (below the chart if its is displayed). Click on the triangle beside its name will cause the widget to change the displayed figure to the household consumption chart. Directly clicking on

household consumption would have had the same effect, first closing the GDP chart and then displaying the newly selected chart.

```
display(fig)
```

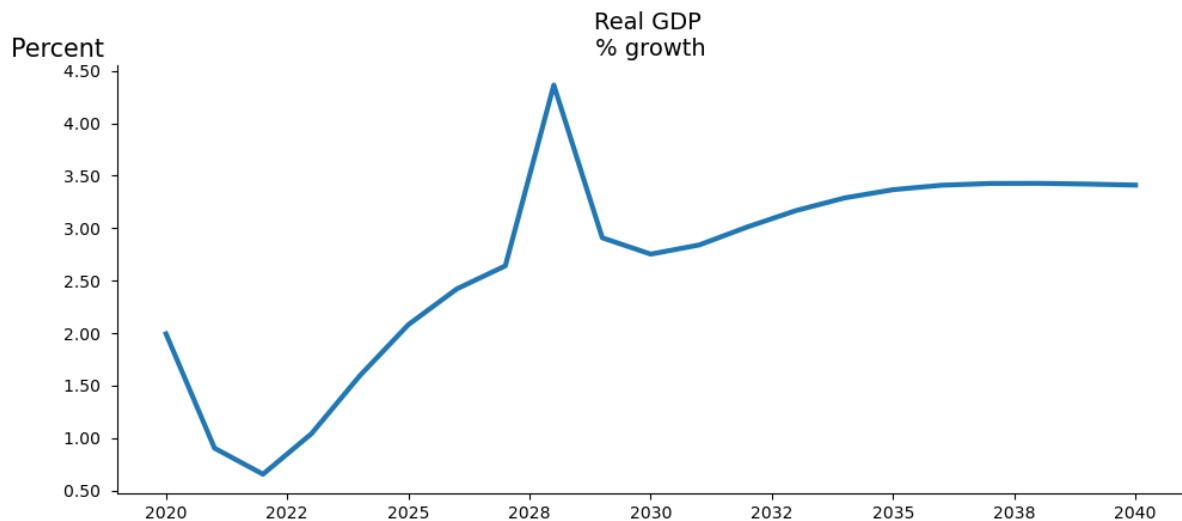


The `.show` method causes each chart in the figure to be displayed sequentially

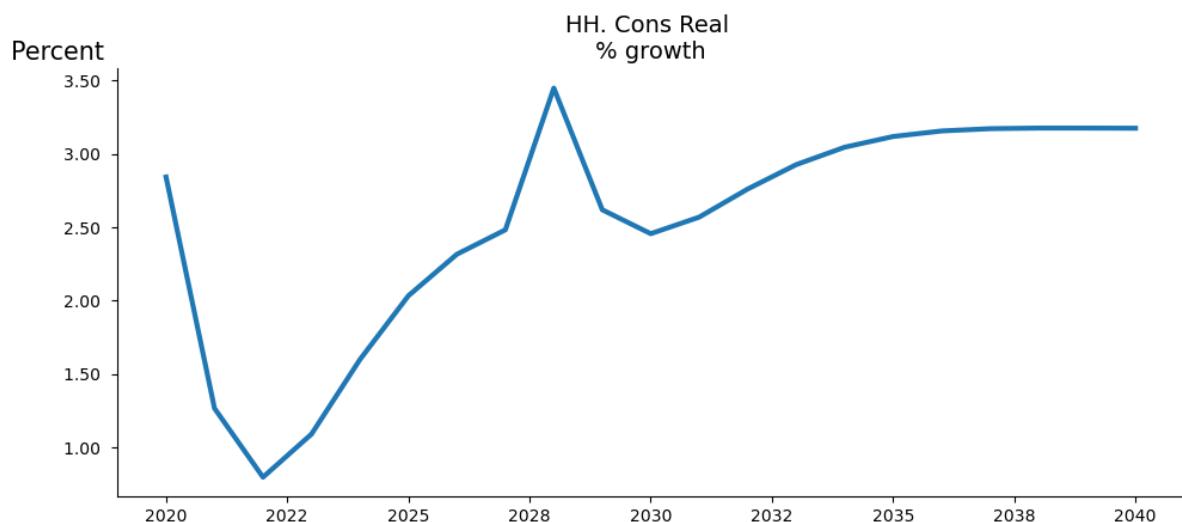
Here the `.set_options()` method is used to adjust the display size of the rendered charts.

```
fig.set_options(size=(10, 5)).show
```

Results from the most-recent simulation



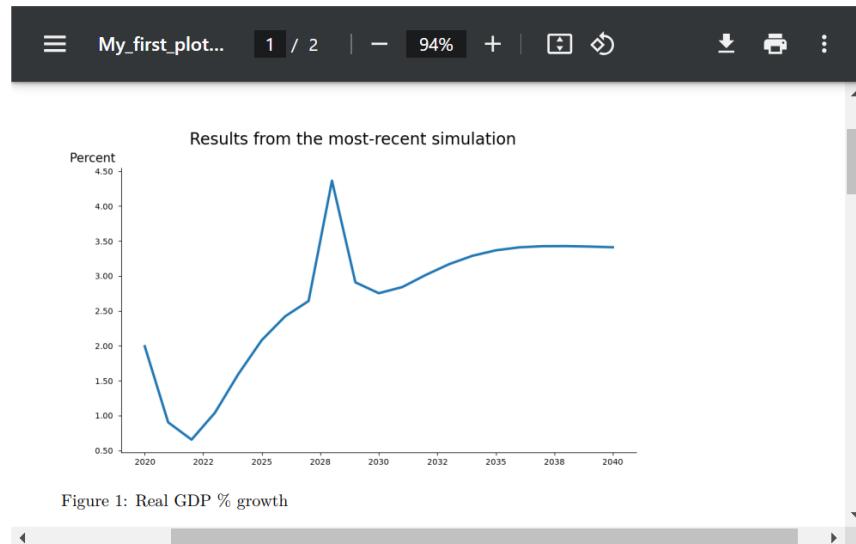
Results from the most-recent simulation



14.4.3 .pdf Creates a .pdf version of the plots

The `.pdf` method will generate a pdf of the chart(s) that form part of the figure. Each chart will appear sequentially in a single pdf. The generated pdf will be found in a sub-directory with the name that was given to the figure as the root of the pdf file that was generated.

```
fig.pdf()
```



14.4.4 '.plot' options

The `.plot()` object has many options. Like Table these can be split into those that can be set only when the figure is instantiated and those that can be revised later using the `.set_options()` method. Both types of options are listed below.

Line options are set when the plot is created with the `.plot()` call and can only be set when the object is instantiated.

| Argument | Default Value | Description |
|-------------------|---------------|---|
| pat | '#Headline' | Pattern with wildcard for variable names |
| datatype | 'growth' | Defines the datatransformation displayes (cf. next table) |
| mul | 1.0 | Multiplier of values before display |
| ax_title_template | " | Template for each chart title |

Plot options can be revised after the plot has been created.

| Argument | Default Value | Description |
|--------------------|---------------|---|
| rename | True | If True use descriptions else variable names |
| scenarios | " | Show for <code>.basedf/.lastdf</code> or selected scenarios |
| name | " | Name for this display. |
| custom_description | {} | Custom description, override default descriptions. |
| title | " | Title (used when samefig=True). |
| samefig | False | If True, displays all chart in the same plot area. |
| ncol | 2 | Number of columns when samefig=True. |
| size | (10, 6) | Specifies the size of each chart |
| legend | True | If True, includes a legend in the display. |
| smpl | (",") | set smpl for this plot |

Like tables, the transformation of the data displayed is controlled by the datatype options, whose parameters are identified below.

14.4.5 Plot Data Types and Scenarios

Tables display output based only on `.basedf` and `.lastdf`, the `.plot` function can also be used to visualize charts based on the scenarios stored in the `.keep_solutions` dictionary.

- If `scenarios=''`, `.plot` will display lines corresponding to the data shown in tables, which is based on `.basedf` and `.lastdf`.
- If `scenarios='<list of scenarios>'`, `.plot` will display lines for the data associated with the scenarios specified in the list.
- If `scenarios='*'`, all available scenarios will be plotted.
- If `scenarios='base_last'`, `.basedf` and `.lastdf` will be plotted as scenarios.

The table below summarizes how the combination of `datatype` and `scenarios` determines the data displayed in the chart.

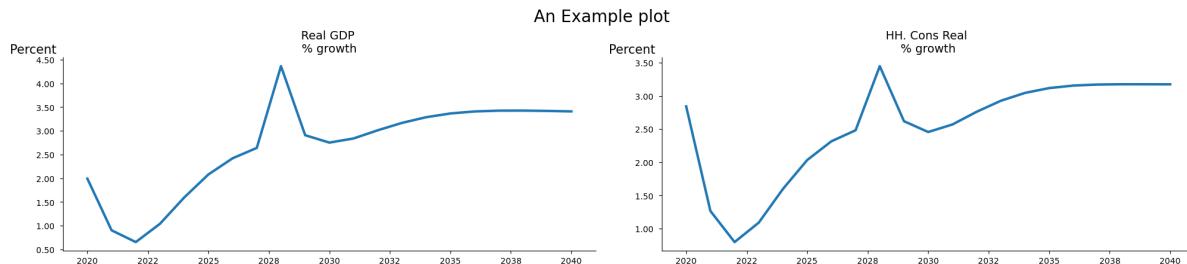
Table: Datatype and Scenario Settings

| Datatype | <code>scenarios="</code> or omitted | <code>scenarios='<list of scenarios>'</code> (Multiple Scenarios) |
|-------------------------------|--|---|
| Value Views | | |
| <code>growth</code> (default) | Growth rate (%) in the most recent dataset (<code>.lastdf</code>). | Growth rates for all scenarios. |
| <code>level</code> | Values from <code>.lastdf</code> . | Values for all scenarios. |
| <code>gdppct</code> | Percentage of GDP in <code>.lastdf</code> . | Percentage of GDP for all scenarios. |
| <code>qoq_ar</code> | Quarterly growth (annualized) for quarterly models in <code>.lastdf</code> . | Quarterly growth (annualized) for all scenarios. |
| Difference Views | | |
| <code>difgrowth</code> | Change (Δ) in growth rates between <code>.lastdf</code> and <code>.basedf</code> . | Change (Δ) in growth rates between the first and other scenarios. |
| <code>diflevel</code> | Change (Δ) in values between <code>.basedf</code> and <code>.lastdf</code> . | Change (Δ) in values between the first and other scenarios. |
| <code>difgdppct</code> | Change (Δ) in the percentage of GDP between <code>.basedf</code> and <code>.lastdf</code> . | Change (Δ) in the percentage of GDP between the first and other scenarios. |
| <code>difqoq_ar</code> | Change in quarterly growth (annualized) between <code>.basedf</code> and <code>.lastdf</code> . | Change (Δ) in quarterly growth (annualized) between the first and other scenarios. |
| <code>difpctlevel</code> | Percentage change (Δ) in values between <code>.basedf</code> and <code>.lastdf</code> . | Percentage change (Δ) in values between the first and other scenarios. |
| .basedf Views | | |
| <code>basegrowth</code> | Growth rate (%) in <code>.basedf</code> . | Growth rates for the first scenario. |
| <code>baselevel</code> | Values from <code>.basedf</code> . | Values for the first scenario. |
| <code>basegdppct</code> | Percentage of GDP in <code>.basedf</code> . | Percentage of GDP for the first scenario. |
| <code>baseqoq_ar</code> | Quarterly growth (annualized) in <code>.basedf</code> . | Quarterly growth (annualized) for the first scenario. |

The 'samefig=True' option causes charts to be rendered on a grid

Often a single call to `.plot()` will result in many charts being produced. The `samefig=True` option will cause them to be displayed in a grid (rather than the alternative where each chart occupies the full width of the screen or page). This approach both saves real-estate but actually facilitates the comparison of results – across variables, as in the case below; or when used in combination with the `by_vars` option, across scenarios.

```
fig.set_options(samefig=True, title='An Example plot').show
```



14.4.6 The `by_var` option determines whether to show results by variable (False) or scenario (True)

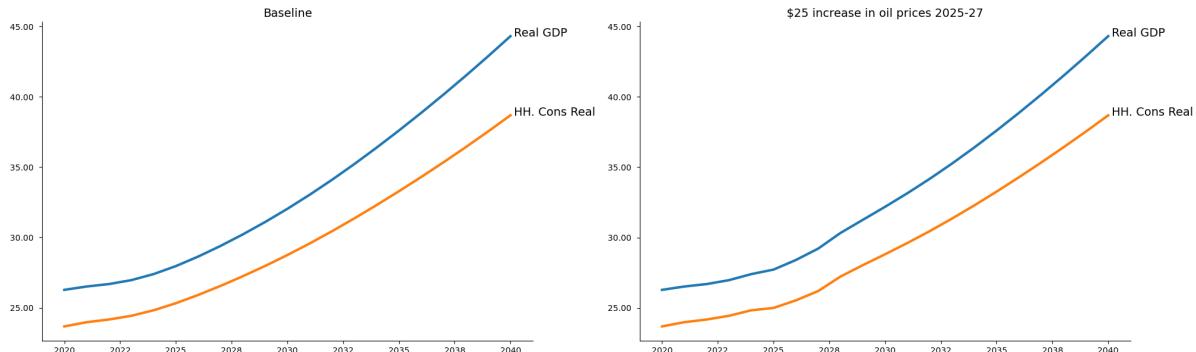
By default, `by_var` is set to `True` and a separate chart is created for each variable in the `pat` pattern. If multiple scenarios are being inspected, each chart will have a separate line for each scenario.

Setting `by_var` to `False` causes `plot` to generate the charts in the figure by scenario. Each chart shows the value for all specified variables in one scenario, with following charts showing the results for different scenarios.

| <code>by_var</code> | a plot for each | a line in each plot for each |
|-----------------------------|-----------------|------------------------------|
| <code>True</code> (default) | Variable | Scenario |
| <code>False</code> | Scenario | Variable |

The example below sets both the `by_var` and `samefig` options to `True`. As a result, the charts are displayed in a grid and data for GDP and household consumption (the variables) from one scenario are shown on each graph, with results for one scenario on each graph.

```
mpak.plot(pat_plot, name='fig_by_scenario',
           scenarios='Baseline|$25 increase in oil prices 2025-27',
           datatype='level', samefig=True,
           by_var=False, legend=False, mul=1/1_000_000).show
```

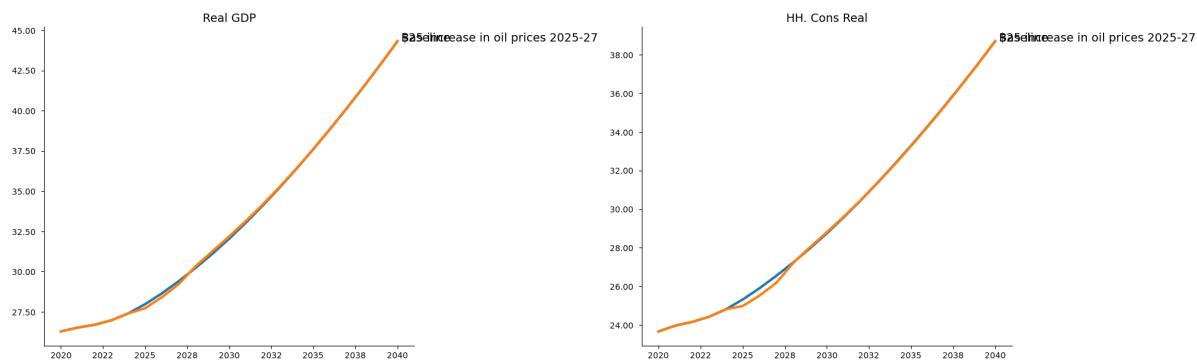


The example also illustrates the use of the **legend** and **mul** options. The option `legend=False` suppresses the legend (as a box with all series description in one place) and instead causes the series labels to be placed to the right of the line of each rendered series.

The `mul=1/1_000_000` option causes the axis to be rendered in millions. Notice python allows large numbers to be split by `_` which makes them more readable. It is not necessary to do this.

Below the same two results are shown but organized by variable with the `by_var` option set to True.

```
mpak.plot(pat_plot, name='fig_by_scenario',
          scenarios='Baseline|$25 increase in oil prices 2025-27',
          datatype='level', samefig=True,
          by_var=True, legend=False, mul=1/1_000_000).show
```



14.4.7 The `scenarios=` option selects the scenarios to be displayed

The `scenarios=` option allows the user to specify which scenarios (from the kept scenarios in the model object) to use in generating the plots. Scenarios names must be the exact match to the text used when the `keep` command was executed, and must be separated by the `|` symbol.

Recall

To find all names of the stored scenarios use:

```
for key in mpak.keep_solutions:
    print(key);
```

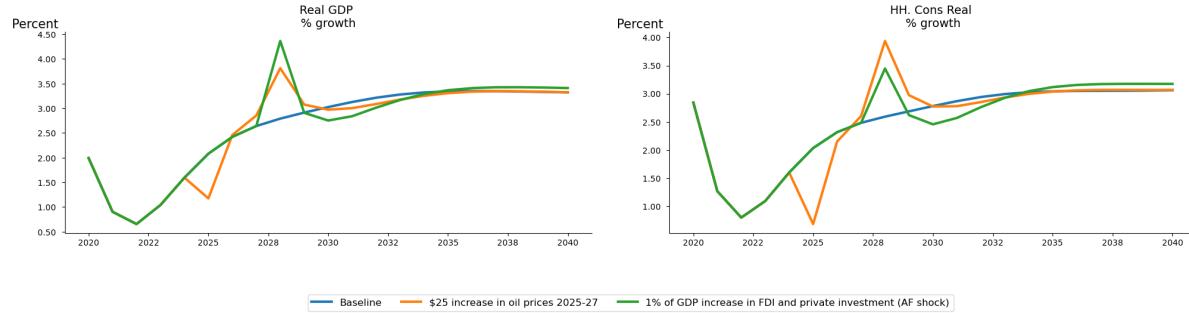
```
Baseline
$25 increase in oil prices 2025-27
2.5% increase in C 2025-40
2.5% increase in C 2025-27 -- exog whole period
2.5% increase in C 2025-27 -- exog whole period --KG=True
2.5% increase in C 2025-27 -- temporarily exogenized
1% of GDP increase in FDI and private investment (AF shock)
```

In the example below, the same pattern as before is utilized (real GDP and real household expenditure), but the `scenarios` option indicates that the charts should be generated from three scenarios:

1. Baseline
2. “\$25 increase in oil prices 2025-27”;
3. “1% of GDP increase in FDI and private investment (AF shock)”.

In this use-case the datatype='growth' causes the growth rate from each scenario to be rendered.

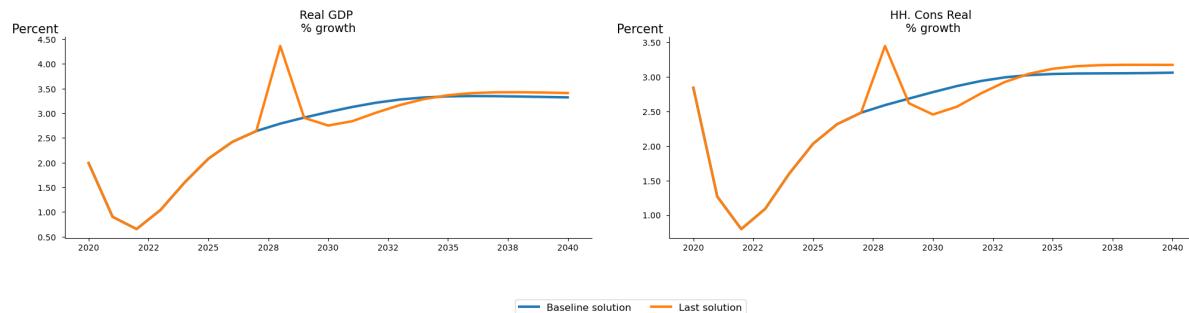
```
mpak.plot(pat_plot, name='fig_select', datatype='growth', samefig=True,
          scenarios='Baseline|$25 increase in oil prices 2025-27|1% of GDP increase in
          ↪FDI and private investment (AF shock)').show
```



The special scenario base_last

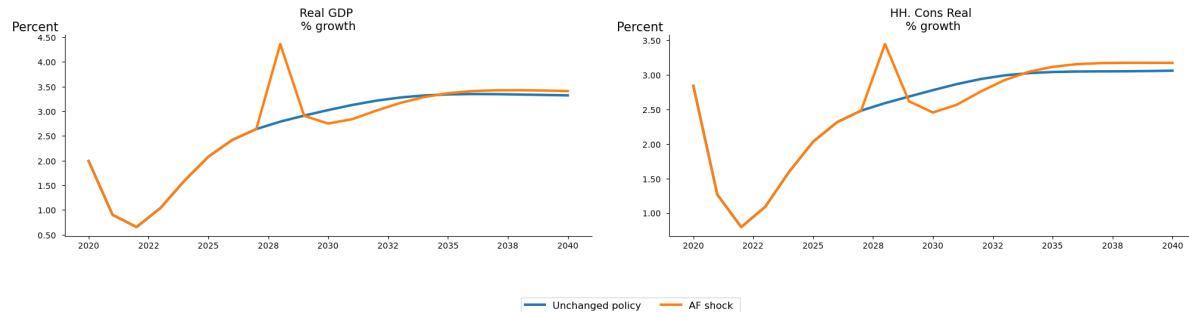
The scenario option `base_last` will use the `.basedf` and `.lastdf` as scenarios dataframes as the scenarios.

```
mpak.plot(pat_plot, name='fig_select', datatype='growth', samefig=True,
          scenarios='base_last').show
```



`.basename.lastname` can be used to rename the scenarios

```
mpak.basename = 'Unchanged policy'
mpak.lastname = 'AF shock'
mpak.plot(pat_plot, name='fig_select', datatype='growth', samefig=True,
          scenarios='base_last').show
```



14.4.8 Template for each chart title: `ax_title_template`

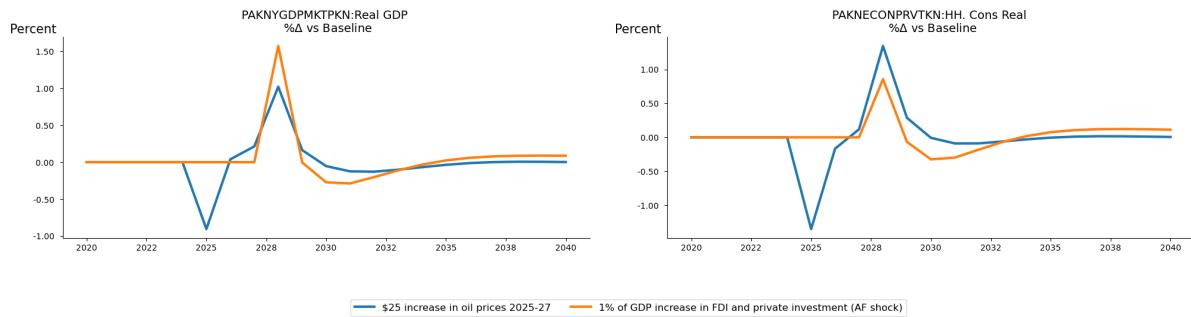
The text associated with each graph can be customized by using the `ax_title_template` option.

`ax_title_template` can be sent to any arbitrary string. If the string contains the special characters `{var_name}`, and/or `{var_description}` or `{compare}` then the name of the variable being displayed (or its description) or the name of the baseline scenario will be substituted for the expression in the figure.

| Placeholder | Replaced by |
|--------------------------------|------------------------------------|
| <code>{var_name}</code> | Variable Mnemonic |
| <code>{var_description}</code> | Variable description |
| <code>{compare}</code> | First scenario used for comparison |

In the example below, all three placeholders are utilized, and the text includes a LaTeX expression (the sub-string `\Delta`) bordered by `$` signs, which renders in the chart as the Greek symbol Δ .

```
mpak.plot(pat_plot, datatype='difgrowth',
           scenarios='Baseline|$25 increase in oil prices 2025-27|1% of GDP increase
           ↪in FDI and private investment (AF shock)',
           samefig=True,
           ax_title_template= r'{var_name}:{var_description} \n %%\Delta vs {compare}' ↪
           ).show
```



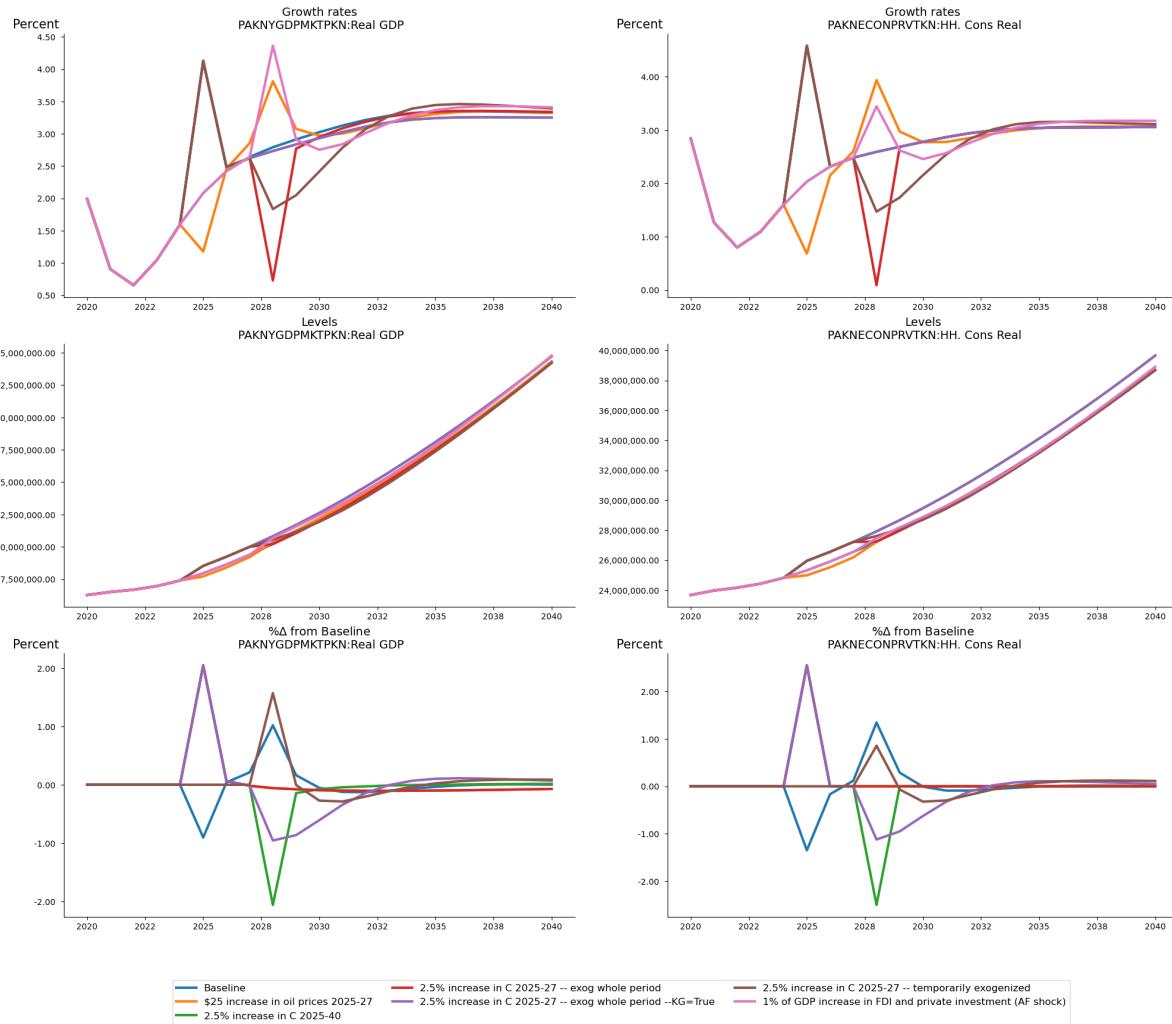
14.4.9 Joing plots with |

Like Tables, Plots can be concatenated together using the `|` operator. This allows plots with different datatypes to be displayed together.

In the example below, results for GDP in each scenario and household consumption for each scenario are displayed first as growth rates, secondly as levels, and finally as changes in the growth rates.

The three separate figures are then joined in the figure called `fig_`, with the `samefig=True` flag set so that the six separate charts are arranged in a grid to facilitate reading and comparison.

```
fig1 = mpak.plot(pat_plot, name='growth', scenarios='*', ax_title_template= r'Growth_
→rates\n {var_name}: {var_description}')
fig2 = mpak.plot(pat_plot, datatype='level', scenarios='*', ax_title_template= r'Levels\
→n {var_name}: {var_description}')
fig3 = mpak.plot(pat_plot, datatype='difgrowth', name= 'dif',
                  ax_title_template= r'%$\Delta$ from {compare}\n {var_name}: {var_
→description}',
                  scenarios='*')
fig_ = (fig1 | fig2 | fig3).set_options(samefig=True, name='fig_')
fig_.show
```



14.4.10 .savefigs() Saving plots in other formats.

Plots are created using the `matplotlib` library. If charts are going to be used further downstream they can be saved to file using a wide range of formats.

.savefigs() options

| Parameter | Type | Description | Default |
|----------------|------|--|-------------------|
| location | str | The folder in which to save the charts. | './graph' |
| experimentname | str | A subfolder under location where charts are saved. | 'experiment1' |
| addname | str | An additional name added to each figure filename. | '' (empty string) |
| extensions | list | A list of file extensions for saving the figures. | ['svg'] |
| xopen | bool | If True, open the saved figure locations in a web browser. | |

Charts can be saved using the following formats:

| Extension | Description |
|---------------|-----------------------------|
| .png | Portable Network Graphics |
| .jpg or .jpeg | Photographic Experts Group |
| .tif or .tiff | Tagged Image File Format |
| .bmp | Bitmap |
| .gif | Graphics Interchange Format |
| .svg | Scalable Vector Graphics |
| .pdf | Portable Document Format |
| .eps | Encapsulated PostScript |
| .ps | PostScript |
| .raw | Raw image data |
| .rgba | Raw RGBA bitmap |
| .pgf | Portable Graphics Format |

An example:

```
fig_for_saving = mpak.plot(pat_plot, name='for_saving')
fig_for_saving.savefigs()
```

```
'Saved at: graph/experiment1'
```

In this example the sub-directory name where the files will be saved is changed to Scenario1 and two copies of the each chart is saved, once as an SVG file the other as a PDF, by setting the extensions option to a list `extensions=['svg', 'pdf']`.

```
fig_for_saving = mpak.plot(pat_plot, name='for_saving', experimentname="Scenario1");
fig_for_saving.savefigs(extensions=['svg', 'pdf'])
```

```
'Saved at: graph/experiment1'
```

14.5 The `.text()` class

The `.text()` class is less complex than the `.plot()` and `.table()` procedures. It is mainly used implicitly in Reports, but can be declared as an object and used either in a report or on its own.

The text class allows the user to define three different types of text: `.text_text` `.html_text` and `.latex_text`. These properties are optional and can be declared explicitly or implicitly.

| object | delimited by | contains |
|--------------------------|---|------------|
| <code>.text_text</code> | nothing | Plain text |
| <code>.html_text</code> | <code><html> </html></code> | html text |
| <code>.latex_text</code> | <code><latex> </latex></code> | latex text |

A text object can be declared in the same way as the `.plot` or `table()`, called in conjunction with a model object. In the example below we create an object `mytext`, and implicitly set the `.text_text` property to "Some text". The html and latex properties are undeclared.

```
mytext=mpak.text("Some text")
mytext.show
```

```
Some text
```

In the example below a more complicated object is declared.

Note this is a multiline text object, and that the plain text, html and latex properties are all declared implicitly with the html text demarcated by the <html></html> tokens and the latex text similarly demarcated by the tokens.

```
multimodeText=mpak.text(r'''
Example of plain output
This is the plain contents of a multiline text
object.
<html><h1> Example of html output</h1>
This is the <b>html content</b> of a multiline text<b>
object.</b>
</html>
<latex>
\section*{Example of latex output}
This is the \textbf{latex content} of a multiline text
\textbf{object}
</latex>
'''')
```

The three different kinds of text that can be included in a text object can be extracted from the text object.

```
print(multimodeText.text_text)
print(multimodeText.html_text)
print(multimodeText.latex_text)
```

```
Example of plain output
This is the plain contents of a multiline text
object.
<h1> Example of html output</h1>
This is the <b>html content</b> of a multiline text<b>
object.</b>
\section*{Example of latex output}
This is the \textbf{latex content} of a multiline text
\textbf{object}
```

14.5.1 Renderings

When rendered as text, the text object will ignore the html and latex portions. When rendered as html it will ignore the text and latex. The pdf rendering ignores text and html.

Warning

Because the text of the three internal components of the text object (plain text, html and latex) are independent of one another they can hold completely unrelated text. While this might be useful in some contexts, in most, the three formulations should contain the same message even if the embellishments may differ somewhat given the capabilities of the rendering engine.

14.5.2 Html renderings

Executing the text object directly from the command line will cause it to render as html. By the same token, the function `Display()` when called on a text object will render it as text. NB: if there is no html object, the plain text version will be rendered.

```
multimodeText
```

Example of html output

This is the `html content` of a multiline text `object`.

```
display(multimodeText)
```

Example of html output

This is the `html content` of a multiline text `object`.

14.5.3 Simple text rendering

The `.show` method will render the object as plain text.

```
multimodeText.show
```

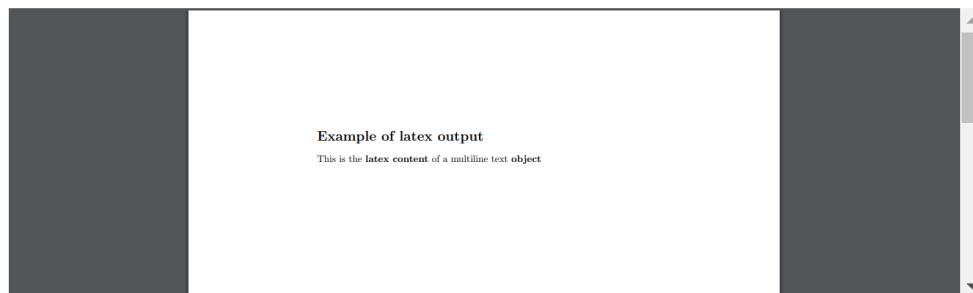
Example of plain output

This is the plain contents of a multiline text object.

14.5.4 Pdf rendering

Finally the `.pdf()` method will render the object's latex contents rendered to a pdf file.

```
multimodeText.pdf()
```



14.6 Reports

Joined tables and plots offer significant flexibility, but have limitations. In particular, joined tables must cover the same time-period, and tables cannot be joined with plots, while transposed tables cannot be joined with other tables.

Reports in ModelFlow are containers that can contain figures, plots, tables of different dimension and text. Moreover, Report objects can be saved in a model object and regenerated on new data without requiring any additional effort by the user.

14.6.1 Creating a report

A report can be generated by combining tables, plots, text and even other reports with the + operator.

Below a small report is generated from two tables generated earlier in this chapter (`tab` and its transpose `tab_t` and a the plot object `fig`.

```
mpak.smpl(2020,2040)
smallreport = (fig.set_options(samefig=True, name='fig_df', scenarios='base_last
˓→') + tab+tab_t).set_name('small report')
```

The code instantiates a new report object called `smallreport`. The objects included in the report are brought together by the + operator.

These include the previously defined object `fig`, whose options are adjusted with the `fig.set_options(samefig=True)` command. This alters the resulting output such that the charts in the original `fig` are organized in a grid (`samefig=True`). The Report is completed by adding the previously defined tables: `tab` with years as columns and, `tab_t` with years as rows.

The method `.set_name('small report')` gives the report a name, which identifies the container (and determines where its file outputs will be saved if a pdf version is generated).

Warning

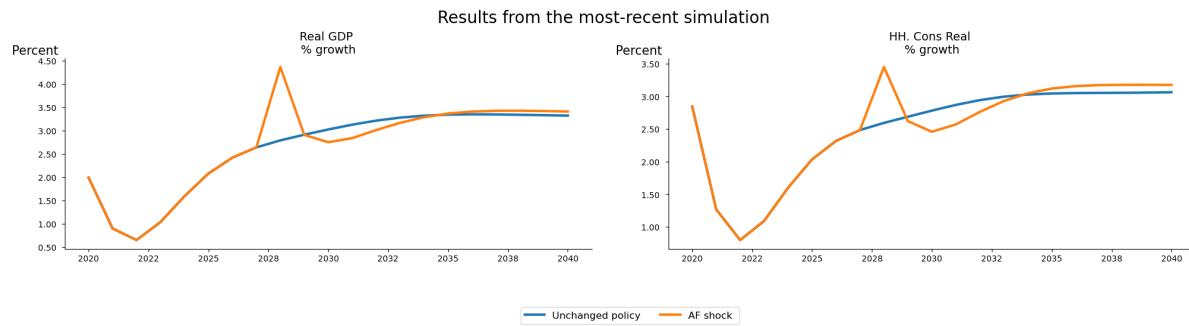
The report name will be used in a dictionary of reports to identify the report. As a result, users should ensure that each report has a unique name.

14.6.2 Rendering a report

Reports can be displayed using the same mechanisms as tables and plot. Either by executing the name of the report itself, or with the command `display(reportname)`, or with `reportname.show` or `reportname.pdf`.

Below is the output from the `.show` method

```
smallreport.show
```



GDP components

	2025	2026	2027	2028	2029
	--- Percent growth ---				
Real GDP	2.08	2.42	2.64	4.36	2.91
HH. Cons Real	2.03	2.32	2.48	3.45	2.62
Investment real	1.10	1.46	1.82	12.46	2.98
Exports real	4.37	4.20	4.04	3.88	3.72
Imports real	3.10	3.10	3.02	4.68	2.84

Source: World Bank
GDP components

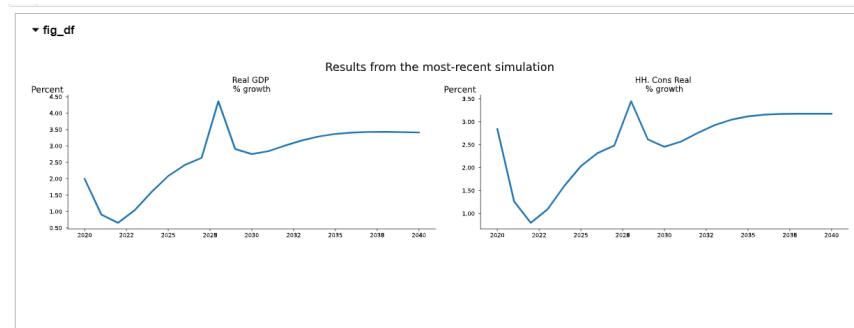
	Real GDP	HH. Cons Real	Real Investment	real Exports	real Imports	real
	--- Percent growth ---					
2022	0.66	0.80	0.70	4.71	3.00	
2023	1.04	1.09	0.63	4.66	2.86	
2024	1.60	1.60	0.79	4.53	2.99	
2025	2.08	2.03	1.10	4.37	3.10	
2026	2.42	2.32	1.46	4.20	3.10	
2027	2.64	2.48	1.82	4.04	3.02	

Source: World Bank

The reportname alone or display(report) will render the report as html (suitable for Jupyter Notebooks or a computer screen. The figure is rendered as an interactive widget the same as a normal figure.

```
display(smallreport)
```

The World Bank's MFMod Framework in Python with Modelflow



GDP components

2025 2026 2027 2028 2029

... Percent growth ...				
Real GDP	2.08	2.42	2.64	4.36
HH. Cons Real	2.03	2.32	2.48	3.45
Investment real	1.10	1.46	1.82	12.46
Exports real	4.37	4.20	4.04	3.88
Imports real	3.10	3.10	3.02	4.68
				2.84

Source: World Bank

GDP components

Real GDP HH. Cons Real Investment real Exports real Imports real

... Percent growth ...				
2022	0.66	0.80	0.70	4.71
2023	1.04	1.09	0.63	4.66
2024	1.60	1.60	0.79	4.53
2025	2.08	2.03	1.10	4.37
2026	2.42	2.32	1.46	4.20
2027	2.64	2.48	1.82	4.04
				3.02

Source: World Bank

The PDF rendering will show each chart separately (like the .show command) or if the samefig=True option is selected (as is the case here) the charts will be organized in a grid.

```
smallreport.pdf()
```

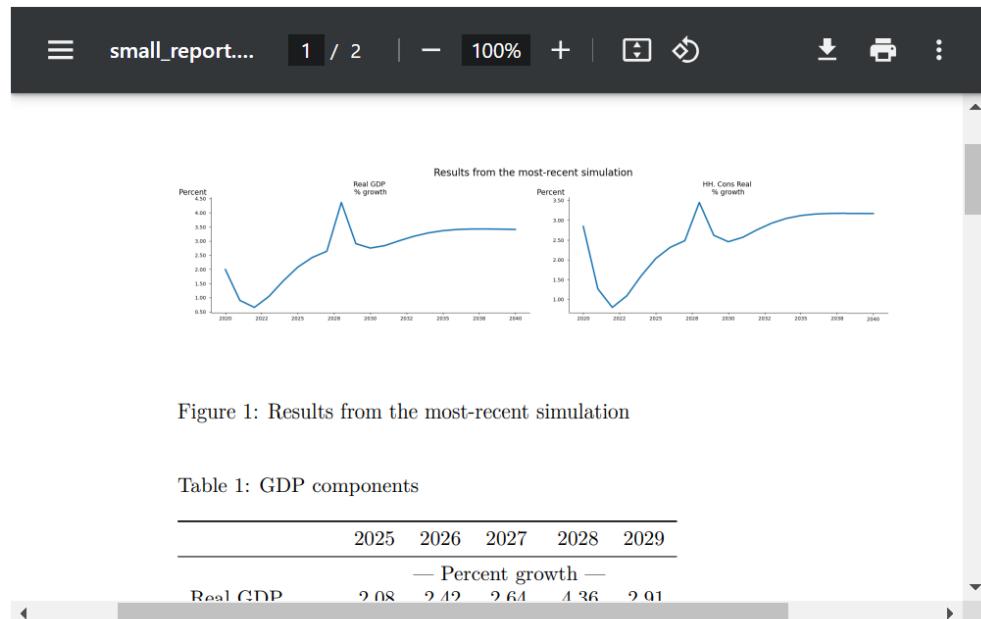


Figure 1: Results from the most-recent simulation

Table 1: GDP components

	2025	2026	2027	2028	2029
— Percent growth —					
Real GDP	2.08	2.42	2.64	4.36	2.91

14.6.3 A report with different text

Unlike a table or a plot, a report can be customized by adding unlimited amounts of text.

In the example below text is entered using three formats – plain text, Latex, and html.

Doing so in this way allows each rendering engine to do its best.

If separate text is not entered between the <html> and </html> tags or between <latex></latex> tags then the html and pdf outputs will use the plain text inputs.

If the text under html or latex tags differs from the plain text, the latex text will be used for pdfs and the html for html outputs.

Below the same content as in the small report but with annotated with text.

```
textreport=(r'''
Text version: Real GDP and household consumption growth under alternative Carbon
↪taxation regimes.

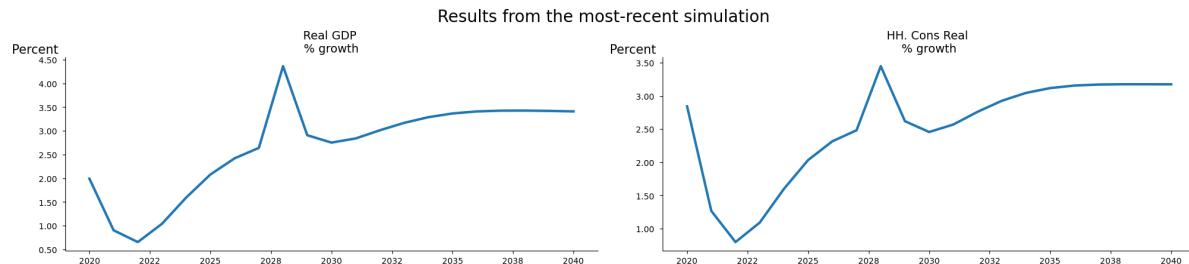
<latex>
\textbf{Text version:} Real GDP and household consumption growth under alternative
↪Carbon taxation regimes.
</latex>

<html>
<h1>Html version:</h1>
<h2> Real GDP and household consumption growth under alternative Carbon taxation
↪regimes.</h2>
</html>
''' +
fig.set_options(samefig=True) +
r'''Real GDP and Expenditure components growth rates following a 1% of GDP injection
↪of foreign investment in 2027
<latex>
\textbf{Real GDP and Expenditure components growth rates following a 1\% of GDP}
↪injection of foreign investment in 2027}
</latex>
<html>
<h1>Real GDP and Expenditure components growth rates following a 1% of GDP injection
↪of foreign investment in 2027</h1>
</html>''' +
tab_t).set_name('report_with_text')
```

When rendered as text, no special formatting of the text is done.

```
textreport.show
```

Text version: Real GDP and household consumption growth under alternative Carbon
↪taxation regimes.



The World Bank's MFMod Framework in Python with Modelflow

Real GDP and Expenditure components growth rates following a 1% of GDP injection
of foreign investment in 2027

GDP components

	Real GDP	HH. Cons	Real Investment	real Exports	real Imports
	--- Percent growth ---				
2022	0.66	0.80	0.70	4.71	3.00
2023	1.04	1.09	0.63	4.66	2.86
2024	1.60	1.60	0.79	4.53	2.99
2025	2.08	2.03	1.10	4.37	3.10
2026	2.42	2.32	1.46	4.20	3.10
2027	2.64	2.48	1.82	4.04	3.02

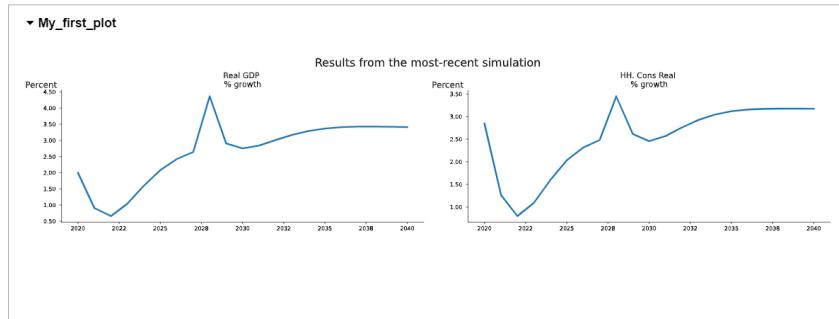
Source: World Bank

When rendered as html, the html text is used and html formatting is rendered.

textreport

Html version:

Real GDP and household consumption growth under alternative Carbon taxation regimes.



Real GDP and Expenditure components growth rates following a 1% of GDP injection of foreign investment in 2027

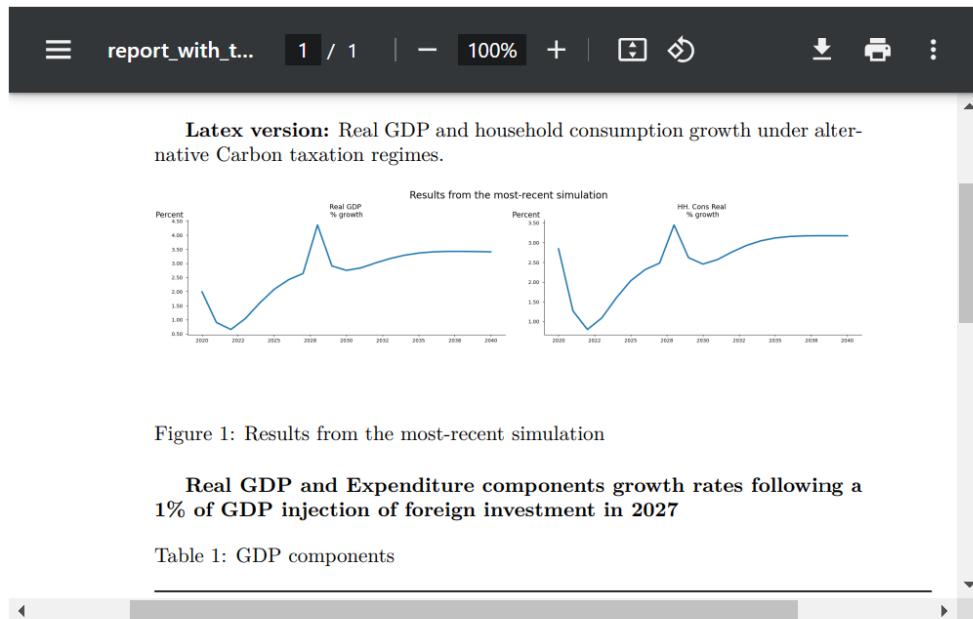
GDP components

	Real GDP	HH. Cons Real	Investment real	Exports real	Imports real
	--- Percent growth ---				
2022	0.66	0.80	0.70	4.71	3.00
2023	1.04	1.09	0.63	4.66	2.86
2024	1.60	1.60	0.79	4.53	2.99
2025	2.08	2.03	1.10	4.37	3.10
2026	2.42	2.32	1.46	4.20	3.10
2027	2.64	2.48	1.82	4.04	3.02

Source: World Bank

Finally when rendered as a pdf, the pdf specific text and formatting is used.

textreport.pdf()



14.7 Storing .reports definitions for later use

The definitions of the different reports can be stored in the `.reports` dictionary that is part of the model object so they can be retrieved and used later.

Notice it is the definition not the content of the table which is stored. So when applied later it is the values of the variables of that model object (notably the `.lastdf` and `.basedf` dataframes of the model object at that point in time that will be rendered.

The `.reports` dictionary is saved when the model is dumped with `.modeldump` and restored when the function `model.load()` is used to load a model.

Note

A `table`, `plot` and `text` objects are each special cases of a `report` and can be stored as well.

14.7.1 The method `.add_report` adds a report to the `.reports` dictionary.

```
mpak.add_report([tab_large, smallreport, textreport])
```

```
Reports added to report repo: large_table, small_report, report_with_text
```

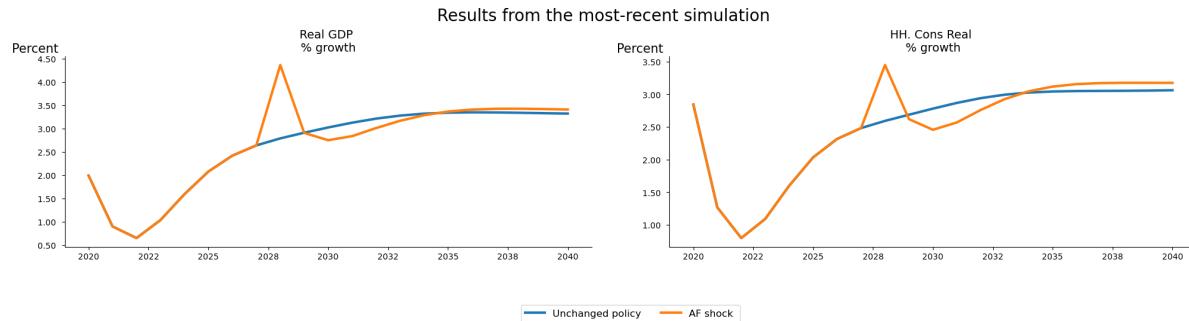
```
for key in mpak.reports:
    print(key);
```

```
large_table
small_report
report_with_text
```

14.7.2 The `.get_report` method creates a report from a stored specification.

A report can be retrieved from the model object and assigned to a variable. Changes to that new report will not be stored unless explicitly saved.

```
newreport=mpak.get_report('small_report').show
```



GDP components					
	2025	2026	2027	2028	2029
--- Percent growth ---					
Real GDP	2.08	2.42	2.64	4.36	2.91
HH. Cons Real	2.03	2.32	2.48	3.45	2.62
Investment real	1.10	1.46	1.82	12.46	2.98
Exports real	4.37	4.20	4.04	3.88	3.72
Imports real	3.10	3.10	3.02	4.68	2.84

Source: World Bank

GDP components

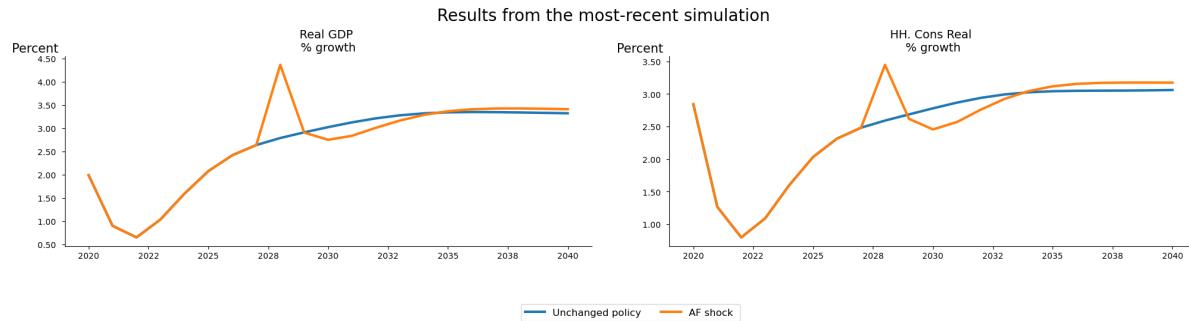
	Real GDP	HH. Cons Real	Investment real	Exports real	Imports real
--- Percent growth ---					
2022	0.66	0.80	0.70	4.71	3.00
2023	1.04	1.09	0.63	4.66	2.86
2024	1.60	1.60	0.79	4.53	2.99
2025	2.08	2.03	1.10	4.37	3.10
2026	2.42	2.32	1.46	4.20	3.10
2027	2.64	2.48	1.82	4.04	3.02

Source: World Bank

14.7.3 Re-using a report on multiple simulation results

In this example the `small_report` is populated with results from three different simulations and the results displayed (see the above listing for names of scenarios in the `kept_scenarios` dictionary).

```
mpak.basedf = mpak.keep_solutions['Baseline']
mpak.lastdf = mpak.keep_solutions['1% of GDP increase in FDI and private investment ↳ (AF shock)']
mpak.smpl(2025,2029);
firstreport=mpak.get_report('small_report').set_name('firstreport').show
```



GDP components

	2025	2026	2027	2028	2029
	--- Percent growth ---				
Real GDP	2.08	2.42	2.64	4.36	2.91
HH. Cons Real	2.03	2.32	2.48	3.45	2.62
Investment real	1.10	1.46	1.82	12.46	2.98
Exports real	4.37	4.20	4.04	3.88	3.72
Imports real	3.10	3.10	3.02	4.68	2.84

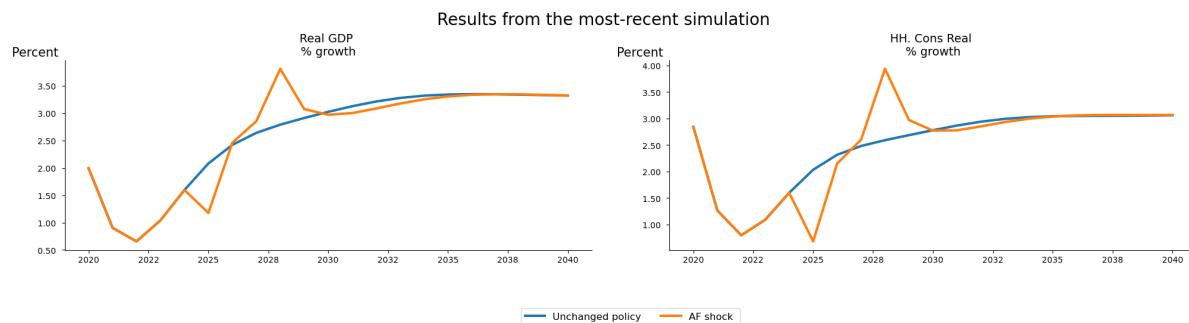
Source: World Bank

GDP components

	Real GDP	HH. Cons Real	Real Investment	real Exports	real Imports	real
	--- Percent growth ---					
2022	0.66	0.80	0.70	4.71	3.00	
2023	1.04	1.09	0.63	4.66	2.86	
2024	1.60	1.60	0.79	4.53	2.99	
2025	2.08	2.03	1.10	4.37	3.10	
2026	2.42	2.32	1.46	4.20	3.10	
2027	2.64	2.48	1.82	4.04	3.02	

Source: World Bank

```
mpak.basedf = mpak.keep_solutions['Baseline']
mpak.lastdf = mpak.keep_solutions['$25 increase in oil prices 2025-27']
mpak.smpl(2025,2029);
secondreport=mpak.get_report('small_report').set_name('secondreport').show
```



GDP components

	2025	2026	2027	2028	2029
	--- Percent growth ---				
Real GDP	1.18	2.46	2.85	3.81	3.08

(continues on next page)

(continued from previous page)

HH. Cons Real	0.68	2.15	2.60	3.94	2.97
Investment real	1.23	0.99	1.62	2.02	2.93
Exports real	4.36	4.21	4.06	3.92	3.78
Imports real	1.57	1.88	2.45	3.98	3.74

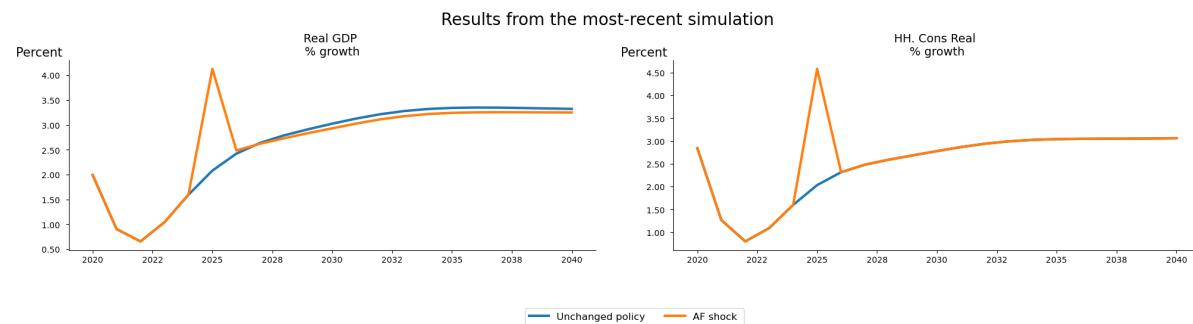
Source: World Bank

GDP components

	Real GDP	HH. Cons Real	Investment real	Exports real	Imports real
--- Percent growth ---					
2022	0.66	0.80	0.70	4.71	3.00
2023	1.04	1.09	0.63	4.66	2.86
2024	1.60	1.60	0.79	4.53	2.99
2025	1.18	0.68	1.23	4.36	1.57
2026	2.46	2.15	0.99	4.21	1.88
2027	2.85	2.60	1.62	4.06	2.45

Source: World Bank

```
mpak.basedf = mpak.keep_solutions['Baseline']
mpak.lastdf = mpak.keep_solutions['2.5% increase in C 2025-27 -- exog whole period --
KG=True']
mpak.smpl(2025,2029);
thirdreport=mpak.get_report('small_report').set_name('thirdreport').show
```



GDP components

Real GDP	4.13	2.49	2.62	2.73	2.84
HH. Cons Real	4.58	2.32	2.48	2.59	2.69
Investment real	2.50	2.12	2.23	2.45	2.69
Exports real	4.35	4.14	3.96	3.80	3.66
Imports real	5.44	3.27	3.18	3.08	2.99

Source: World Bank

GDP components

	Real GDP	HH. Cons Real	Investment real	Exports real	Imports real
--- Percent growth ---					
2022	0.66	0.80	0.70	4.71	3.00
2023	1.04	1.09	0.63	4.66	2.86
2024	1.60	1.60	0.79	4.53	2.99
2025	4.13	4.58	2.50	4.35	5.44
2026	2.49	2.32	2.12	4.14	3.27
2027	2.62	2.48	2.23	3.96	3.18

Source: World Bank

14.7.4 Reports are stored in .pcim files, and restored when a model is loaded

Once a report has been added to the model object it will be saved with the model object.

Below we save the current state of mpak, inclusive of the reports that were added to the model object. So first dump the model and data:

```
mpak.modeldump('testpak', keep=True)
```

Next a new model object is declared tempmodel, and the state and data from the saved mpak are read into that object.

```
xpak, baseline = model.modelload('testpak', run=True )
```

Zipped file read: testpak.pcim

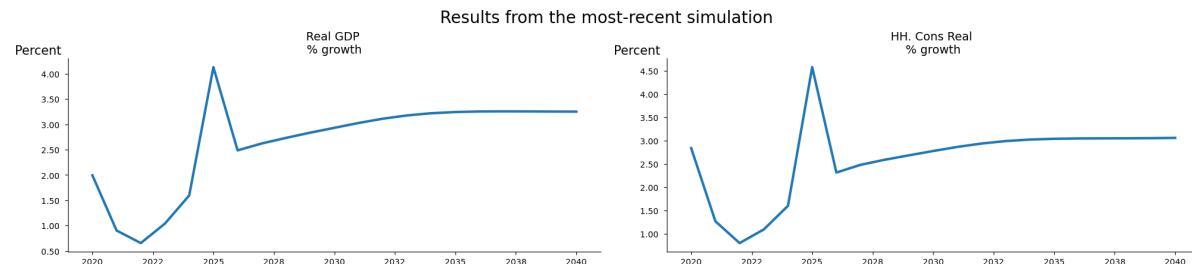
Next the .basedf and .lastdf dataframes are set equal to the data frames from the kept Baseline and investment shock scenarios.

```
xpak.basedf = xpak.keep_solutions['Baseline']
xpak.lastdf = xpak.keep_solutions['1% of GDP increase in FDI and private investment ↪ (AF shock)']
```

With that, the reports can be run (and should show precisely the same results).

```
trep=xpak.get_report('report_with_text')
trep.show
```

Text version: Real GDP and household consumption growth under alternative Carbon-taxation regimes.



Real GDP and Expenditure components growth rates following a 1% of GDP injection of foreign investment in 2027

GDP components

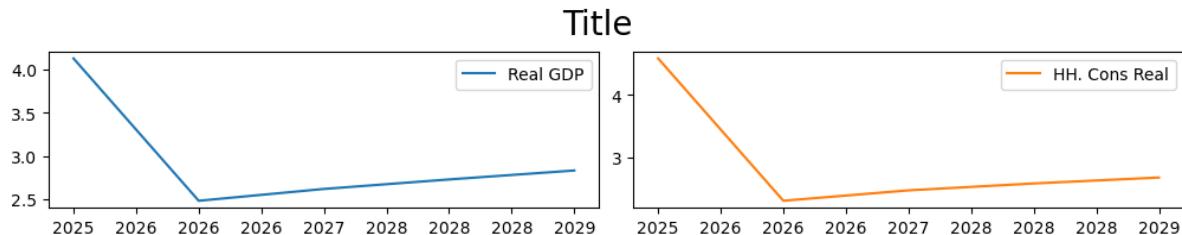
	Real GDP	HH. Cons	Real Investment	real Exports	real Imports	real
	--- Percent growth ---					
2022	0.66	0.80	0.70	4.71	3.00	
2023	1.04	1.09	0.63	4.66	2.86	
2024	1.60	1.60	0.79	4.53	2.99	
2025	4.13	4.58	2.50	4.35	5.44	
2026	2.49	2.32	2.12	4.14	3.27	
2027	2.62	2.48	2.23	3.96	3.18	

Source: World Bank

14.8 [].rtable and [].rplot - reports from []

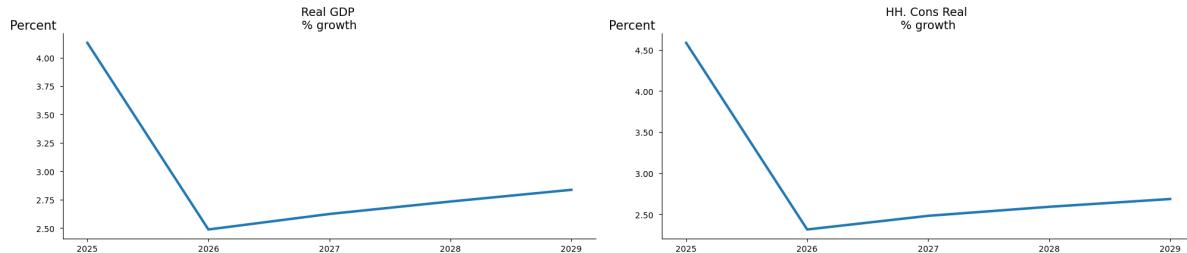
Report tables and plots can also be created directly from the variable selector `[]`. The purpose is to enable easy reuse of variable selection like this.

```
mpak ['PAKNYGDPMKTPKN PAKNECONPRVTKN'].growth.rename().plot();
```



To a plot like this:

```
mpak ['PAKNYGDPMKTPKN PAKNECONPRVTKN'].rplot(samefig=1).show
```



Or a table like this:

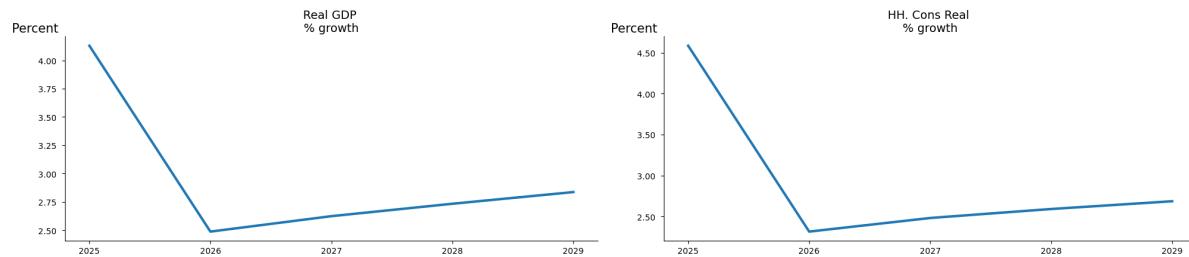
```
mpak ['PAKNYGDPMKTPKN PAKNECONPRVTKN'].rtable(samefig=1).show
```

Table					
	2025	2026	2027	2028	2029
--- Percent growth ---					
Real GDP	4.13	2.49	2.62	2.73	2.84
HH. Cons Real	4.58	2.32	2.48	2.59	2.69

The `[].rtable` and `[].rplot` are just wrappers around the `model instance.plot` and `model instance.table` so the arguments - except the variable selection - are the same, and the returned values can be used just the returned values of `model instance.plot` and `model instance.table`. So `+ and |` can also be used.

```
(mpak.text('Growth in {city_name}') +
mpak ['PAKNYGDPMKTPKN PAKNECONPRVTKN'].rplot(samefig=1) +
mpak ['PAKNYGDPMKTPKN PAKNECONPRVTKN'].rtable()).show
```

Growth in Pakistan



Table

	2025	2026	2027	2028	2029
--- Percent growth ---					
Real GDP	4.13	2.49	2.62	2.73	2.84
HH. Cons Real	4.58	2.32	2.48	2.59	2.69

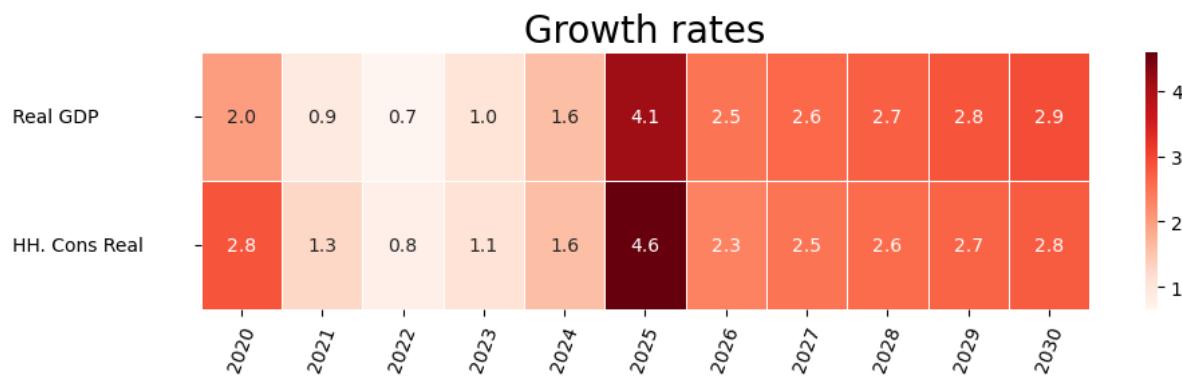
14.9 Some other supported outputs

Below are an illustration of some charts and table features not discussed above, which may be of interest in some contexts:

14.9.1 Heatmaps

For some model types heatmaps can be helpful, and they come out of the box.

```
with mpak.set_smpf(2020, 2030):
    mpak[['PAKNYGDPMKTBNK PAKNECONPRVTBNK']].pct.rename().heat(title='Growth rates',
    ↴ annot=True, dec=1, size=(10, 3))
```



Part V

Model Analytics

MODEL STRUCTURE AND CAUSAL CHAINS

A model has a well defined logical and causal structure. Kogiku(1968) provides an introduction to causal analysis of models, while Berndsen(1995) gives a more elaborate discussion.

At the simplest level, the equations of a model can be organized into blocks.

- **Simultaneous block** include equations that have are co-determined simultaneously. They contain feedback loops that may require several iterations before a solution that satisfies them all is found. A classic simultaneous block would include GDP, Income and Consumption. Consumption depends on income. Income depends on GDP, but Consumption also determines GDP.
- **Recursive blocks** include equations that are a simple function of other variables. For example, the current account balance is just the difference between Export Revenues and Import Revenues. These can be solved with just one pass once the values of the simultaneous blocks have been resolved.

At the equation level, each endogenous variable is a function of one or more variables, but because some of these variables are also dependent on other variables in the model, those right hand side variables that are endogenous can have their equations substituted into the first level equation to get an extended set of dependencies. Moreover, the endogenous right hand side variables of these second level variables can also have their right hand sides substituted into the equation etc.

ModelFlow uses the [networkx](https://networkx.org/) (<https://networkx.org/>) python package to analyze the interrelationships within the model and between equations and includes a number of methods and properties to present these interrelationships both in tabular and graphical form¹, a subset of which is exposed in this chapter.

15.1 Setting up the python environment and loading a pre-existing model

In this chapter - Model structure and causal chains

This chapter introduces tools and techniques for analyzing the structure and behavior of models in the ModelFlow framework. In a system of equations like World Bank models, impacts on variables in a simulation can be the results of the direct effect from changes imposed on variables in the equation of a variable, but also indirect effects caused by changes in variables that are not in the equation of a specific variable, but do generate changes in variables that are in the equation.

The chapter presents tools that explore the structure of a model, and display both the direct and indirect determinants of given equations.

Following a simulation, the routines presented can display the precise contribution of a given variable to the change in another.

¹ The relational graphs produced by ModelFlow use the Graphviz <https://graphviz.org/> program, and are based on the relationships determined by the Networkx package.

In addition to displaying which variables impacted a given variable, the routines can also be used to understand which variables are most impacted by changes in a given variable.

Results can be displayed both graphically and in tabular form.

```
mpak,baseline = model.modelload('../models/pak.pcim',alfa=0.7,run=1)
mpak.model_description="World Bank climate aware model of Pakistan as described in
→Burns et al. (2019)"
```

Zipped file read: ..\models\pak.pcim

```
latex=True # Enables the charts in latex
```

Note

latex=True

The default behavior when displaying graphs in a *jupyter notebook* is to produce images in .svg format. These images scale well and the mouseover feature can be used. That is: On mouseover of a node, the variable and the equation are displayed. On mouseover on a joining line, the extent to which the variable contributed to the change in the dependent variable is displayed.

Unfortunately this *jupyter book* (book – not notebook) requires images be in the jpg or PNG format. To enable generation of PDFs the `latex=True` option is set, which forces ModelFlow routines to render graphics in the PNG format – effectively disabling the features of svg.

If using the Notebooks that for this book, the variable `latex` could be set equal to `False` in which case the same code will generate graphics in the more versatile SVG format.

15.2 Model information

As noted before, the model object contains information about the model itself, its name, its structure (does it contain simultaneous equations or is it recursive), the number of variables it contains and the number that are exogenous and endogenous (have associated equations).

```
mpak
```

```
<
Model name : PAK
Model structure : Simultaneous
Number of variables : 839
Number of exogeneous variables : 461
Number of endogeneous variables : 378
>
```

15.3 Model structure

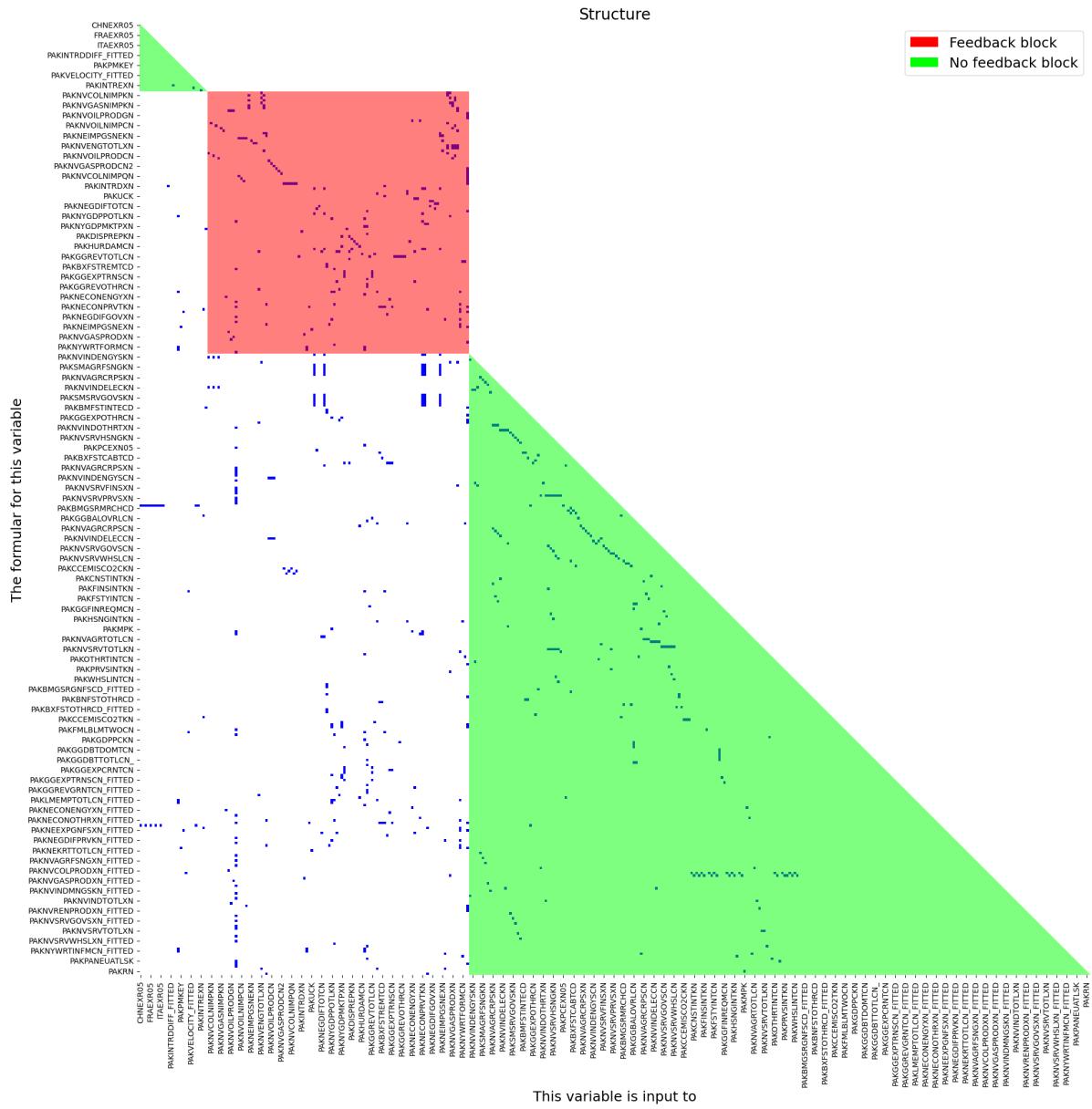
A quick way to visualize the structure of a model is to plot its adjacency matrix (https://en.wikipedia.org/wiki/Adjacency_matrix) (https://en.wikipedia.org/wiki/Adjacency_matrix).

The adjacency matrix plots the relationships between endogenous variables in the model, dividing them into one or more simultaneous blocks and one or more recursive blocks.

Below is the adjacency matrix for the Pakistan model. Variables in the red square block (simultaneous block) depend on one or more variables that in turn depend upon them, requiring the model to solve for their values simultaneously. The variables in the green triangles (recursive blocks) do not enter directly or indirectly as an argument in the variables that determine them and therefore can be solved in one iteration once the values for the simultaneous variables are determined.

Internally ModelFlow takes these factors into consideration and solves the simultaneous block(s) first and then solves the recursive blocks.

```
mpak.plotadjacency(size=(20,20));
```



As is evident in the above chart, the majority of the variables in MFMod Pakistan are recursive (green) and depend simply on the values of other variables. The core of the model lies in the simultaneous (red) block, where the main wages, prices, real and nominal variables that drive other variables are determined.

15.4 The dependencies of individual endogenous variables (the `.tracepre()` method)

As noted above, every endogenous variables is directly dependent on the variables that occur on its right hand side, but is also indirectly dependent on the variables that determine its RHS variables and in turn those that determine the variables to the right of them *ad infinitum*.

ModelFlow includes several methods and properties that allow these dependencies to be explored.

The `.fml` property returns the normalized formula of an equation, from which the right hand variables for the equation

The World Bank's MFMod Framework in Python with Modelflow

can be discerned and these are reported along with their descriptions following the formula.

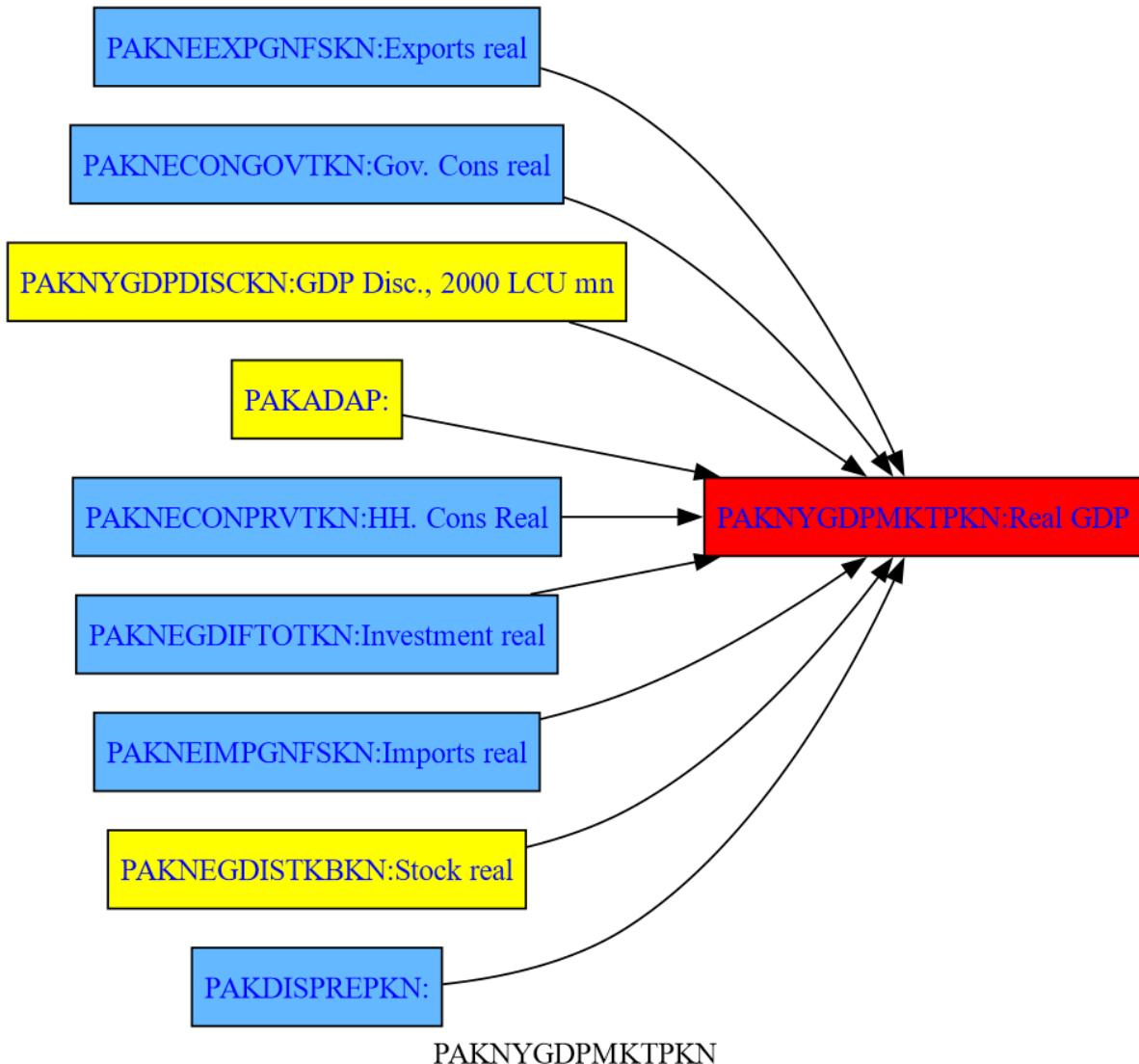
```
mpak.PAKNECONPRVTKN.frml
```

```
Endogeneous: PAKNECONPRVTKN: HH. Cons Real
Formular: FRML <DAMP,STOC> PAKNECONPRVTKN = (PAKNECONPRVTKN(-1)*EXP (PAKNECONPRVTKN_
↪A+ (-0.2*(LOG(PAKNECONPRVTKN(-1))-LOG(1.21203101101442)-LOG(((PAKBXFSTREMTCD(-
↪1)-PAKBMFSTREMTCD(-1))*PAKPANUSATLS(-1))+PAKGEXPTRNSCN(-1)+PAKNYYWBTOTLCN(-
↪1)*(1-PAKGREVDRCTXN(-1)/100))/PAKNECONPRVTXN(-1)))+0.
↪763938860758873*((LOG(((PAKBXFSTREMTCD-
↪PAKBMFSTREMTCD)*PAKPANUSATLS)+PAKGEXPTRNSCN+PAKNYYWBTOTLCN*(1-PAKGREVDRCTXN/
↪100))/PAKNECONPRVTXN))-(LOG(((PAKBXFSTREMTCD(-1)-PAKBMFSTREMTCD(-
↪1))*PAKPANUSATLS(-1))+PAKGEXPTRNSCN(-1)+PAKNYYWBTOTLCN(-1)*(1-PAKGREVDRCTXN(-
↪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
PAKGEXPTRNSCN : Current Transfers
PAKGREVDRCTXN : Direct Revenue Tax Rate
PAKNECONPRVTKN_A: Add factor:HH. Cons Real
PAKNECONPRVTKN_D: Fix dummy:HH. Cons Real
PAKNECONPRVTKN_X: Fix value:HH. Cons Real
PAKNECONPRVTXN : Implicit LCU defl., Pvt. Cons., 2000 = 1
PAKNYYWBTOTLCN : Total Wage Bill
PAKPANUSATLS : Exchange rate LCU / US$ - Pakistan
```

The method `.tracepre()` provides a graphical representation of this relationship, showing all the variables that directly determine an endogenous variable (in this example real GDP), distinguishing between RHS variables that are endogenous (in blue) and those that are exogenous (yellow).

```
mpak.PAKNYGDPMKTPKN.tracepre(png=latex,size=(4,3))
```



If the model has been solved, `.tracepre()` goes one step further and uses the thickness of the lines to reflect the relative importance of each variable in the change of the dependent variable in the preceding scenario.

15.4.1 Shock the model

Below a \$30 nominal Carbon tax is applied beginning in 2025.

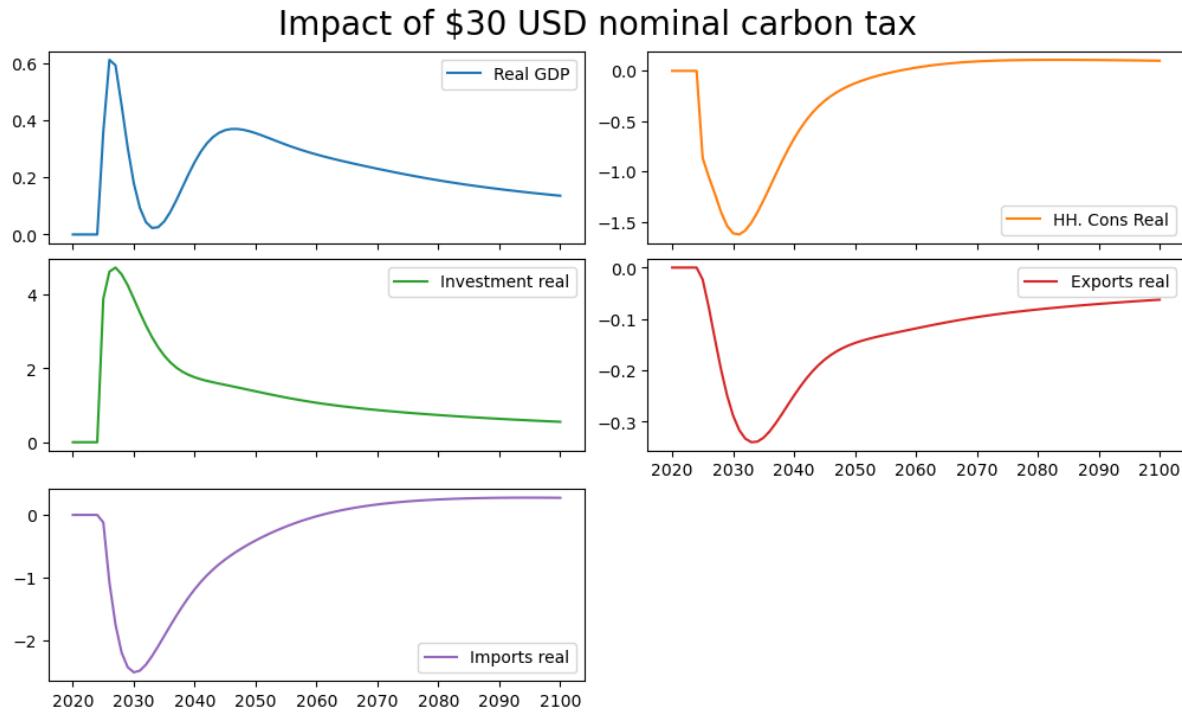
```

alternative = baseline.upd("<2025 2100> PAKGGREVC02CER PAKGGREVC02GER"
                           "PAKGGREVC02OER = 30")
result = mpak(alternative, 2020, 2100) # simulates the model
    
```

As a result GDP, consumption investment and most all variables in the model change, as illustrated in the below graphs that show the percent deviation of the main components of GDP from their baseline values.

```

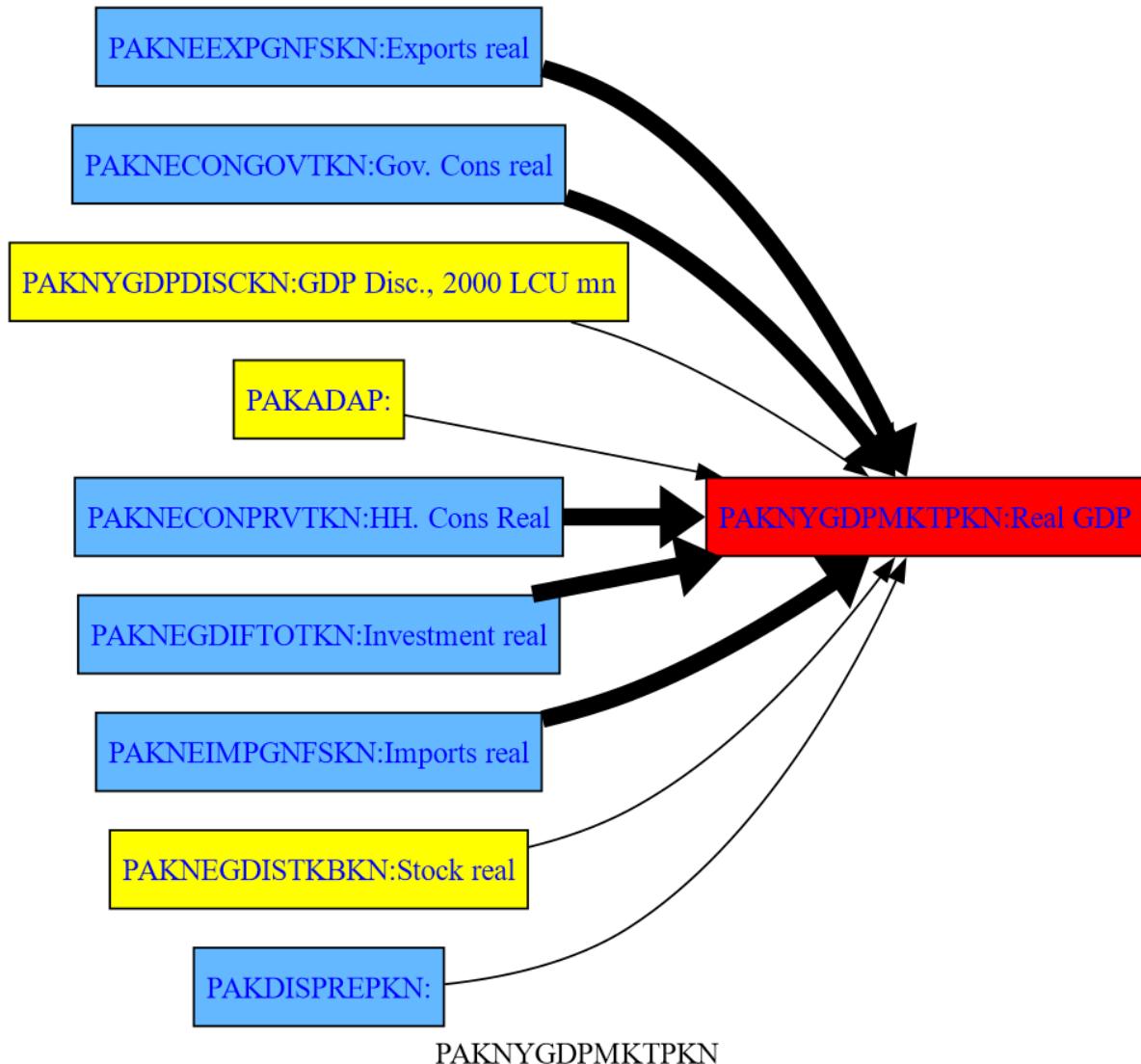
mpak['PAKNYGDPMKTPKN PAKNECONPRVTKN PAKNEGDIFTOTKN PAKNEEXPGNFSKN PAKNEIMPGNFSKN'].difpctlevel.rename().plot(title="Impact of $30 USD nominal carbon tax");
    
```



15.4.2 .tracepre() following a shock

Below the same `.tracepre()` command is executed again, but because a shock has been simulated, the width of the lines representing the causal links between variables is thicker the more important the change in a given variable was in the previous simulation in explaining the change in the level of the dependent variable (GDP).

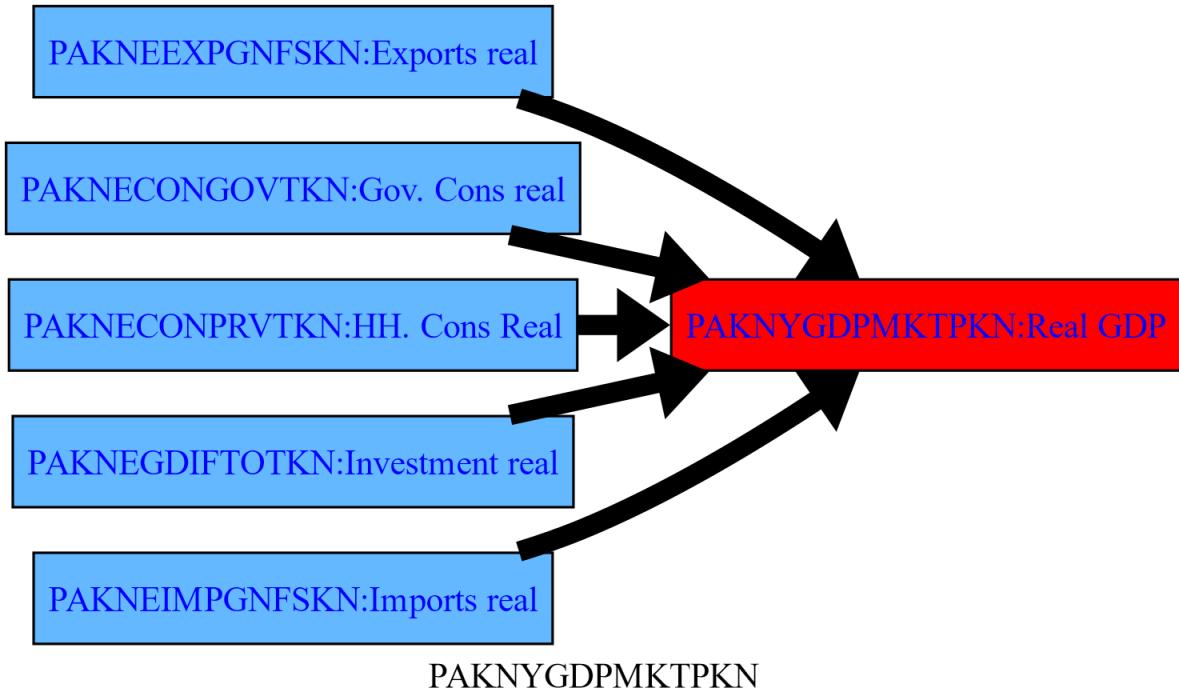
```
latex=True
mpak.PAKNYGDPMKTPKN.tracepre(png=latex, size=(5, 3))
```



15.4.3 The filter option, restricts the output of .tracepre()

The filter option can be used to restrict the output of `.tracepre()` to RHS variables that have had a large impact on the dependent variable. In the example below, the option `filter=20` instructs `tracepre` to only draw those rhs variables that contributed 20 percent or more to the total change in GDP.

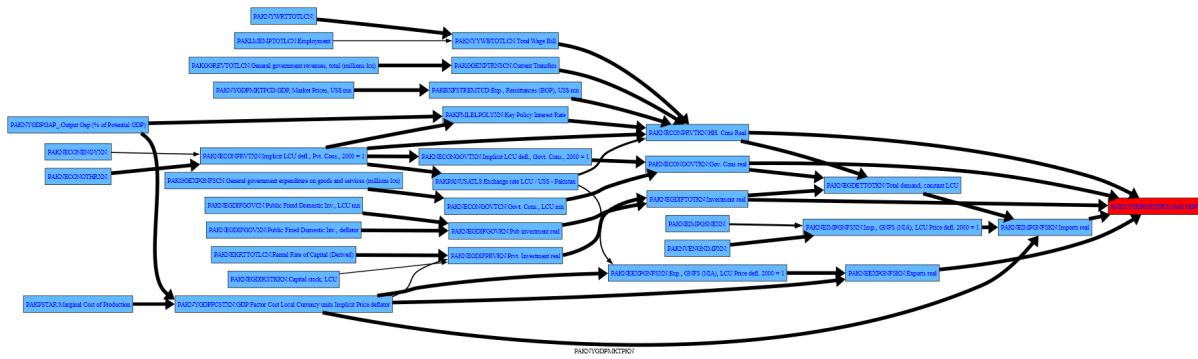
```
mpak.PAKNYGDPMKTPKN.tracepre(filter=20, png=latex, size=(5, 3))
```



15.4.4 The up option, extends the .tracepre plot beyond the first level of causal variables

The up option allows .tracepre dependencies to be followed beyond the first level of causal variables. Below, it is extended to variables as much as three levels back, and restricted to those whose variation explains at least 20 percent of the change in the variable of which they are a right-hand-side variable.

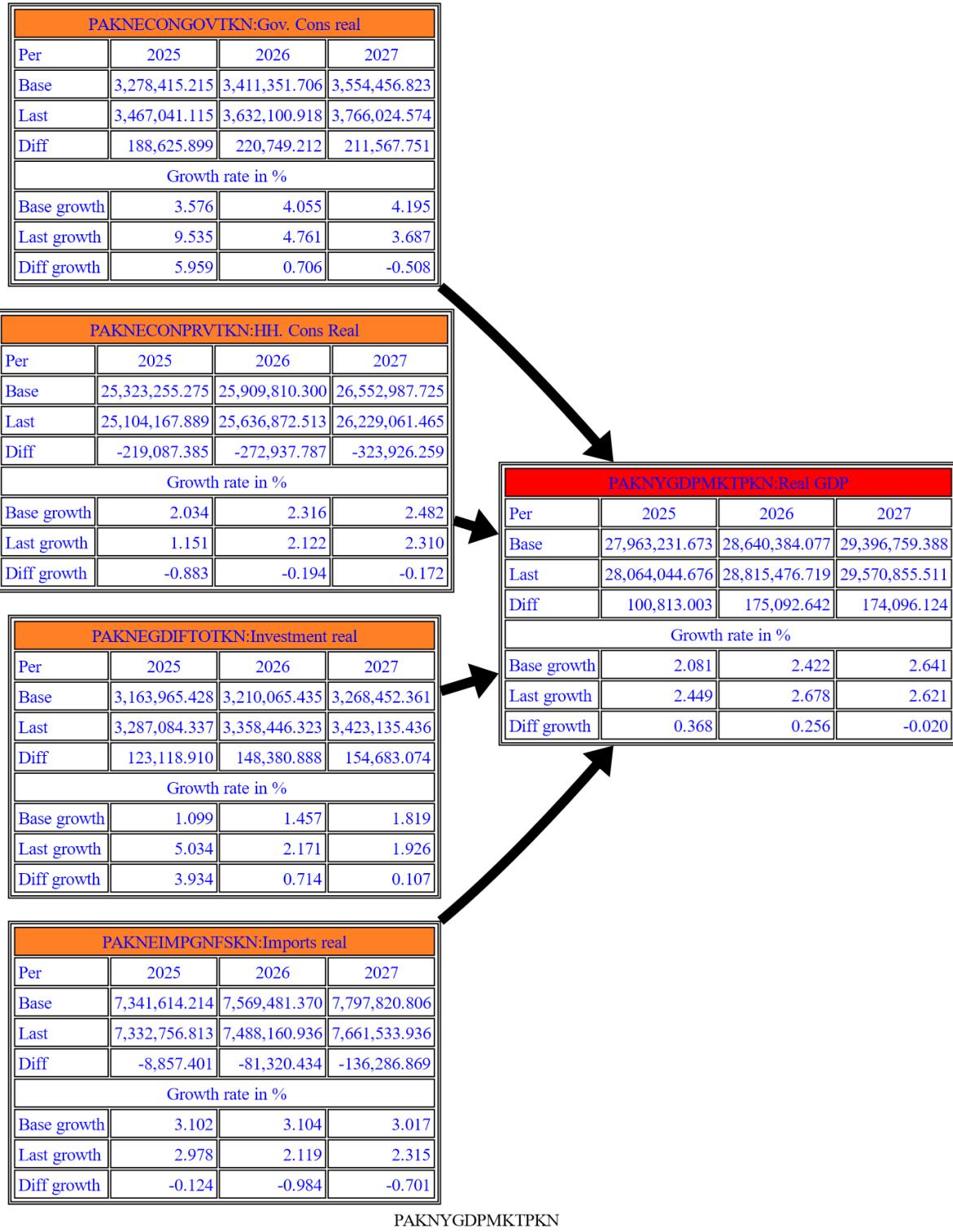
```
mpak.PAKNYGDPMKTPKN.tracepre(filter = 20, up=3, png=latex)
```



Adding a table to the causal graph

The Fokus2 option causes a table of values to be added to the causal flow graph. In this example, the showgrowth=True option instructs ModelFlow to show the table in both level and growth rate terms.

```
with mpak.set_smpl(2025,2027):
    print(mpak.PAKNYGDPMKTPKN.tracepre(filter = 20,
                                          fokus2='PAKNEGDIFFOTKN PAKNECONPRVTKN PAKNECONGOVTKN PAKNYGDPMKTPKN_
→PAKNEIMPGNFSKN',
                                          growthshow=True,
                                          png=latex)
        )
```

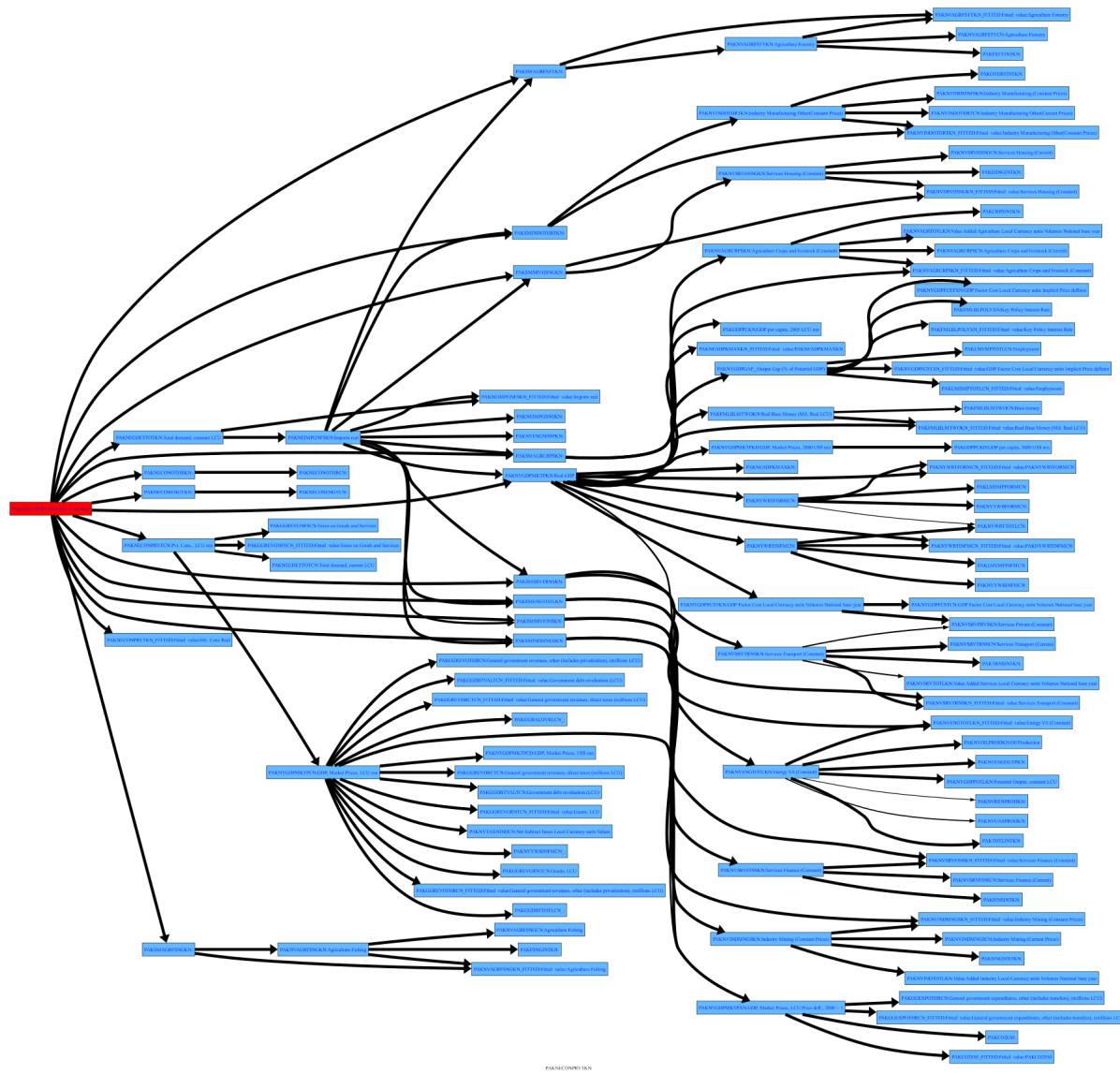


None

15.4.5 .tracedep (down=xx) traces the impact of a variable on other variables

The preceding examples have focused on understanding how changes in other variables have impacted the variable of interest. The closely related `tracedep()` method shows what other variables depend on the specified variable, with the `down=xx` option indicating how many levels of substitution to display. Here, the direction of the dependency graph is reversed and the chart shows the impact that the changes in the selected variable had on those which depend upon it. Below is the impact of change in consumption on all of the variables up to 3 levels below the consumption equation.

```
mpak.PAKNECONPRVTKN.tracedep(down=3,filter=20,png=latex)
```

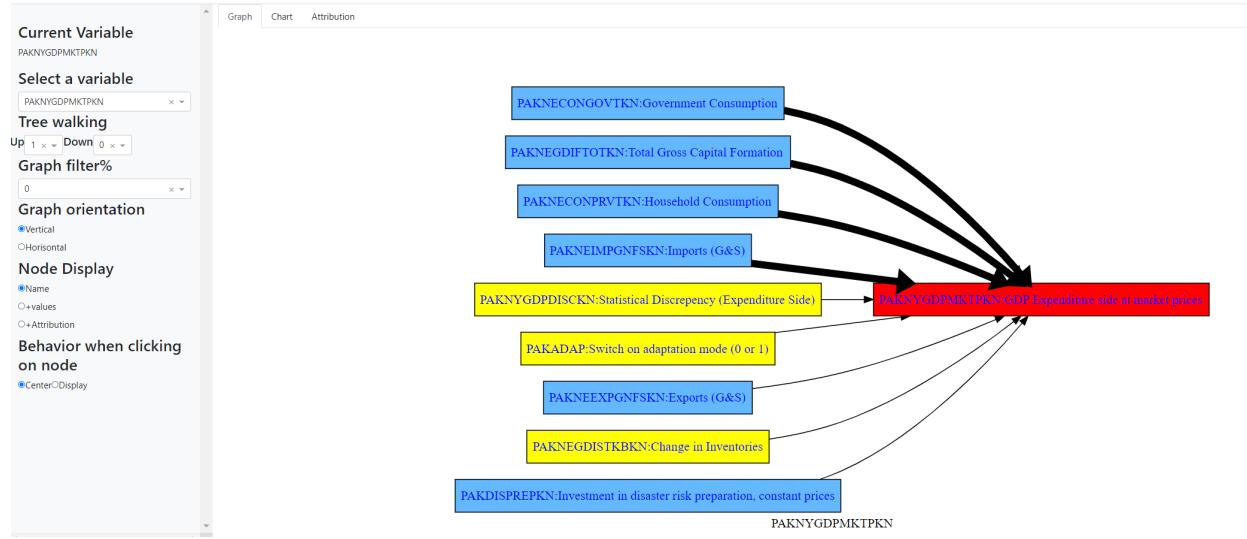


15.4.6 .modeldash() An interactive way to explore dependencies

The `.modeldash()` method generates a widget that allows you to dynamically adjust the arguments to the `tracepre()` and `tracedep` functions.

```
with mpak.set_smpl(2022, 2026):
    mpak.modeldash('PAKNYGDPMKTPKN', jupyter=True, inline=False)
```

The above commands generate a dashboard that looks like the below, where the panel to the left allows the user to change options including the filter, the depth of the trace among other things.



CHAPTER
SIXTEEN

ANALYZING THE IMPACT OF A SHOCK

When working with a model, it is often useful to have a quantitative sense of the contribution of different channels to a final result. For example, an increase in interest rates will tend to reduce investment and consumer demand – contributing to a reduction in GDP. At the same time, lower inflation as the higher interest rate takes effect will tend to work in the opposite direction.

The `tracedep()` and `tracepre()` methods introduced in the previous chapter give a sense of impacts. The `ModelFlow` methods `.dekomp()` and `.totdif()` take that one step further by calculating the contribution of each channel to the overall result.

In this chapter - Analyzing the Impact of a Shock

This chapter provides the tools and techniques to dissect and understand the ripple effects of shocks within macroeconomic systems.

This chapter focuses on methods to analyze the impacts of external shocks within a macroeconomic model. The previous chapter explored the impacts of changes in one variable on other variables in a model by following the causal chain of any individual variable.

The techniques explored in this chapter illustrate and extend this by:

- attributing changes in any given model variable to the changes in the variables in the model that were shocked.
In the techniques used in the previous chapter it may be determined that increased inflation was the proximate cause of a decline in consumption, the methods presented here seek to illuminate which of the changes to exogenous variables caused the increase in inflation that caused the decline in consumption (say an increase in oil prices).
- Impacts can be traced either through a single equation method or using a model level decomposition

Examples include a range of graphical and textual visualization of results.

```
# set default precision
pd.set_option("display.precision", 2)
```

```
# For display reasons
latex = 1
```

16.1 Load the existing model, data and descriptions

The file `pak.pcim` contains a dump of model equations, dataframe, simulation options and variable descriptions for the World Bank climate aware model for Pakistan described in Burns *et al.* (2021).

The code below:

- Loads the model and simulates it using the `DataFrame` stored in the `pcim` file to establish a baseline.
- Creates a `DataFrame` that is a copy (from 2020) of the active solution in `mpak` and changes the tax rate to 30 USD/Ton for carbon emissions from coal, oil and natural gas.
- Runs a simulations with these new carbon taxes.

The results from this simulation will be used below to explore the attribution functionality of ModelFlow.

Single equation Impact Decomposition

```
mpak,baseline = model.modelload('..models/pak.pcim',alfa=0.7,run=1,relconv=0.
+0000000001 ,keep='Business as Usual')
alternative = baseline.upd("<2020 2100> PAKGGREVCO2CER PAKGGREVCO2GER_
+PAKGGREVCO2OER = 30")
```

Zipped file read: ..\models\pak.pcim

```
#simulate the model
result = mpak(alternative,2025,2100,keep='Nominal carbon tax of 30 USD',
              ljit=False,           # do not compile the model (default value for this_
+option)
              nfirst=800,
              maxiteration=100    # if no convergence after 100 iterations - stop
            ) # simulates the model
```

16.2 The mathematics of decomposition

At its root the idea of attribution is to take the total derivative of the model to identify the sensitivity of the equation of interest to changes elsewhere in the model and then combine that with the changes in other variables.

Take a variable y that is a function of two other variables a and b . In the model, the relationship might be written as:

$$y = f(a, b)$$

If there are two sets of results designated with a subscript 0 and 1, these can be written as:

$$y_0 = f(a_0, b_0) \quad (16.1)$$

$$y_1 = f(a_1, b_1) \quad (16.2)$$

If the change in the three variables is specified as $\Delta y, \Delta a, \Delta b$, the total derivative of y can be written as:

$$\Delta y = \underbrace{\Delta a \frac{\partial f}{\partial a}(a, b)}_{\Omega_a} + \underbrace{\Delta b \frac{\partial f}{\partial b}(a, b)}_{\Omega_b} + \text{Residual}$$

The first expression can be called Ω_a or the contribution of changes in a to changes in y , and the second Ω_b , or the contribution of changes in b to changes in y .

ModelFlow performs a numerical approximation of Ω_a and Ω_b by performing two runs of the $f()$:

$$y_0 = f(a_0, b_0) \quad (16.3)$$

$$y_1 = f(a_0 + \Delta a, b_0 + \Delta b) \quad (16.4)$$

and calculates Ω_a and Ω_b as:

$$\Omega a = f(a_1, b_1) - f(a_1 - \Delta a, b_1) \quad (16.5)$$

$$\Omega b = f(a_1, b_1) - f(a_1, b_1 - \Delta b) \quad (16.6)$$

And:

$$residual = \Omega a + \Omega b - (y_1 - y_0) \quad (16.7)$$

If the model is fairly linear, the residual will be small.

16.3 Model-level decomposition or single equation decomposition?

Above, the relationship between y , a , and b was summarized by the function $f()$.

$f(a, b)$ could represent **a single equation** in the model or it could represent **the entire model**.

In the **single equation** mode, Δa and Δb would be treated as exogenous variables in the attribution calculation as they are both on the right hand side of the equation (i.e. exogenous to this equation – even if they might be endogenous variables in some other equation). Here results will show the direct impact of changes in the RHS variable(s) on the LHS variable.

When analyzing the total derivative for the **entire model** instance, a and b will be purely exogenous variables. In this case, the decomposition shows the cumulative effect – potentially operating through multiple channels of a change in the exogenous (or exogenized variables) on different endogenous variables in the model. Say we are looking at inflation, an exogenous change in wages would influence prices directly through higher costs of production and indirectly by inducing higher demand. The model-level decompoistion will return the sum of the two or more influences.

Assume the simple equation example such that a and b are simple variables. When Δy , Δa and Δb reflect the difference across scenarios (say the value of the three variables in `.lastdf` less the value in `.basedf`) then;

Ω_a , Ω_b are the absolute contribution of a and b to the change in y , and $100 * \left[\frac{\Omega_a}{\Delta y} \right]$ is the share of the change in y

explained by expressed as a percent and $100 * \left[\frac{\Omega_b}{\Delta y} \right]$ is the share of the change in y explained by b expressed as a percent.

If Δy , Δa and Δb are the changes over time ($\Delta y_t = y_t - y_{t-1}$), then Ω_a , Ω_b are the contributions of a and b to the rate of growth of y , while $100 * \left[\frac{\Omega_a}{\Delta y_{t-1}} \right]$ $100 * \left[\frac{\Omega_b}{\Delta y_{t-1}} \right]$ are are the contributions of a and b to the rate of growth of y .

16.4 Decomposing the source of changes to a single endogenous variable

The ModelFlow method `.dekomp()` is used to calculate the contribution of RHS variables to the change in an endogenous (LHS) variable.

This method takes advantage of the fact that the model object stores the initial and most recent simulation result in two dataframes called `.basedf` and `.lastdf`, as well as all of the equations of the model.

The `decomp()` method calculates the contribution to changes in the level of the dependent variable in a given equation. It does not calculate what caused the changes to the RHS variables.

In the example below, the contribution to the change in Total emissions is decomposed into the contribution from each of three sources in the model, the consumption of Crude Oil, Natural Gas and Coal. As the equation for total emissions is just the sum of the three this is a fairly trivial decomposition, but it provides an easily understood illustration of the process at work.

Note that, initially some carbon taxes were negative because the associated energy products benefited from some sort of subsidy. As a result, although each carbon tax is set to 30 in the simulation that was run earlier, the change in the levels of the Carbon tax is different across carbon taxes, with the increase in the net taxation on the carbon emissions from natural gas being particularly large.

```
print("Change in carbon taxes:")
with mpak.set_smp1(2023, 2030):
    print(mpak['PAKGGREVC02CER PAKGGREVC02GER PAKGGREVC02OER'].dif.rename().df)
```

```
Change in carbon taxes:
    Carbon tax on coal (USD/t)   Carbon tax on gas (USD/t)  \
2023            35.55                71.0
2024            35.55                71.0
2025            35.55                71.0
2026            35.55                71.0
2027            35.55                71.0
2028            35.55                71.0
2029            35.55                71.0
2030            35.55                71.0
    Carbon tax on oil (USD/t)
2023            38.71
2024            38.71
2025            38.71
2026            38.71
2027            38.71
2028            38.71
2029            38.71
2030            38.71
```

```
dekomp_result = mpak.PAKCEMISCO2TKN.dekomp(start=2024, end=2027);
```

```

Formula      : FRML <IDENT> PAKCCEMISCO2TKN =_
  ↵PAKCCEMISCO2CKN+PAKCCEMISCO2OKN+PAKCCEMISCO2GKN $
  2024        2025        2026        2027
Variable    lag
Base        0   230370296.36 236240661.98 242537408.00 248995663.61
Alternative 0   230370296.36 183872963.10 190234954.74 196979009.56
Difference   0       0.00 -52367698.88 -52302453.26 -52016654.05
Percent     0      -0.00     -22.17     -21.56     -20.89
Contributions to difference for PAKCCEMISCO2TKN
  2024        2025        2026        2027
Variable    lag
PAKCCEMISCO2CKN 0       0.00 -22807768.70 -22763629.77 -22643891.23
PAKCCEMISCO2OKN 0       0.00 -13105358.04 -13576625.98 -13868109.25
PAKCCEMISCO2GKN 0       0.00 -16454572.13 -15962197.50 -15504653.56
Share of contributions to difference for PAKCCEMISCO2TKN
  2024        2025        2026        2027
Variable    lag
PAKCCEMISCO2CKN 0       44%      44%      44%
PAKCCEMISCO2GKN 0       31%      31%      30%
PAKCCEMISCO2OKN 0       25%      26%      27%
Total        0       0       100%     100%     100%
Residual     0      -100      -0%       0%      -0%
Difference in growth rate PAKCCEMISCO2TKN
  2024        2025        2026        2027
Variable    lag
Base        0      2.3%      2.5%      2.7%      2.7%
Alternative 0      2.3%     -20.2%     3.5%      3.5%
Difference   0      0.0%     -22.7%      0.8%      0.9%
None
Contribution to growth rate PAKCCEMISCO2TKN
  2024        2025        2026        2027
Variable    lag
PAKCCEMISCO2CKN 0      0.0%     -9.9%      0.4%      0.4%
PAKCCEMISCO2OKN 0      0.0%     -5.7%     -0.0%      0.1%
PAKCCEMISCO2GKN 0      0.0%     -7.1%      0.5%      0.5%
Total        0      0.0%     -22.7%      0.9%      1.0%
Residual     0      0.0%      0.0%      0.1%      0.1%

```

The above results from the call to `.dekompo()` are presented in several sections.

Section	Table	Contents
The first section		the normalized formula of the RHS variable PAKCCEMISCO2TKN
The second section		Shows the changes in level terms.
	diff_level	First by showing the results of the simulation base , then the previous level last , then the difference and then the difference expressed as a percent
	att_level	This is followed by a table showing the contribution of the changes in every LHS variable to the observed change in the dependent variable.
The third section	att_pct	Shows the same results for the RHS variables, but expressed as a percent of the total change in the dependent variable.
The fourth section		Shows the same results but for the change in the growth rate of the dependent variable.
	diff_gr	The first table shows the post-shock growth rate of the dependent variable from the .laststdf dataframe, followed by the pre-shock growth rate and the difference in the growth rates.
	att_gro	The second table of this section shows the contribution to the change in the growth rate from each RHS variable.

The object returned by `.dekompo()` is a [namedtuple](https://realpython.com/python-namedtuple/) (<https://realpython.com/python-namedtuple/>) that contains each of these tables which can then be referred to later.

The code below extracts the different sub-components of the `.dekompo()` results and displays them individually.

```
# Loop over the elements in the result of dekompo.
# a named tuple can be used both as a straight tuple and the elements
# can be accessed through the field name.
with pd.option_context('display.float_format', '{:.2f}'.format):
    for f,df in zip(dekompo_result._fields, dekompo_result):
        display(f)
        display(df)
```

'diff_level'

		2024	2025	2026	2027
Variable	lag				
Base	0	230370296.36	236240661.98	242537408.00	248995663.61
Alternative	0	230370296.36	183872963.10	190234954.74	196979009.56
Difference	0		0.00	-52367698.88	-52302453.26
Percent	0		-0.00	-22.17	-21.56
					-20.89

'att_level'

		2024	2025	2026	2027
Variable	lag				
PAKCCEMISCO2CKN	0	0.00	-22807768.70	-22763629.77	-22643891.23
PAKCCEMISCO2OKN	0	0.00	-13105358.04	-13576625.98	-13868109.25
PAKCCEMISCO2GKN	0	0.00	-16454572.13	-15962197.50	-15504653.56

```
'att_pct'
```

	lag	2024	2025	2026	2027
Variable					
PAKCCEMISCO2CKN	0		NaN	43.55	43.52
PAKCCEMISCO2GKN	0		NaN	31.42	30.52
PAKCCEMISCO2OKN	0		NaN	25.03	25.96
Total	0		0.00	100.00	100.00
Residual	0	-100.00	-0.00	0.00	-0.00

```
'diff_growth'
```

	lag	2024	2025	2026	2027
Variable					
Base	0	2.27	2.55	2.67	2.66
Alternative	0	2.27	-20.18	3.46	3.55
Difference	0	0.00	-22.73	0.79	0.88

```
'att_growth'
```

	lag	2024	2025	2026	2027
Variable					
PAKCCEMISCO2CKN	0	0.00	-9.90	0.40	0.44
PAKCCEMISCO2OKN	0	0.00	-5.69	-0.01	0.09
PAKCCEMISCO2GKN	0	0.00	-7.14	0.53	0.50
Total	0	0.00	-22.73	0.92	1.02
Residual	0	0.00	0.00	0.13	0.14

16.5 A more complex example

The above decomposition is fairly straight forward because the decomposed equation is a simple identity, where Total Emissions are just the sum of its three component parts: Total Carbon emissions = Emissions from Oil+ Emissions from Coal + Emissions from Natural Gas.

The following single-equation decomposition looks to the impact of the same shock (introduction of a carbon tax) on a different variable (inflation). The inflation equation is more complex and has more direct causal variables, so the decomposition is more interesting.

Recall the inflation equation is given by the `.frml` method for its normalized version and `.eviews` for its original specification. The equation for the consumer price level (PAKNECONPRVTXN) was originally specified in eviews as:

```
mpak [ 'PAKNECONPRVTXN' ].eviews
```

```
PAKNECONPRVTXN :
@IDENTITY PAKNECONPRVTXN = ((PAKNECONENGYSH^PAKCESENGYCON) * PAKNECONENGYXN^(1 - 
    ↵ PAKCESENGYCON) + (PAKNECONOTHRS^PAKCESENGYCON) * PAKNECONOTHRXN^(1 - 
    ↵PAKCESENGYCON))^(1 / (1 - PAKCESENGYCON))
```

The normalized equation is given by `mpak ['PAKNECONPRVTXN'].frml`.

Note in the Pakistan model, consumer inflation is derived as a constant elasticity of transformation (CET) aggregation of the price of energy goods(PAKNECONENGYXN) and non-energy goods (PAKNECONOTHRXN).

```
mpak [ 'PAKNECONPRVTXN' ] .frml
```

```
PAKNECONPRVTXN : FRML <IDENT> PAKNECONPRVTXN =  
↳ ( (PAKNECONENGYSH**PAKCESENGYCON) *PAKNECONENGYXN** (1-  
↳ PAKCESENGYCON) + (PAKNECONOTHRSRSH**PAKCESENGYCON) *PAKNECONOTHRXN** (1-  
↳ PAKCESENGYCON) ) ** (1 / (1-PAKCESENGYCON)) $
```

Note further the normalized equation is solving for the **level** of the price deflator – not inflation which is the rate of growth of this index.

Because the equation solves for the level of the price deflator, the decomposition show the contributions of each explanatory variable to the increase in the price level (not that of the inflation rate). However, the 4th table is showing the impacts on the rate of growth of the price level – i.e. the level of inflation.

```
mpak [ 'PAKNECONPRVTXN' ] .dekomp (start=2024, end=2027);
```

Formula	:	FRML <IDENT> PAKNECONPRVTXN =		
↳ ((PAKNECONENGYSH**PAKCESENGYCON)*PAKNECONENGYXN** (1-				
↳ PAKCESENGYCON) + (PAKNECONOTHRSRSH**PAKCESENGYCON)*PAKNECONOTHRXN** (1-				
↳ PAKCESENGYCON))** (1/(1-PAKCESENGYCON)) \$				
	2024	2025	2026	2027
Variable	lag			
Base	0	2.30	2.45	2.60
Alternative	0	2.30	2.51	2.68
Difference	0	0.00	0.06	0.07
Percent	0	-0.00	2.47	2.74
Contributions to difference for	PAKNECONPRVTXN			
	2024	2025	2026	2027
Variable	lag			
PAKNECONENGYSH	0	-0.00	-0.00	-0.00
PAKCESENGYCON	0	-0.00	-0.00	-0.00
PAKNECONENGYXN	0	-0.00	0.01	0.01
PAKNECONOTHRSRSH	0	-0.00	-0.00	-0.00
PAKNECONOTHRXN	0	-0.00	0.05	0.06
Share of contributions to difference for	PAKNECONPRVTXN			
	2024	2025	2026	2027
Variable	lag			
PAKNECONOTHRXN	0	77%	81%	83%
PAKNECONENGYXN	0	23%	20%	18%
PAKNECONENGYSH	0	-0%	-0%	-0%
PAKCESENGYCON	0	-0%	-0%	-0%
PAKNECONOTHRSRSH	0	-0%	-0%	-0%
Total	0	100%	100%	100%
Residual	0	-100	0%	0%
Difference in growth rate PAKNECONPRVTXN				
	2024	2025	2026	2027
Variable	lag			
Base	0	7.3%	6.7%	6.2%
Alternative	0	7.3%	9.3%	6.5%
Difference	0	0.0%	2.6%	0.3%
None				
Contribution to growth rate PAKNECONPRVTXN				
	2024	2025	2026	2027
Variable	lag			
PAKNECONENGYSH	0	-0.0%	0.0%	-0.0%
PAKCESENGYCON	0	-0.0%	0.0%	-0.0%
PAKNECONENGYXN	0	-0.0%	0.6%	-0.0%
PAKNECONOTHRSRSH	0	-0.0%	0.0%	-0.0%
PAKNECONOTHRXN	0	-0.0%	2.0%	0.3%
Total	0	-0.0%	2.7%	0.3%
Residual	0	-0.0%	0.0%	-0.0%

Interestingly only 23% of the increase in the price level each period is due to the direct channel (the impact on the price of energy consumed by households), the bulk of the increase comes indirectly through other prices. Indeed as time progresses this share rises from 77% in the first year of the price change (2020) to 83% by 2024.

16.5.1 Non-energy prices

Below is the formula for nonenergy consumer prices and their decomposition. This equation is written out as a more standard augmented-phillips-curve type inflation equation reflecting changes in the cost of local goods production (PAKNYGDPCSTXN), Government taxes on goods and services (PAKGGREVGNFSXN), the price of imports (PAKNEIMPGNFSXN) and the influence of the economic cycle (PAKNYGDPGAP_) on the price level.

```
mpak [ 'PAKNECONOTHRXN' ].eviews
```

```
PAKNECONOTHRXN :  
DLOG(PAKNECONOTHRXN) = 0.590372627657176*DLOG(PAKNYGDPCSTXN) + D(PAKGGREVGNFSXN/  
+100) + (1 - 0.590372627657176)*DLOG(PAKNEIMPGNFSXN) + 0.2*PAKNYGDPGAP_/100
```

```
mpak [ 'PAKNECONOTHRXN' ].dekomp(start=2025, end=2029);
```

Formula :

```
FRML <DAMP, STOC> PAKNECONOTHRXN = (PAKNECONOTHRXN(-1)*EXP(PAKNECONOTHRXN_A+ (0.  
-590372627657176* ((LOG(PAKNYGDPCSTXN)) - (LOG(PAKNYGDPCSTXN(-  
-1)))) + ((PAKGGREVGNFSXN/100) - (PAKGGREVGNFSXN(-1)/100)) + (1-0.  
-590372627657176) * ((LOG(PAKNEIMPGNFSXN)) - (LOG(PAKNEIMPGNFSXN(-1)))) + 0.  
+2*PAKNYGDPGAP_/_100) ) * (1-PAKNECONOTHRXN_D) + PAKNECONOTHRXN_X*PAKNECONOTHRXN_D_  
+ $
```

		2025	2026	2027	2028	2029
Variable	lag					
Base	0	2.50	2.65	2.81	2.96	3.11
Alternative	0	2.55	2.71	2.87	3.03	3.19
Difference	0	0.05	0.06	0.07	0.08	0.08
Percent	0	1.95	2.26	2.47	2.61	2.65

Contributions to difference for PAKNECONOTHRXN

		2025	2026	2027	2028	2029
Variable	lag					
PAKNECONOTHRXN	-1	-0.00	0.05	0.06	0.07	0.08
PAKNECONOTHRXN_A	0	-0.00	-0.00	-0.00	-0.00	-0.00
PAKNYGDPCSTXN	0	0.00	0.01	0.01	0.02	0.02
	-1	-0.00	-0.00	-0.01	-0.01	-0.02
PAKGGREVGNFSXN	0	-0.00	-0.00	-0.00	-0.00	-0.00
	-1	-0.00	-0.00	-0.00	-0.00	-0.00
PAKNEIMPGNFSXN	0	0.05	0.05	0.05	0.05	0.05
	-1	-0.00	-0.05	-0.05	-0.05	-0.05
PAKNYGDPGAP_	0	0.00	0.00	0.00	0.00	0.00
PAKNECONOTHRXN_D	0	-0.00	-0.00	-0.00	-0.00	-0.00
PAKNECONOTHRXN_X	0	-0.00	-0.00	-0.00	-0.00	-0.00

Share of contributions to difference for PAKNECONOTHRXN

		2025	2026	2027	2028	2029
Variable	lag					
PAKNECONOTHRXN	-1	-0%	87%	91%	95%	98%
PAKNEIMPGNFSXN	0	95%	79%	70%	63%	59%
PAKNYGDPCSTXN	0	1%	12%	18%	22%	25%
PAKNYGDPGAP_	0	4%	6%	4%	3%	1%
PAKNECONOTHRXN_A	0	-0%	-0%	-0%	-0%	-0%
PAKGGREVGNFSXN	0	-0%	-0%	-0%	-0%	-0%
	-1	-0%	-0%	-0%	-0%	-0%
PAKNECONOTHRXN_D	0	-0%	-0%	-0%	-0%	-0%
PAKNECONOTHRXN_X	0	-0%	-0%	-0%	-0%	-0%
PAKNYGDPCSTXN	-1	-0%	-1%	-11%	-17%	-22%

(continues on next page)

(continued from previous page)

PAKNEIMPGNFSXN	-1	-0%	-84%	-74%	-67%	-63%
Total	0	100%	99%	99%	99%	99%
Residual	0	0%	-1%	-1%	-1%	-1%
Difference in growth rate PAKNECONOTHRXN						
		2025	2026	2027	2028	2029
Variable	lag					
Base	0	6.7%	6.2%	5.8%	5.4%	5.1%
Alternative	0	8.7%	6.5%	6.0%	5.6%	5.2%
Difference	0	2.1%	0.3%	0.2%	0.1%	0.0%
None						
Contribution to growth rate PAKNECONOTHRXN						
		2025	2026	2027	2028	2029
Variable	lag					
PAKNECONOTHRXN	-1	0.0%	2.0%	0.3%	0.2%	0.1%
PAKNECONOTHRXN_A	0	0.0%	-0.0%	-0.0%	0.0%	-0.0%
PAKNYGDPFCSTXN	0	0.0%	0.3%	0.2%	0.1%	0.1%
	-1	0.0%	-0.0%	-0.3%	-0.2%	-0.1%
PAKGREVGNSXN	0	0.0%	-0.0%	-0.0%	0.0%	-0.0%
	-1	0.0%	-0.0%	-0.0%	0.0%	-0.0%
PAKNEIMPGNFSXN	0	2.0%	-0.1%	-0.1%	-0.1%	-0.1%
	-1	0.0%	-2.0%	0.1%	0.1%	0.1%
PAKNYGDPGAP_	0	0.1%	0.0%	-0.0%	-0.0%	-0.0%
PAKNECONOTHRXN_D	0	0.0%	-0.0%	-0.0%	0.0%	-0.0%
PAKNECONOTHRXN_X	0	0.0%	-0.0%	-0.0%	0.0%	-0.0%
Total	0	2.1%	0.3%	0.2%	0.1%	0.0%
Residual	0	0.0%	-0.0%	-0.0%	-0.0%	-0.0%

These results indicate that much of the initial impact on prices is coming from the increase in the price of imported goods (which includes a large fuel component). As time progresses, the imported inflation component declines (because fuel and import prices are no longer rising) and the lagged consumption price dominates (the level this period is basically determined by the price level in the previous period). Other factors such as the cost of domestically produced goods play a larger role and the net impact of imported prices (the total of the contemporaneous and lagged value) approaches zero. Cyclical pressure are initially adding to inflation before declining and eventually turning negative.

16.6 The `get_att()` method provides more control over the outputs of `.dekomp()`

Following a call to the `.dekomp()` method, the `.get_att()` method provides a range of mechanisms that allow the results to be displayed in different ways.

16.6.1 The default display of `.get_att()`

By default `.get_att()` displays the share contributions of RHS variables to the total change in the LHS variable. The `start=` and `end=` options allow the period for which results are displayed to be restricted.

```
mpak.PAKNECONPRVTKN.get_att(start=2025, end=2035)
```

	2020	2021	2022	2023	2024	2025	2026	2027	2028	2029	2030
Percent att.											
PAKNECONPRVTKN(-1)	0%	60%	64%	67%	72%	77%	80%	83%	85%	87%	88%
PAKNECONPRVTXN	154%	131%	111%	96%	86%	80%	77%	76%	76%	77%	78%
PAKNYYWBOTLCN(-1)	0%	25%	30%	30%	29%	29%	31%	35%	39%	43%	48%
PAKBXFSTREMTCD(-1)	0%	12%	12%	11%	10%	9%	8%	7%	7%	7%	7%
PAKGEXPTRNSCN(-1)	0%	1%	1%	1%	1%	1%	2%	2%	2%	2%	2%
PAKFMLBLPOLYXN	13%	11%	9%	7%	6%	5%	4%	3%	2%	2%	1%
PAKPANUSATLS	2%	2%	1%	1%	1%	1%	1%	1%	1%	1%	1%
PAKBMFSTREMTCD	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
PAKNECONPRVTKN_A	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
PAKGGREVDRCTXN(-1)	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
PAKGGREVDRCTXN	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
DURING_2009	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
PAKNECONPRVTKN_D	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
PAKNECONPRVTKN_X	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
PAKBMFSTREMTCD(-1)	0%	-0%	-0%	-0%	-0%	-0%	-0%	-0%	-0%	-0%	-0%
PAKPANUSATLS(-1)	0%	-1%	-1%	-1%	-1%	-1%	-1%	-1%	-1%	-1%	-1%
PAKGEXPTRNSCN	-1%	-2%	-2%	-2%	-2%	-2%	-2%	-2%	-2%	-3%	-3%
PAKBXFSTREMTCD	-22%	-21%	-18%	-14%	-12%	-11%	-10%	-9%	-9%	-9%	-9%
PAKNECONPRVTXN(-1)	0%	-65%	-58%	-52%	-49%	-46%	-45%	-45%	-46%	-46%	-48%
PAKNYYWBOTLCN	-45%	-51%	-49%	-43%	-41%	-42%	-45%	-49%	-53%	-59%	-65%

16.6.2 Options: Lag=True/False

Because the decomposition of the equation is based on the normalized (levelized) version of the equation, for equations initially written as growth rates or ECMs many variables will occur several times in the attribution table with both the contribution of the current value of the variable and those of any lagged versions that appear in the normalized equation.

The `Lag=False` option changes the default behavior of `.get_att()` and aggregates the contributions of different lags.

By aggregating the lags, the net effect of changes in the variables can be more easily determined. Below it is clearer that the initial impact of higher import prices drove most of the inflation response. In subsequent periods, import prices were stable or even falling so most of the contribution to the change in the level of other goods inflation was from the lagged dependent variable and the changed state of the economic cycle (the Gap variable which initially was adding to inflationary pressures eventually subtracts from inflation as the economy slows).

```
mpak.PAKNECONOTHRXN.get_att(lag=False, start=2025, end=2035)
```

	2020	2021	2022	2023	2024	2025	2026	2027	2028	2029	2030
Percent att.											
PAKNECONPRVTKN(-1)	0%	60%	64%	67%	72%	77%	80%	83%	85%	87%	88%
PAKNECONPRVTXN	154%	131%	111%	96%	86%	80%	77%	76%	76%	77%	78%
PAKNYYWBTOTLCN(-1)	0%	25%	30%	30%	29%	29%	31%	35%	39%	43%	48%
PAKBXFSTREMTCD(-1)	0%	12%	12%	11%	10%	9%	8%	7%	7%	7%	7%
PAKGGEPRTRNSCN(-1)	0%	1%	1%	1%	1%	1%	2%	2%	2%	2%	2%
PAKFMLBLPOLYXN	13%	11%	9%	7%	6%	5%	4%	3%	2%	2%	1%
PAKPANUSATLS	2%	2%	1%	1%	1%	1%	1%	1%	1%	1%	1%
PAKBMFSTREMTCD	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
PAKNECONPRVTKN_A	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
PAKGGREVDRCTXN(-1)	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
PAKGGREVDRCTXN	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
DURING_2009	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
PAKNECONPRVTKN_D	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
PAKNECONPRVTKN_X	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
PAKBMFSTREMTCD(-1)	0%	-0%	-0%	-0%	-0%	-0%	-0%	-0%	-0%	-0%	-0%
PAKPANUSATLS(-1)	0%	-1%	-1%	-1%	-1%	-1%	-1%	-1%	-1%	-1%	-1%
PAKGGEPRTRNSCN	-1%	-2%	-2%	-2%	-2%	-2%	-2%	-2%	-2%	-3%	-3%
PAKBXFSTREMTCD	-22%	-21%	-18%	-14%	-12%	-11%	-10%	-9%	-9%	-9%	-9%
PAKNECONPRVTXN(-1)	0%	-65%	-58%	-52%	-49%	-46%	-45%	-45%	-46%	-46%	-48%
PAKNYYWBTOTLCN	-45%	-51%	-49%	-43%	-41%	-42%	-45%	-49%	-53%	-59%	-65%

16.6.3 Options: Type="growth/pct/Level"

The option `Type` controls which of the tables generated by `dekompo()` is displayed. The contributions of RHS variables to the level of the dependent variable (`=level`), the share of the observed change in the RHS variable attributable to each dependent variable (`=pct`), and the change in the growth rate (`=growth`) of the LHS available attributable to the changes in the growth rate of each RHS variable.

16.6.4 Options: threshold=xx"

The `threshold=` option will suppress from the output those variables whose contribution is less than the stated threshold.

In the example below, lags are suppressed, and only contributions to the growth rate of variables whose largest contribution was more than ± 0.1 percent are displayed. The dropped variables influence is aggregated and displayed in a row labeled **small**.

Options: bare=True/False

If bare is set to False then the values of the LHS variable in the `basedf` and `lastdf` dataframes and their difference are also displayed. By default this option is True which suppresses the display.

```
mpak.PAKNECONPRVTKN.get_att(lag=False, type='growth', bare=False, threshold=0.1,
                           start=2020, end=2024)
```

	2020	2021	2022	2023	2024
	Growth percent				
Alternative	1.81%	0.93%	0.46%	0.77%	1.39%
Base	2.84%	1.27%	0.80%	1.09%	1.60%
Difference	-1.03%	-0.33%	-0.34%	-0.32%	-0.21%
PAKBXFSTREMTCD	0.23%	0.12%	0.09%	0.07%	0.06%
PAKFMLBLPOLYXN	-0.14%	-0.15%	-0.16%	-0.15%	-0.14%
PAKNECONPRVTKN	-0.00%	0.20%	0.27%	0.34%	0.40%
PAKNECONPRVTXN	-1.59%	-0.89%	-0.90%	-0.88%	-0.85%
PAKNYYWBTOTLCN	0.46%	0.35%	0.32%	0.26%	0.27%
Small	-0.01%	0.01%	0.01%	0.01%	0.01%

16.6.5 Several examples

Here the default type (pct) is displayed and the threshold is set to 10, so only variables whose aggregate impact was more than 10 percent of the total in one or more of the displayed years are shown.

```
mpak.PAKNECONPRVTKN.get_att(lag=False, threshold=10, start=2025, end=2035)
```

	2020	2021	2022	2023	2024	2025	2026	2027	2028	2029	2030
	Percent att.										
PAKBXFSTREMTCD	-22%	-9%	-5%	-3%	-2%	-2%	-2%	-2%	-2%	-2%	-2%
PAKFMLBLPOLYXN	13%	11%	9%	7%	6%	5%	4%	3%	2%	2%	1%
PAKNECONPRVTKN	0%	60%	64%	67%	72%	77%	80%	83%	85%	87%	88%
PAKNECONPRVTXN	154%	66%	53%	43%	37%	34%	32%	31%	30%	30%	31%
PAKNYYWBTOTLCN	-45%	-26%	-19%	-13%	-12%	-12%	-13%	-14%	-15%	-15%	-16%
Small	1%	-1%	-1%	-0%	-0%	-0%	-0%	-0%	-0%	-1%	-1%

The three examples below show the impact on real consumption of the changes induced on its LHS variables as changes in percent level and growth, with the threshold set to focus only on the main channels.

```
mpak.PAKNECONPRVTKN.get_att(lag=False,threshold=10,bare=False,start=2025,end=2029);  
mpak.PAKNECONPRVTKN.get_att(lag=False,threshold=10,type='level',bare=False,start=2025,  
    ↪end=2029);  
mpak.PAKNECONPRVTKN.get_att(lag=False,threshold=0.1,type='growth',bare=False,  
    ↪start=2025,end=2029);
```

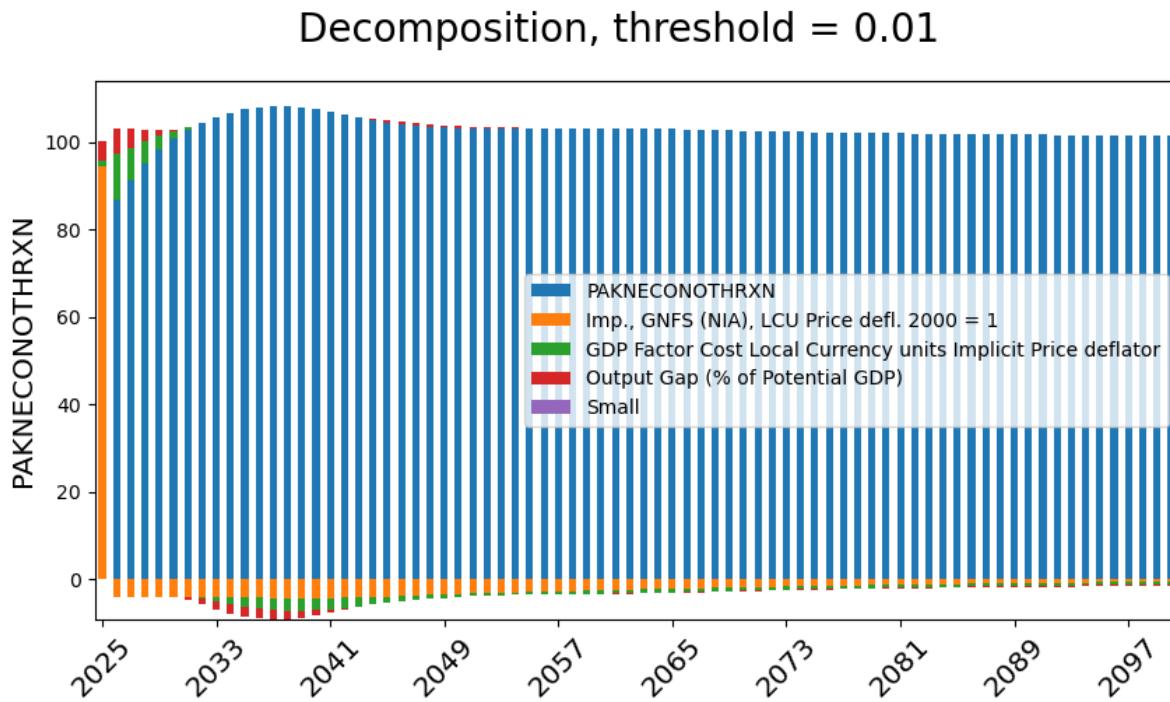
The World Bank's MFMod Framework in Python with Modelflow

	2020	2021	2022	2023	2024
Level/percent					
Alternative	23,435,146	23,653,959	23,761,637	23,944,155	24,276,822
Base	23,672,888	23,972,815	24,164,128	24,427,863	24,818,524
Difference	-237,742	-318,856	-402,491	-483,708	-541,702
Percent	-1	-1	-2	-2	-2
PAKBXFSTREMTCD	-22	-9	-5	-3	-2
PAKFMLBLPOLYXN	13	11	9	7	6
PAKNECONPRVTKN	0	60	64	67	72
PAKNECONPRVTXN	154	68	53	43	37
PAKNYYWBTOTLCN	-45	-26	-19	-13	-12
Small	1	-1	-1	-0	-0
Level/level					
Alternative	23,435,146.20	23,653,959.18	23,761,636.53	23,944,155.30	24,276,822.30
Base	23,672,888.34	23,972,815.36	24,164,128.02	24,427,863.05	24,818,524.47
Difference	-237,742.14	-318,858.17	-402,491.49	-483,707.75	-541,702.17
Percent	-1.00	-1.33	-1.67	-1.98	-2.18
PAKBMFSTREMTCD	-164.32	-69.68	-42.28	-27.30	-20.21
PAKBXFSTREMTCD	52,608.22	28,181.82	20,725.07	15,876.55	13,244.17
PAKFMLBLPOLYXN	-31,247.32	-35,551.65	-36,963.21	-35,725.07	-32,661.88
PAKGGEEXPTRNSCN	3,246.67	3,568.18	3,701.48	3,738.48	3,734.52
PAKNECONPRVTKN	-0.02	-191,775.59	-255,902.57	-323,920.60	-391,556.25
PAKNECONPRVTXN	-386,498.13	-209,229.25	-212,956.18	-209,719.17	-202,966.11
PAKNYYWBTOTLCN	106,192.22	83,170.66	75,443.01	62,515.16	64,803.97
PAKPANUSATLS	-4,290.41	-1,699.98	-1,591.77	-1,439.85	-1,289.11
Small	-0.13	-0.07	-0.04	-0.08	-0.09
Growth percent					
Alternative	1.81%	0.93%	0.46%	0.77%	1.39%
Base	2.84%	1.27%	0.80%	1.09%	1.60%
Difference	-1.03%	-0.33%	-0.34%	-0.32%	-0.21%
PAKBXFSTREMTCD	0.23%	0.12%	0.09%	0.07%	0.06%
PAKFMLBLPOLYXN	-0.14%	-0.15%	-0.16%	-0.15%	-0.14%
PAKNECONPRVTKN	-0.00%	0.20%	0.27%	0.34%	0.40%
PAKNECONPRVTXN	-1.59%	-0.89%	-0.90%	-0.88%	-0.85%
PAKNYYWBTOTLCN	0.48%	0.35%	0.32%	0.28%	0.27%
Small	-0.01%	0.01%	0.01%	0.01%	0.01%

16.6.6 Displaying .dekomp() results graphically

The dekomp_plot method allows single-equation decompositions to be displayed graphically. Below in the initial periods, the import price, and cost-push factors dominate, but as the model equilibrates the lagged level of the price deflator explains virtually all of the movement in the level of the price.

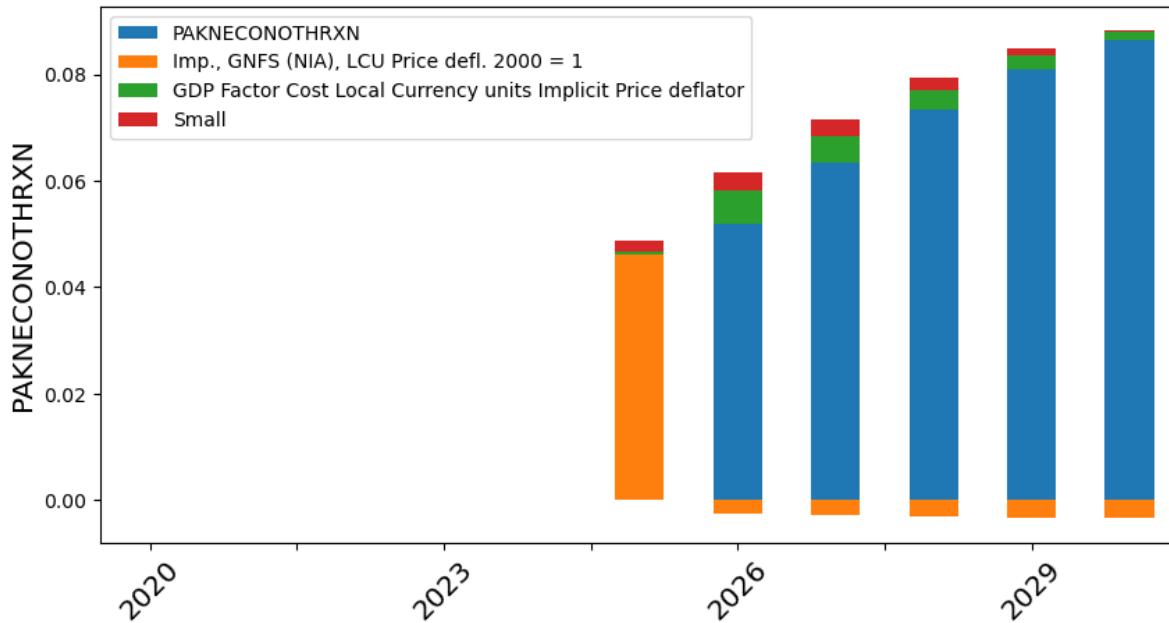
```
fig=mpak.dekomp_plot('PAKNECONOTHRXN', pct=True, rename=True, threshold=.01, lag=False);
#decomp of the change in the level
```



In the following example the change in the level of the dependent variable is displayed for a restricted time period. Here the distinction between the initial impulse (import prices) and the lagged effect of past prices is very evident.

```
with mpak.set_smpl(2020,2030):
    fig=mpak.dekomp_plot('PAKNECONOTHRXN', pct=False, rename=True, threshold=.005,
    lag=False); #decomp of the change in the level
```

Decomposition, threshold = 0.005



16.6.7 the time_att option

The above displays focused on the difference between the values in the two dataframes `basedf` and `lastdf`.

By setting the `time_att` option to True, `get_att()` displays the contribution of changes in the levels of the RHS variables between `t` and `t-1`, in explaining the changes in the LHS variable between `t` and `t-1` with all data pulled from the same `lastdf` dataframe.

Note

With the `time_att` option set **only the `.lastdf` dataframe** is used. The comparison is not `.basedf` vs `.lastdf` but the influence of last year's changes on the level of this year's variable. The attribution is calculated by lagging each right hand side variable one year and recalculating the equation.

```
mpak.PAKCEMISCO2TKN .get_att(time_att= True, type='level', bare=0);
```

```
<pandas.io.formats.style.Styler at 0x167fffbcb9e0>
```

Level	2025	2026	2027	2028	2029	2030	2031	2032	2033	2034	2035	2036	2037	2038
Difference	-40,497,333.45	6,361,989.87	6,744,052.52	6,980,145.17	7,238,795.89	7,475,883.03	7,709,082.30	7,921,378.57	8,105,197.47	8,284,550.39	8,411,112.37	8,557,688.01	8,716,013.58	8,892,517.29
Percent	-20.18	3.40	3.55	3.54	3.55	3.55	3.54	3.53	3.50	3.49	3.41	3.38	3.30	3.22
t	183,872,909.87	190,234,040.24	196,979,001.78	203,650,146.94	211,195,942.83	218,071,025.80	226,307,705.16	234,302,088.73	242,407,284.20	250,071,834.59	259,082,946.40	267,640,655.54	276,356,949.22	284,406.40
t-1	230,370,299.02	183,872,909.87	190,234,040.24	196,979,001.78	203,650,146.94	211,195,942.83	218,071,025.80	226,307,705.16	234,302,088.73	242,407,284.20	250,071,834.59	259,082,946.40	267,640,655.54	276,356,949.22
PAKCEMISCO2TKN	-21,419,382.51	1,405,857.83	1,573,495.08	1,627,058.22	1,683,800.62	1,734,075.59	1,784,326.26	1,832,001.44	1,870,807.48	1,907,516.21	1,942,494.08	1,978,191.47	2,016,705.39	2,050,807.84
PAKCEMISCO2KHN	-8,601,228.13	3,256,545.04	3,441,046.29	3,546,117.39	3,656,134.63	3,758,800.43	3,860,857.1	3,964,940.10	4,055,972.96	4,087,383.59	4,128,057.87	4,169,208.79	4,262,665.13	4,350,829.80
PAKCEMISCO2ORN	-19,560,743.80	1,606,587.00	1,726,612.16	1,808,859.08	1,893,826.85	2,073,852.93	2,159,477.03	2,229,417.04	2,296,851.59	2,342,569.83	2,390,460.42	2,438,283.07	2,482,290.02	2,527,807.84

```
help(mpak.get_att)
```

```
Help on method get_att in module modelclass:
get_att(n, type='pct', filter=False, lag=True, start='', end='', time_att=False, _
(continues on next page)
```

(continued from previous page)

```



```

help(mpak.dekomp_plot)

```

Help on method dekomp_plot in module modelclass:
dekomp_plot(varnavn, sort=True, pct=True, per='', top=0.9, threshold=0.0, lag=True,
    ↪ rename=True, nametrans=<function Dekomp_Mixin.<lambda> at 0x00000167FD9472E0>, ↪
    ↪ time_att=False) method of modelclass.model instance
    Returns a chart with attribution for a variable over the smpl
    Parameters
    -----
    varnavn : TYPE
        variable name.
    sort : TYPE, optional
        . The default is False.
    pct : TYPE, optional
        display pct contribution . The default is True.
    per : TYPE, optional
        DESCRIPTION. The default is ''.
    threshold : TYPE, optional
        cutoff. The default is 0.0.
    rename : TYPE, optional
        Use descriptions instead of variable names. The default is True.
    time_att : TYPE, optional
        Do time attribution . The default is False.
    lag : TYPE, optional
        separate by lags The default is True.
    top : TYPE, optional
        where to place the title
    Returns
    -----
    a matplotlib figure instance .

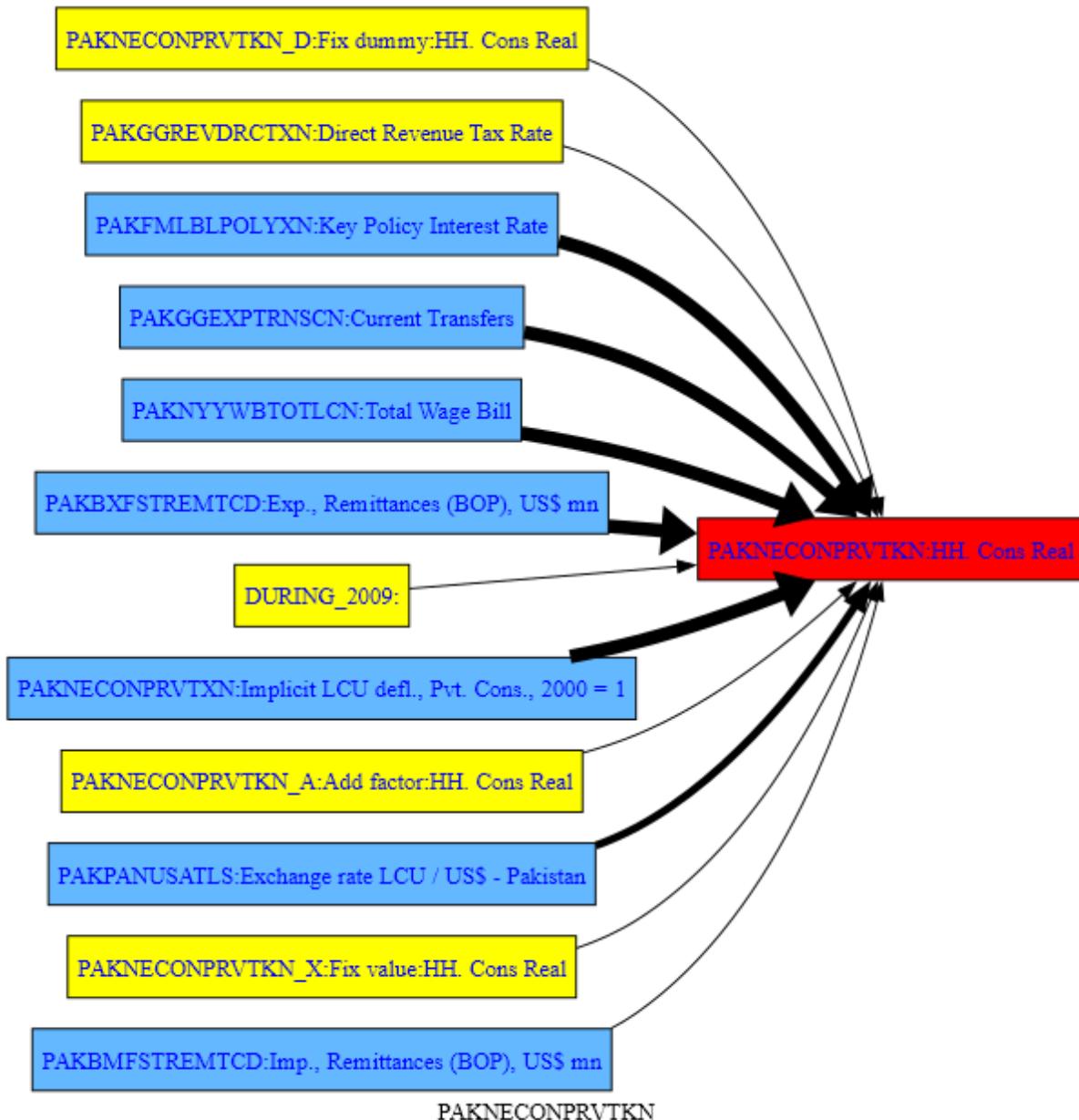
```

16.7 Trace and decomposition combined

The `.tracepre()` method can combine the graphical representation of the `tracepre()` method described in the previous chapter and the tabular results from `dekomp()`.

This is implicit in the standard call to `.tracepre()` where the thickness of the lines is derived from the empirical importance of the changes in each LHS variable in determining the change in the RHS variable.

```
mpak.PAKNECONPRVTKN.tracepre(png=latex,size=(2,4));
```

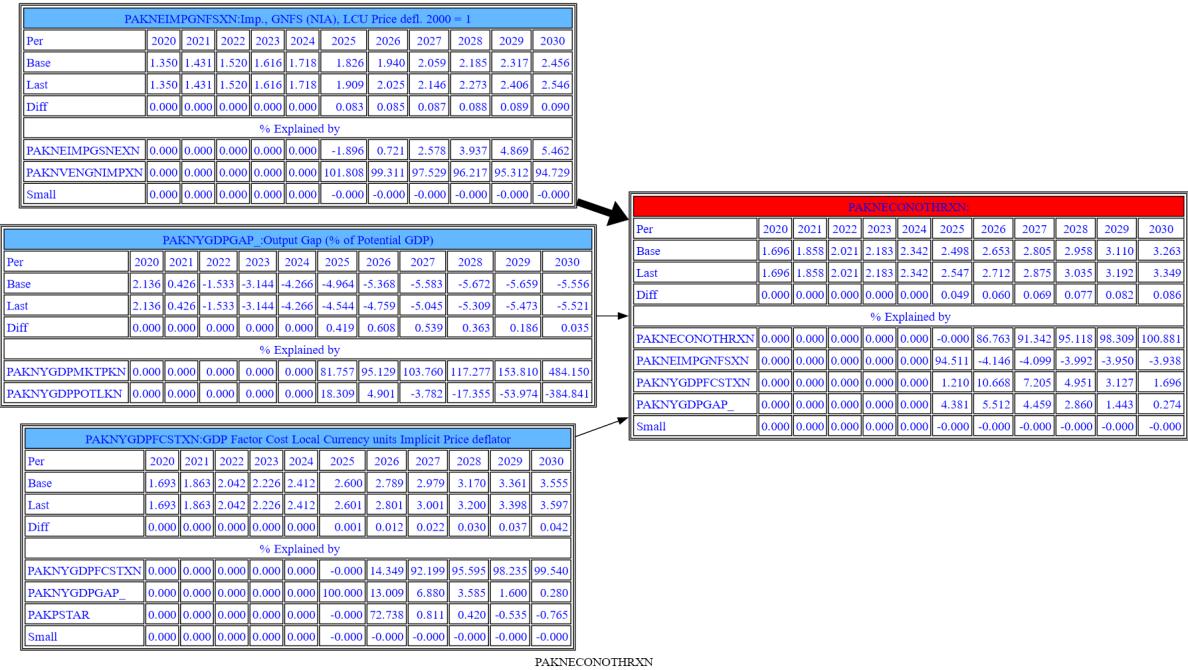


16.8 Tabular output from tracepre()

The results for `.tracepre` can be displayed in a number of ways and the results can be saved as pictures.

up = xx		determines how many levels of parents to include
showdatasd=True		Causes the tables of attribution for each displayed variable to be displayed
showdatasd=<'pattern of variable names'>		will include a table of values for each variable matching the pattern (including wildcharts)
attshowlats = True		adds in the contribution of each to the total change
growthshowlgs = True		will include a table of growth for each variable
HR = True		will reorient the dependency graph
filter=<xx>		restrict outputs to variables that explain at least xx% of the change in the level of dependent variable
browser = True		Opens a browser with the resulting dependency graph - useful for zooming on a big graph or table
png = True		will display as a png picture
svg = True		will display as a svg picture which can be zoomed
pdf = True		will display as a pdf picture
eps = True		will create a eps file (a latex format)
saveas = <a file name without extension>		will save the picture with the filename with an added extension reflecting the picture type

```
with mpak.set_smpl(2020, 2030):
    mpak.PAKNECONOTHRXN.tracepre(filter=5.0, HR=False, showdata=True, attshow=True,
    per=2020, png=latex)
```



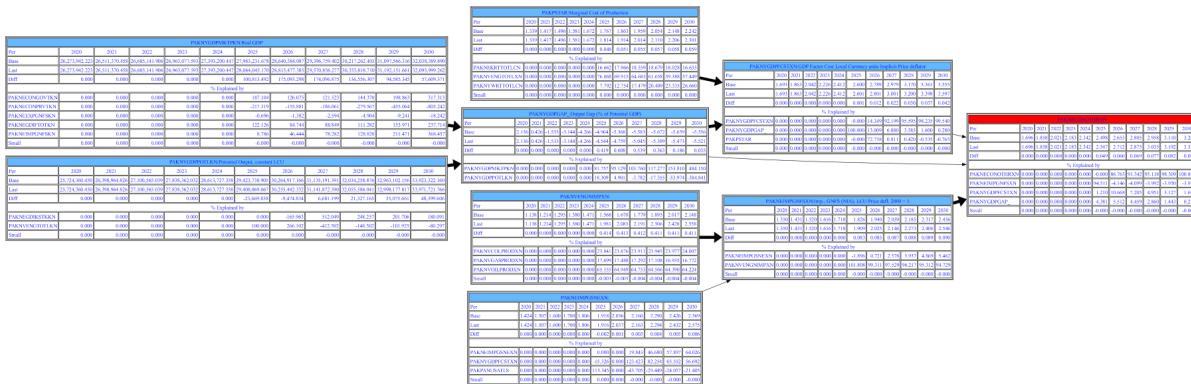
The big difference with this representation is the contributions of both the direct that directly impact the LHS variable

The World Bank's MFMod Framework in Python with Modelflow

(as well as the influence of those variables one or two steps up the causal chain) can be traced.

Below the same command as above but we specify that we want to go up two levels in the causal chain.

```
with mpak.set_smpl(2020,2030):
    mpak.PAKNECONOTHRXN.tracepre(up=2,filter=5,HR=False,sd= True,ats=True,png=latex)
```

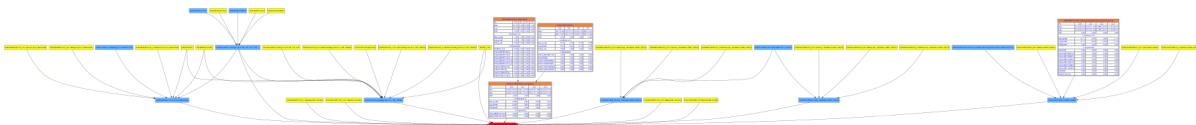


```
with mpak.set_smpl(2020,2023):
    mpak.PAKNECONPRVTKN.tracepre(sd='*lcn',filter=10,HR=1,ats=1,up=2
        ,growthshow=1, png=latex)
```

```
No graph PAKNECONPRVTKN
The graph is empty
Perhaps filter prunes to much
```

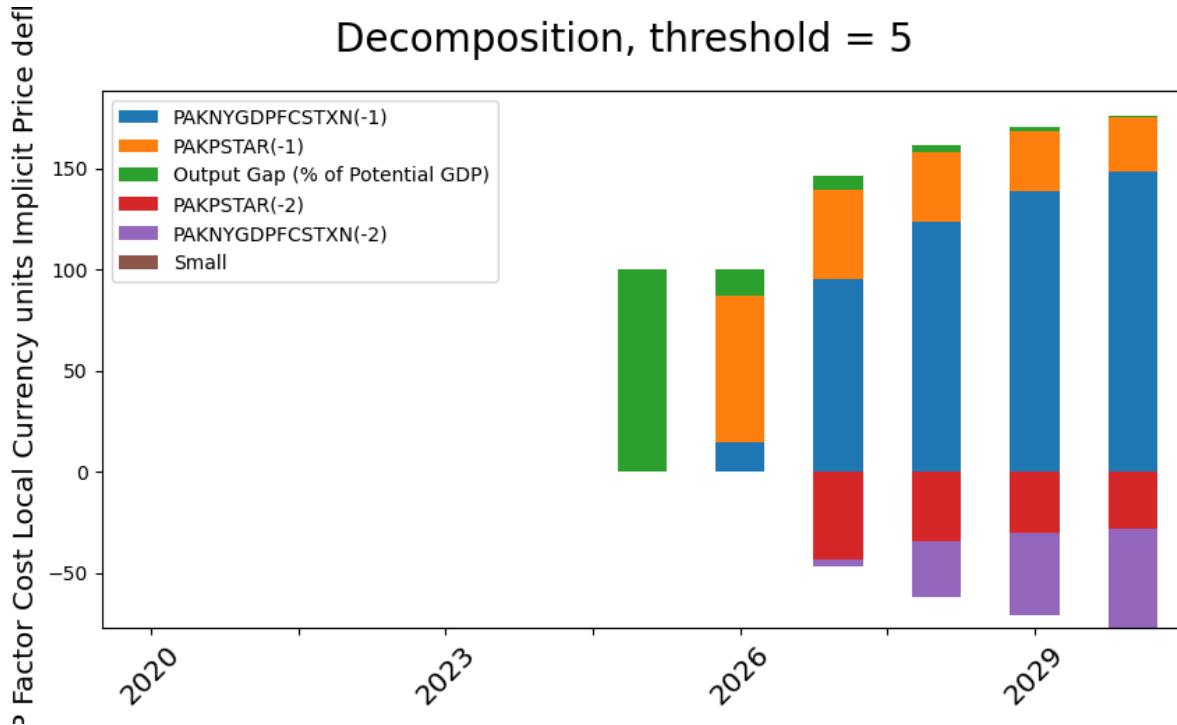
As indicated by the error message the filter is too fine, and has eliminated all variables from the output. Below the same command without the filter option.

```
with mpak.set_smpl(2020,2023):
    mpak.PAKNECONPRVTKN.tracepre(sd='*lcn',HR=1,ats=1,up=2
        ,growthshow=1, png=latex)
```



16.9 Chart of the contributions over time

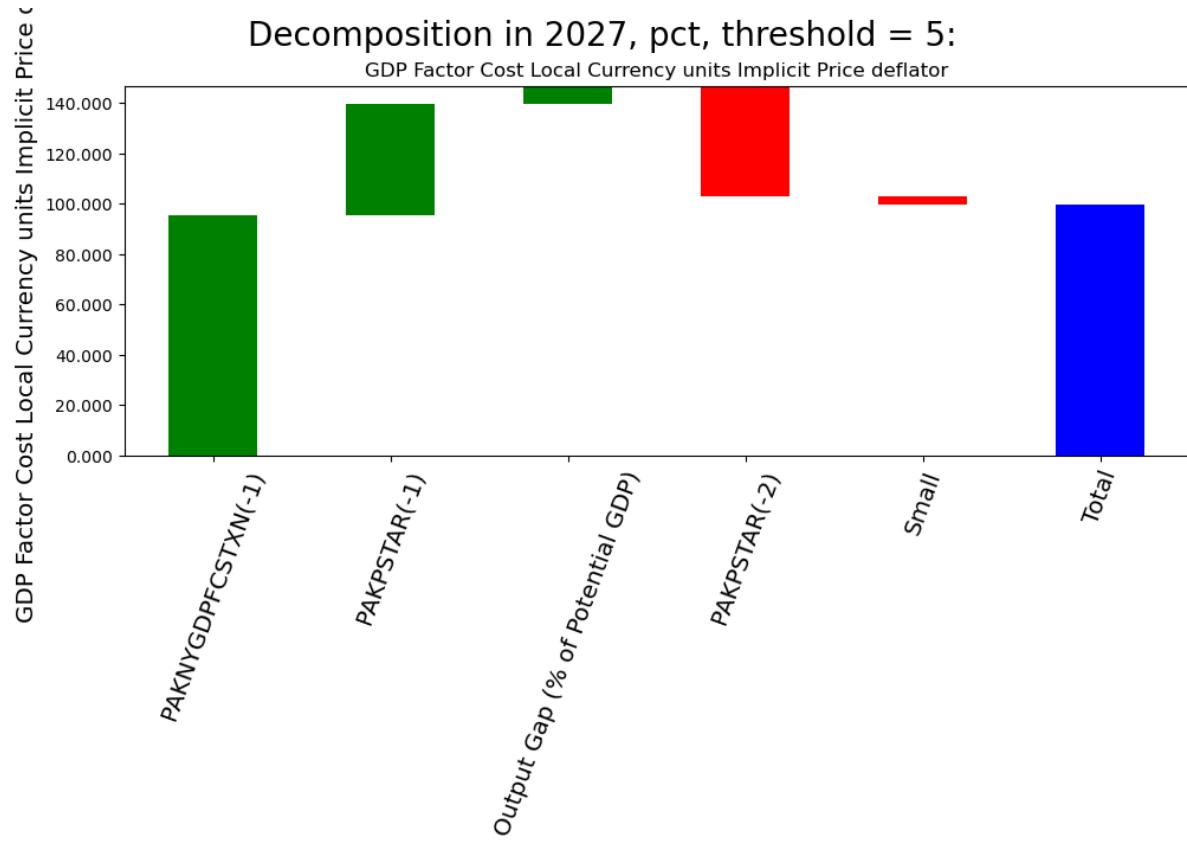
```
with mpak.set_smpl(2020,2030):
    mpak.dekomp_plot('PAKNYGDPFCSTXN',threshold=5); # gives a waterfall of
    ↳contributions
```



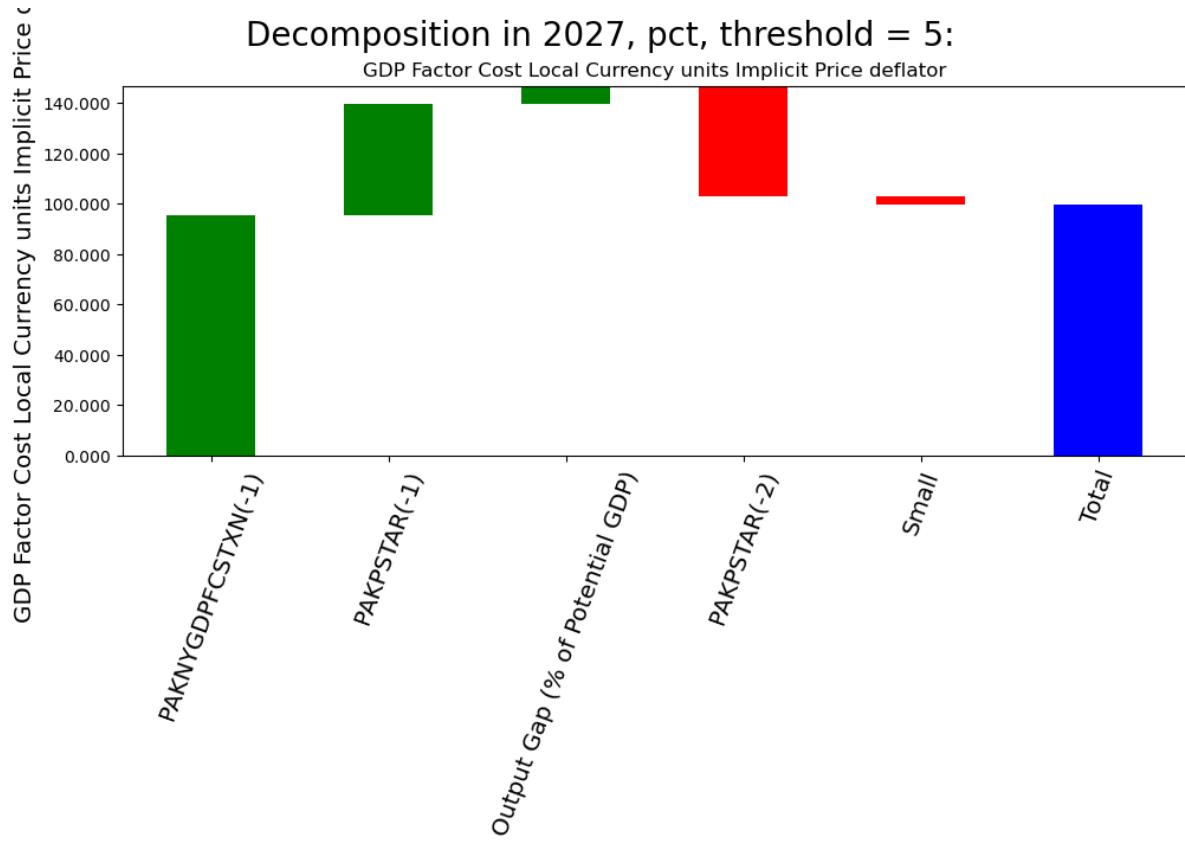
16.10 Chart of the contributions for one year

It can be useful to visualize the attribution as a waterfall chart for a single year

```
mpak.dekomp_plot_per('PAKNYGDPFCSTXN',per=2027,threshold=5) # gives a waterfall of
    ↳contributions
```

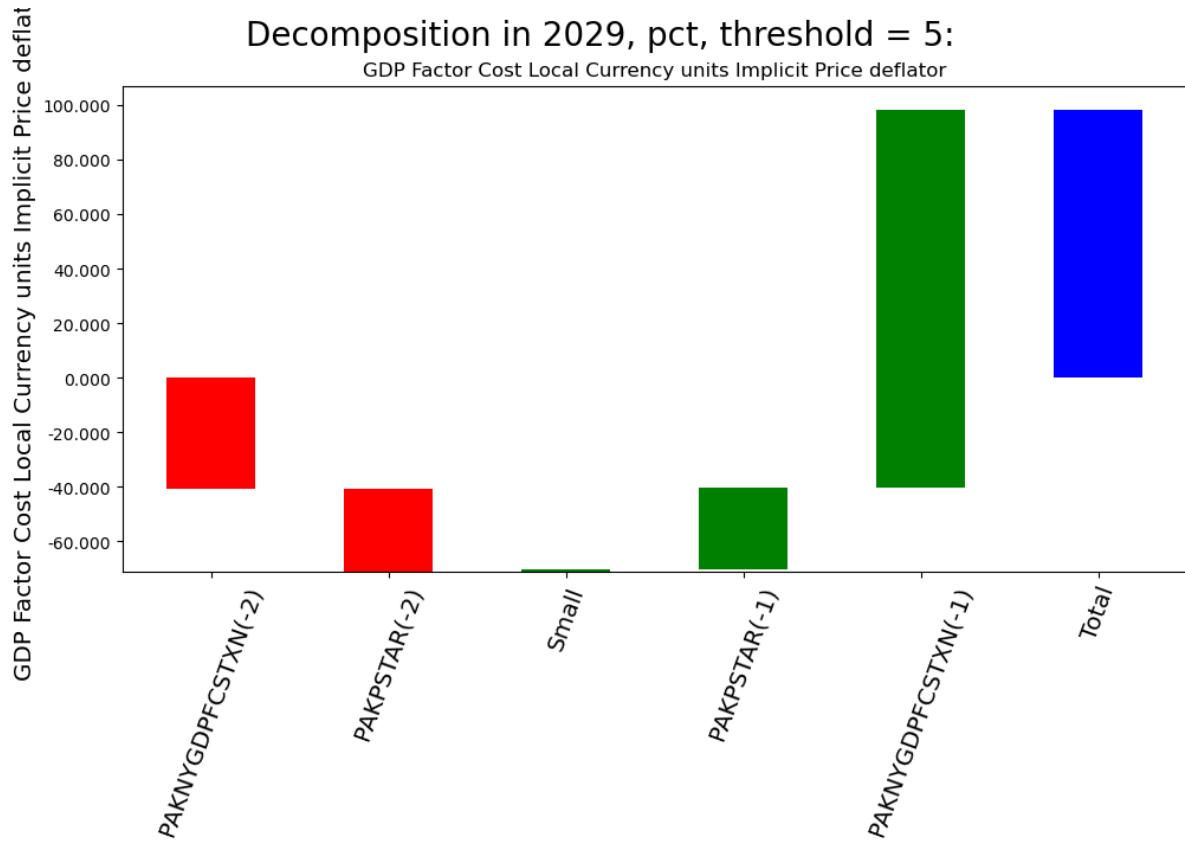


```
mpak.dekomp_plot_per('PAKNYGDPFCSTXN',per=2027,threshold=5) # gives a waterfall of contributions
```



16.11 Sorted waterfall of contributions

```
mpak.dekomp_plot_per('PAKNYGDPFCSTXN', per=2029, threshold=5, sort=True) # gives a
➥waterfall of contributions
```



16.12 Impacts at the model level: the `.totdif()` method

The method `.totdif()` returns an instance of the `totdif` class, which provides a number of methods and properties to explore decomposition at the model level.

It works by solving the model numerous times, each time changing one of the right hand side variables and calculating the impact on all dependent variables. By default it uses the values from the `.lastdf` DataFrame as the shock values and the values in `.basedf` as the initial values. Separate simulations are run for every exogenous (or exogenized) variables that have changed between the two DataFrames.

For advanced users the RHS variables can be grouped into user defined blocks, which in cases where there are many changes can help identify the main causal pathways.

16.12.1 The `.exo_dif()` method

The `.exodif()` method displays only the exogenous variables that have changed between the two DataFrames (the shock). Exogenous variables whose results have not changed are omitted. It determines which of the exogenous variables have changed between `.lastdf` and `.basedf` and then returns a DataFrame with the changes in the values.

In this case the DataFrame contains the effect of updating the CO^2 tax to 30 for coal, gas and oil. `.exo_dif()` is automatically called by the `.totdif()` method but can also be called directly by the user.

```
mpak.exodif()
```

	PAKGGREVC02CER	PAKGGREVC02GER	PAKGGREVC02OER
2020	35.55	71.0	38.71
2021	35.55	71.0	38.71
2022	35.55	71.0	38.71
2023	35.55	71.0	38.71
2024	35.55	71.0	38.71
...
2096	35.55	71.0	38.71
2097	35.55	71.0	38.71
2098	35.55	71.0	38.71
2099	35.55	71.0	38.71
2100	35.55	71.0	38.71
[81 rows x 3 columns]			

16.12.2 The `.totdif()` command calculates the contribution of each changed variable to the changes in a specified LHS variable

This involves solving the model a number of times, so can take some time. How long it takes to execute will depend on the computer, the model and the number of changes made. In this instance the `.totaldif` takes between 2 and 5 seconds depending on computer.

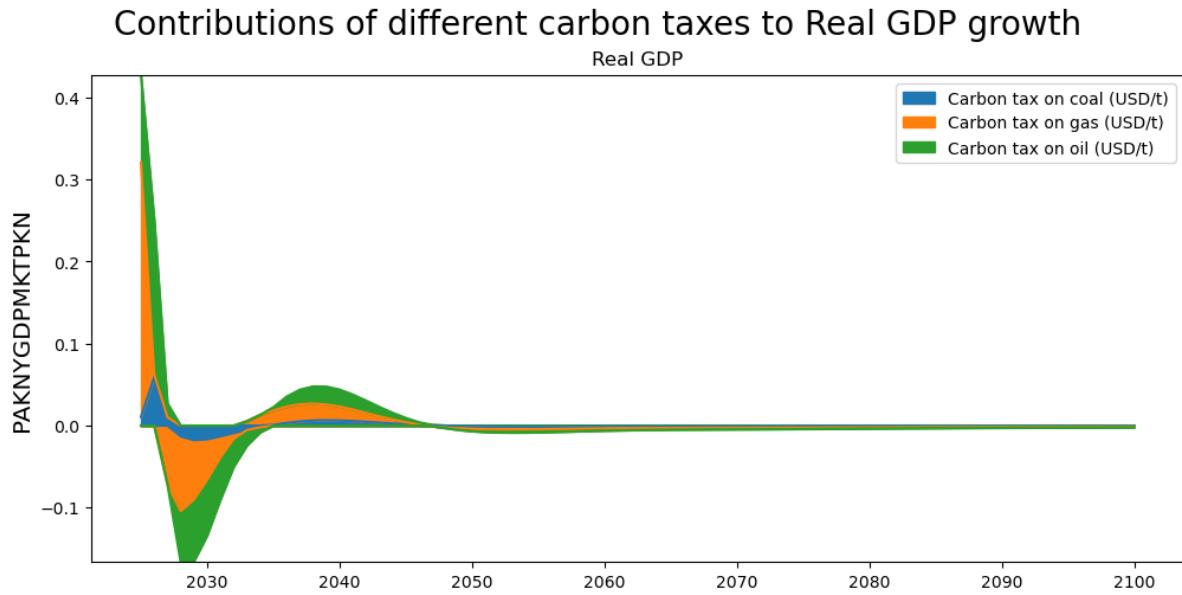
```
totdecomp = mpak.totdif() # Calculate the total derivatives of all equations in the
                           ↪model.
```

Total dekomp took : 6.308 Seconds

16.12.3 The method `.explain_all()` presents the results graphically

In the example below, the relative importance of the three shocked carbon taxes on the change in real GDP are presented.

```
showvar = 'PAKNYGDPMKTPKN'
totdecomp.explain_all(showvar, kind='area', use='growth', stacked=True,
                      title="Contributions of different carbon taxes to Real GDP_
                           ↪growth") ;
```



```
help(totdekom.explain_all)
```

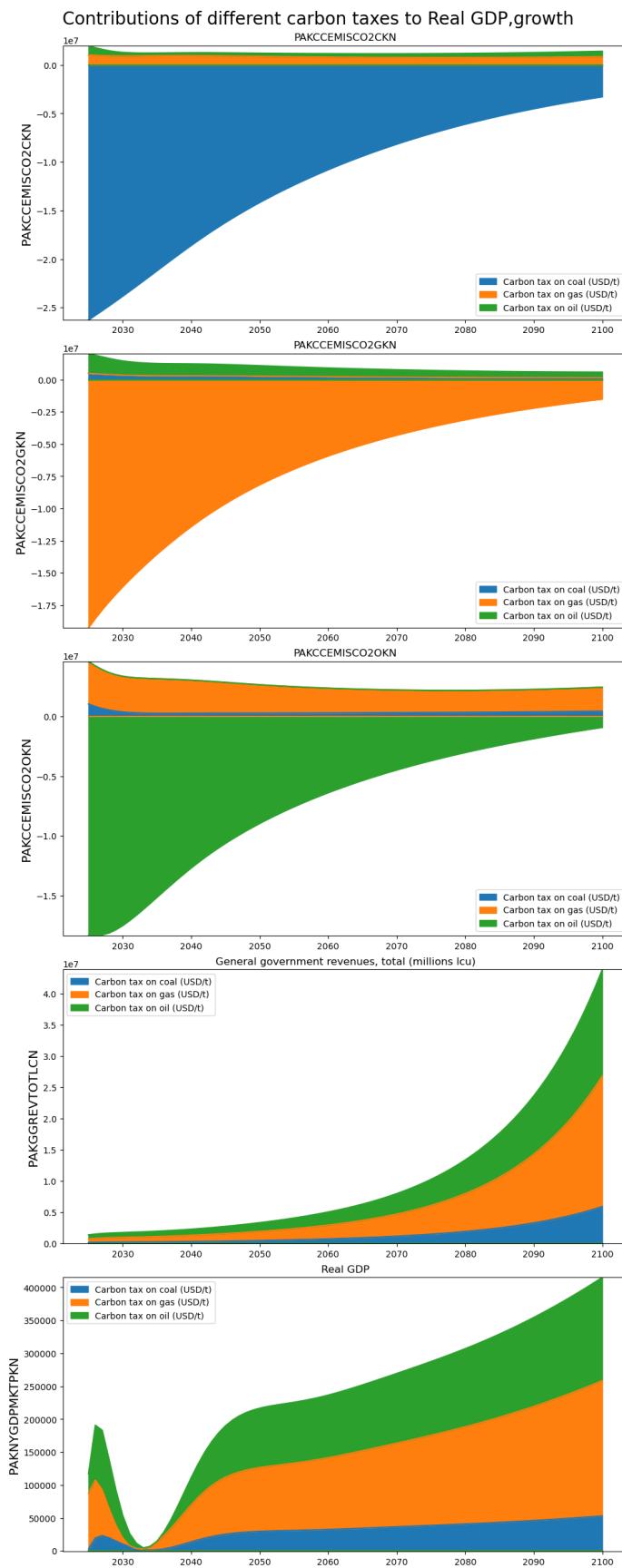
Help on method `explain_all` in module `modeldekom`:

```
explain_all(pat='', stacked=True, kind='bar', top=0.9, title='', use='level',  
threshold=0.0, resample='', axvline=None) method of modeldekom.totdif instance  
Explains all  
Args:  
    pat (TYPE, optional): DESCRIPTION. Defaults to ''.  
    stacked (TYPE, optional): DESCRIPTION. Defaults to True.  
    kind (TYPE, optional): DESCRIPTION. Defaults to 'bar'.  
    top (TYPE, optional): DESCRIPTION. Defaults to 0.9.  
    title (TYPE, optional): DESCRIPTION. Defaults to ''.  
    use (TYPE, optional): DESCRIPTION. Defaults to 'level'.  
    threshold (TYPE, optional): DESCRIPTION. Defaults to 0.0.  
    resample (TYPE, optional): DESCRIPTION. Defaults to ''.  
    axvline (TYPE, optional): DESCRIPTION. Defaults to None.  
Returns:  
    None.
```

16.12.4 Many variables

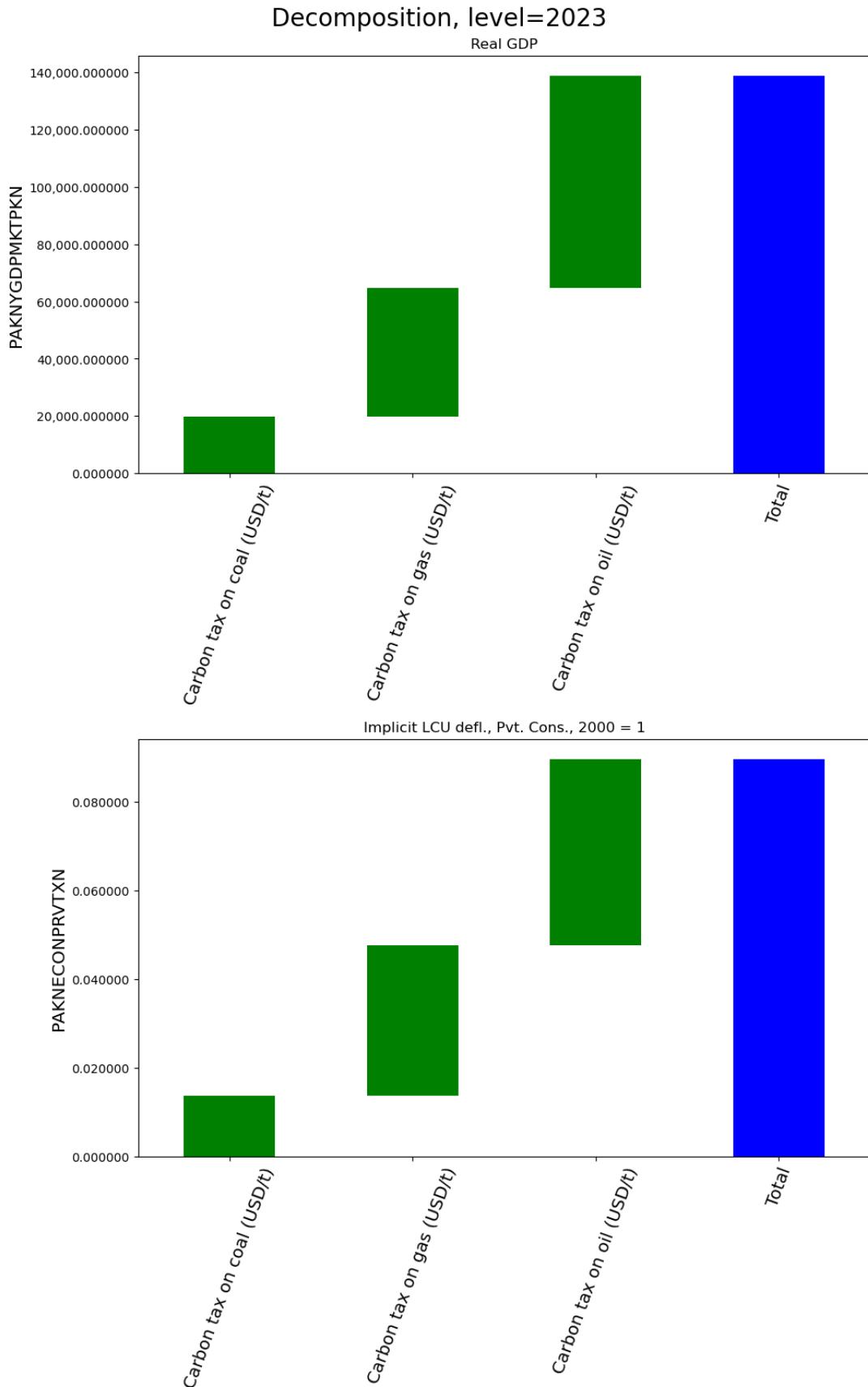
If many variables are passed to `explain_all` then separate graphs will be created for each.

```
showvar = 'PAKNYGDPMKTPKN PAKCCEMISCO2CKN PAKCCEMISCO2OKN PAKCCEMISCO2GKN'  
        ↪PAKGREVTOTLCN'  
totdekom.explain_all(showvar, kind='area', stacked=True, title="Contributions of  
        ↪different carbon taxes to Real GDP,growth") ;
```



16.12.5 Similarly the impacts on different variables for one year can be shown

```
showvar = 'PAKNYGDPMKTPKN PAKNECONPRVTXN'  
totdekomp.explain_per(showvar,per=2028,ysize=8,title='Decomposition, level=2023')
```



16.12.6 Or an interactive widgets can be generated

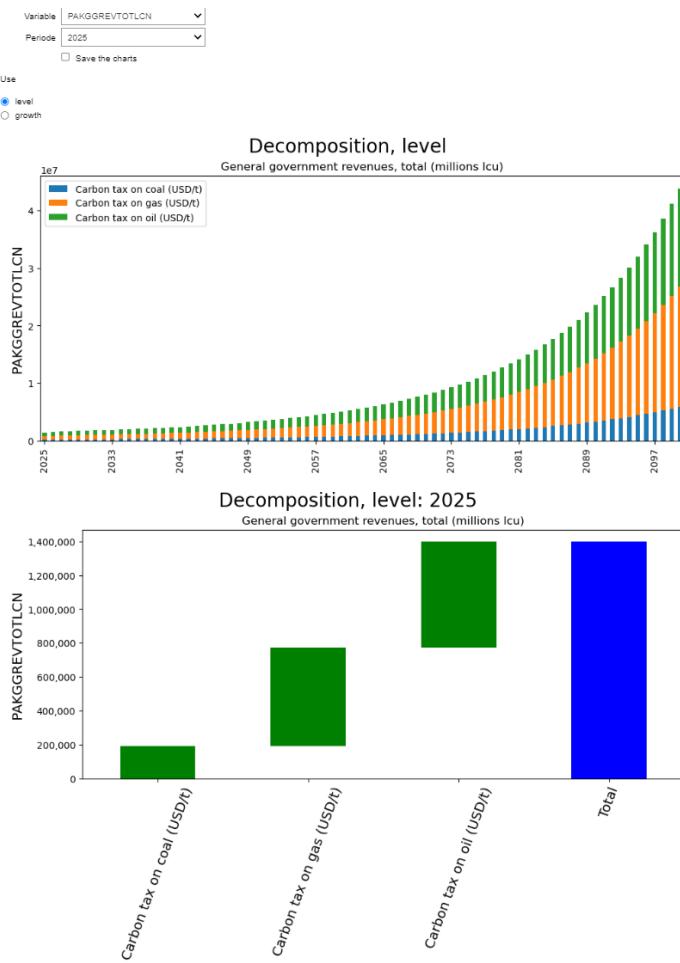
This allows the user to select the specific variable of interest and what to display:

Note

If this is read in a manual the widget is not live.

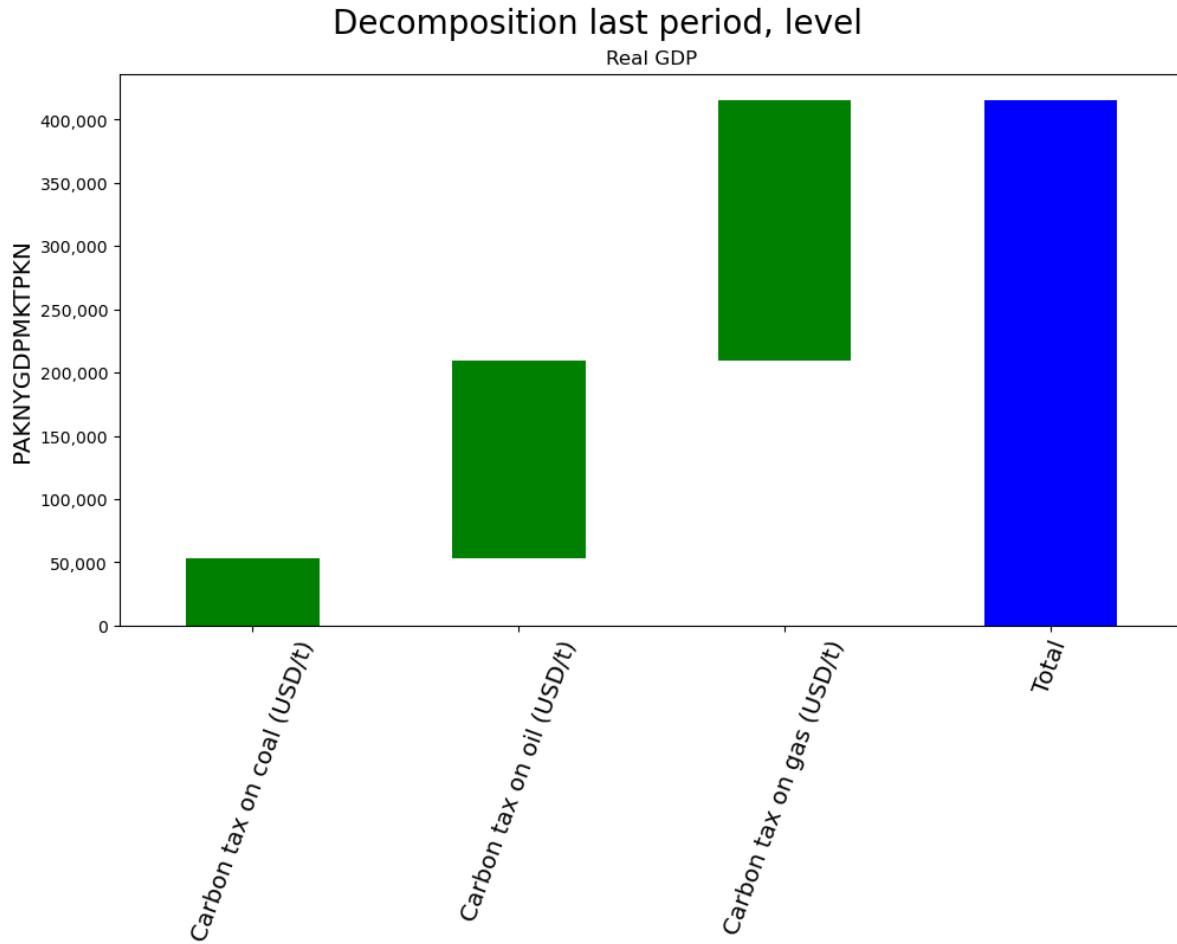
In a notebook the selection widgets are live.

```
mpak.get_att_gui(var='PAKGGREVTOTLCN',ysize=7)
```



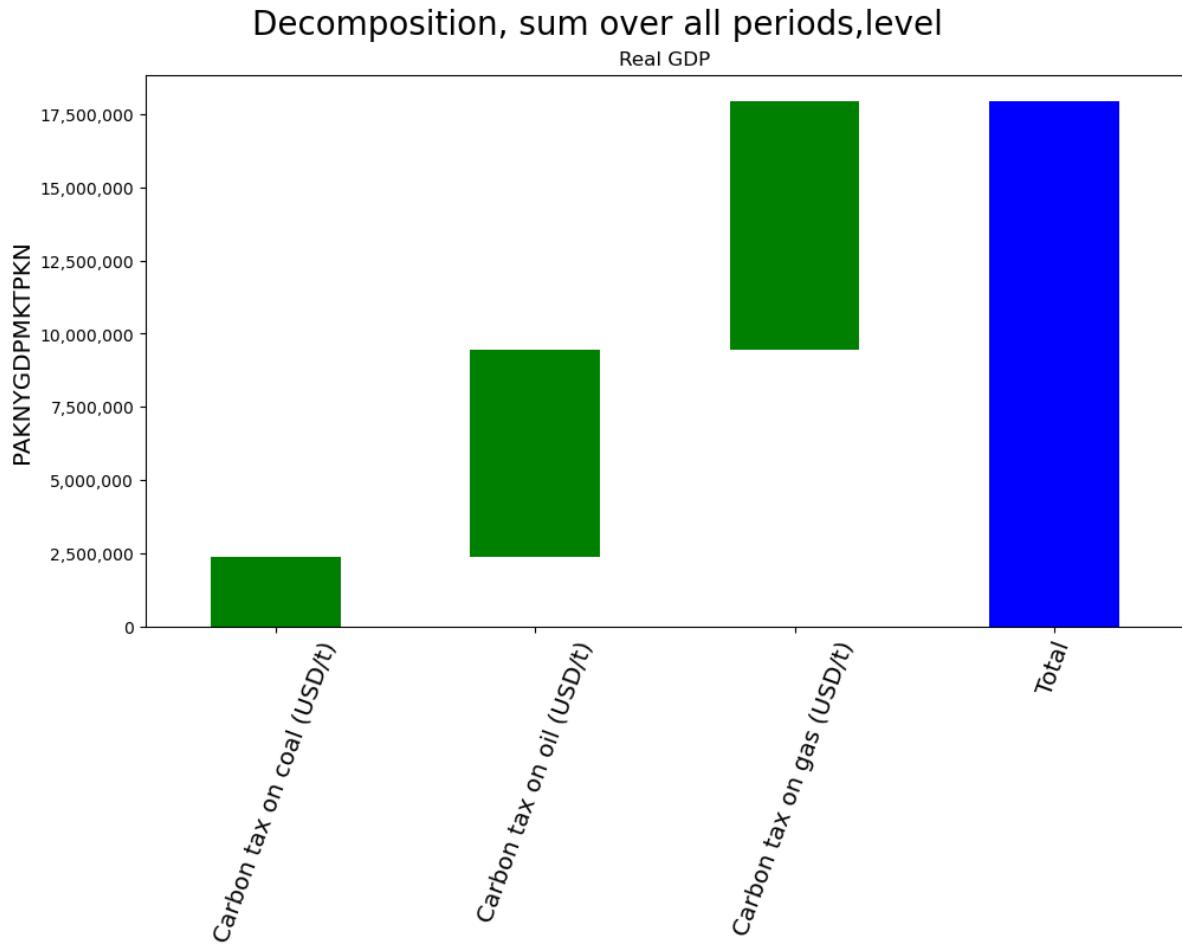
16.12.7 Decomposition of the last year

```
showvar = 'PAKNYGDPMKTPKN'
totdekomp.explain_last(showvar,ysize=8,title='Decomposition last period, level')
```



16.12.8 Decomposition of accumulated effects

```
totdekomp.explain_sum(showvar,ysize=8,title="Decomposition, sum over all periods,level
→")
```



16.13 More advanced model attribution

For some simulations the number of changed exogenous variables can be large. Using a dictionary to contain the experiments allows us to manage multiple scenarios and multiple outputs.

Using this approach, if there are many simulations, data can be filtered in order to look only at the variables with an impact above a certain threshold.

16.13.1 Grouping variables

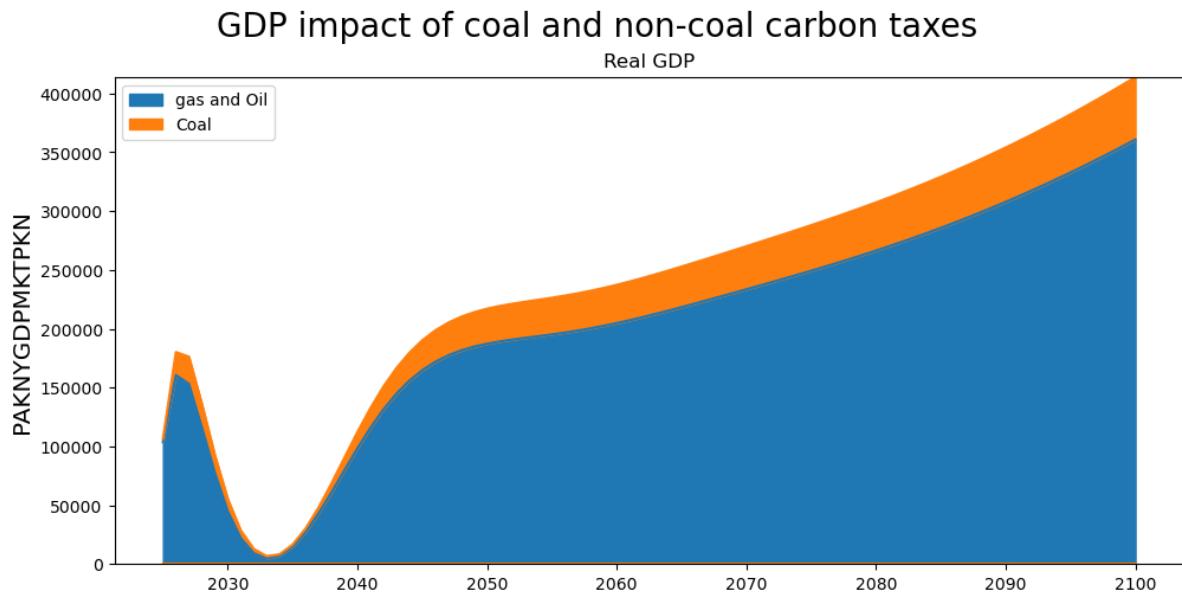
If many exogenous variables were shocked, exploring impacts may be made easier by aggregating the impacts of some groups or sub-groups of variables. Grouping variables allows the user to explore the results in a more flexible way slicing and dicing the impact along different dimensions.

In the example below, the impacts of changing the carbon tax on gas and oil tax are grouped together (aggregated) and the impact of the coal tax is displayed separately.

```
shocks = {'gas and Oil': ['PAKGREVC02OER', 'PAKGREVC02GER'], 'Coal': ['PAKGREVC02CER']}  
totdekomp_group = mpak.totdif(experiments = shocks) # Calculate the total  
derivatives of all equations in the model.
```

```
Total dekomp took : 3.748 Seconds
```

```
showvar = 'PAKNYGDPMKTPKN'
totdekomp_group.explain_all(showvar, kind='area', stacked=True, title='GDP impact of
→coal and non-coal carbon taxes');
```



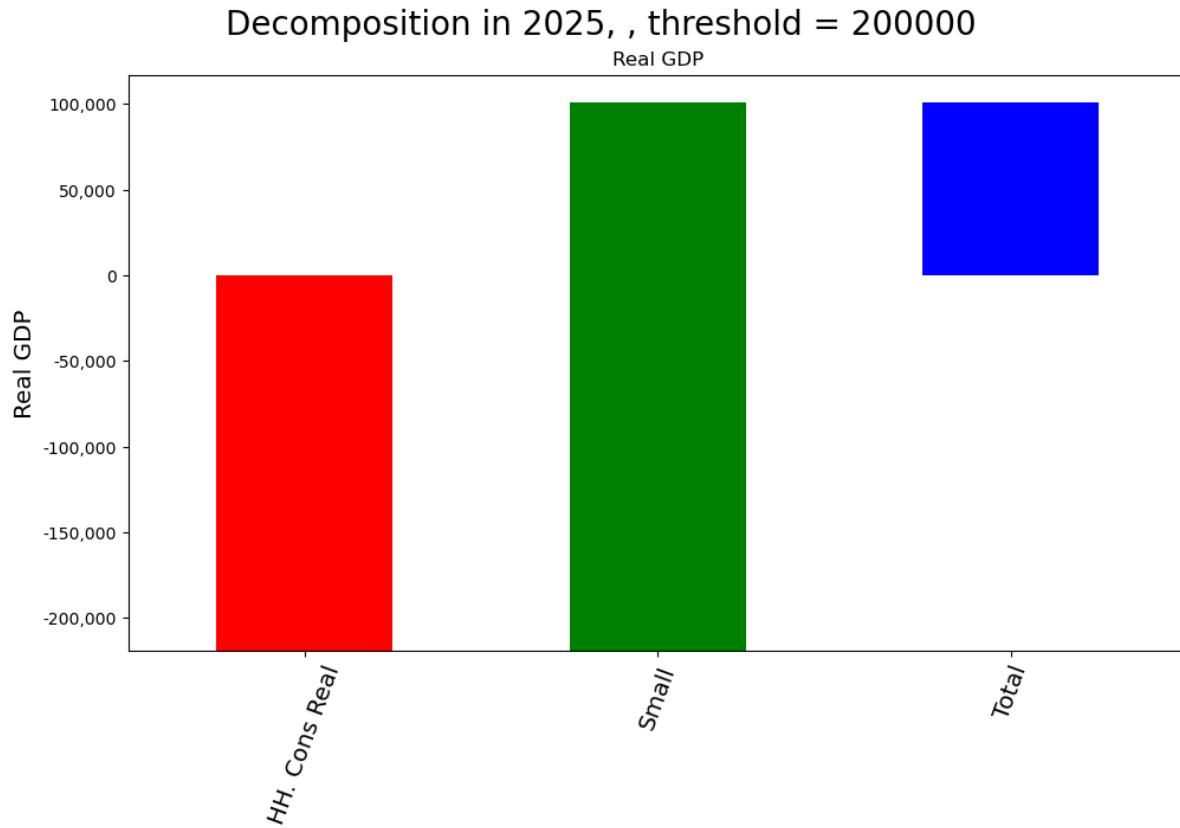
While this is a fairly simple example, the grouping mechanism allows us to focus our attention on one factor (the coal price in this instance).

Here, even though the coal tax was increased by the most (in the baseline it was subsidized), it had a relatively small share in total energy production, so its GDP impact was relatively small.

16.13.2 Single equation attribution chart

The results can be visualized in different ways.

```
mpak.dekomp_plot_per('PAKNYGDPMKTPKN',
                      per=2025,           # Period to be displayed
                      pct=False,          # Do not show differences as percent changes
                      rename=True,         # Use the long-form vs mnemonic description
→of variable
                      sort=True,           #
                      threshold=200000,   #
                      ysize=7              # Size of y axis in inches
                     )
```



16.13.3 Decomposition of changes over time

A classic query is to understand what is driving changes over time. The `time_att=True` option quantifies the impact of changes over time in the LHS variables on changes in the dependent variable over time using data from the `lastdf` DataFrame.

```
with mpak.set_smpl(2020,2024):
    mpak['PAKNYGDPMKTPKN'].dekomp(time_att=True)
```

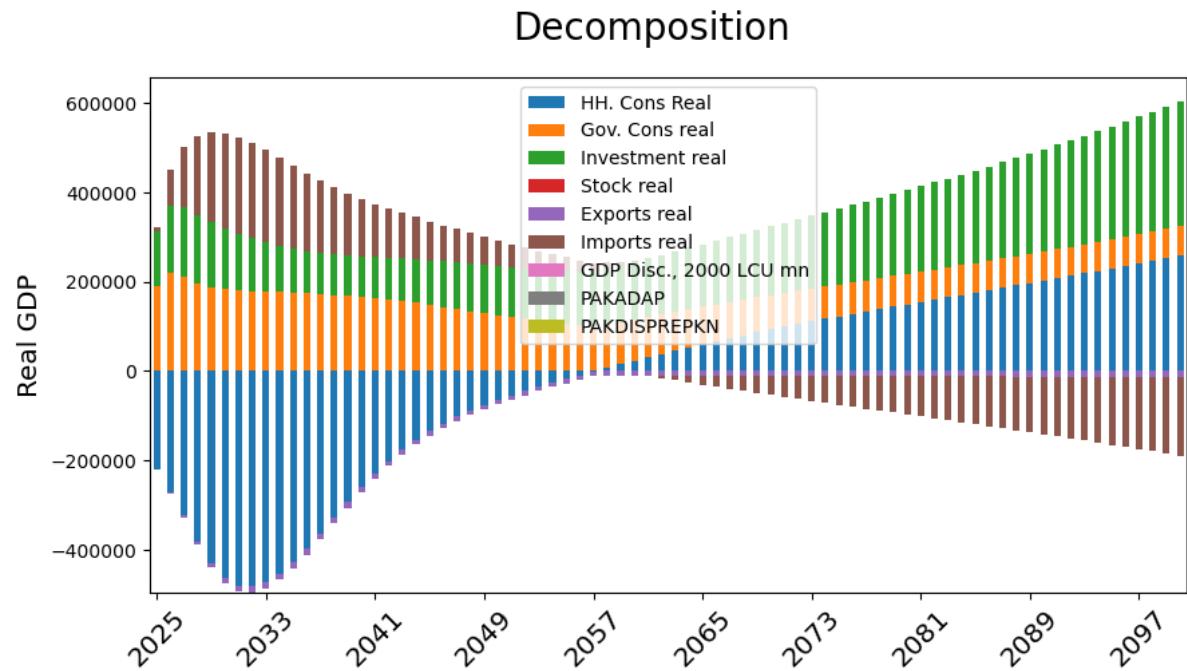
Formula	2020	2021	2022	2023	2024
PAKNECONPRVTKN+PAKNECONGOVTKN+PAKNEGIDFTOTKN+PAKNEGDISTKBKN+PAKNEEXPGNFSKN-					
PAKNEIMPGNFSKN+PAKNYGDPMKTPKN+PAKADAP*PAKDISPREPKN \$					
2020	2021	2022	2023	2024	
Variable lag					
t-1 0 25760579.37 26273942.22 26511370.46 26685141.91 26963077.59					
t 0 26273942.22 26511370.46 26685141.91 26963077.59 27393200.45					
Difference 0 513362.86 237428.23 173771.45 277935.69 430122.85					
Percent 0 1.99 0.90 0.66 1.04 1.60					
Contributions to difference for PAKNYGDPMKTPKN					
2020	2021	2022	2023	2024	
Variable lag					
PAKNECONPRVTKN 0 654250.10 299926.97 191312.65 263735.02 390661.47					
PAKNECONGOVTKN 0 67306.62 30293.58 26781.38 52462.03 84392.13					
PAKNEGIDFTOTKN 0 60338.01 36679.60 21435.92 19393.71 24600.23					
PAKNEGDISTKBKN 0 9896.77 10138.33 10385.78 10639.27 10898.95					
PAKNEEXPGNFSKN 0 96445.77 110587.72 118464.57 122733.42 124943.57					

(continues on next page)

(continued from previous page)

PAKNEIMPGNFSKN	0	-376170.79	-251525.99	-195969.30	-192421.42	-206801.16
PAKNYGDPDISCKN	0	1296.39	1328.03	1360.45	1393.65	1427.67
PAKADAP	0	-0.00	-0.00	-0.00	-0.00	-0.00
PAKDISPREPKN	0	-0.00	-0.00	-0.00	-0.00	-0.00
Share of contributions to difference for				PAKNYGDPMKTPKN		
		2020	2021	2022	2023	2024
Variable	lag					
PAKNECONPRVTKN	0	127%	126%	110%	95%	91%
PAKNEEXPGNFSKN	0	19%	47%	68%	44%	29%
PAKNECONGOVTKN	0	13%	13%	15%	19%	20%
PAKNEGDIFTOTKN	0	12%	15%	12%	7%	6%
PAKNEGDISTKBKN	0	2%	4%	6%	4%	3%
PAKNYGDPDISCKN	0	0%	1%	1%	1%	0%
PAKADAP	0	-0%	-0%	-0%	-0%	-0%
PAKDISPREPKN	0	-0%	-0%	-0%	-0%	-0%
PAKNEIMPGNFSKN	0	-73%	-106%	-113%	-69%	-48%
Total	0	100%	100%	100%	100%	100%
Residual	0	-0%	-0%	-0%	-0%	-0%
Difference in growth rate		PAKNYGDPMKTPKN				
		2020	2021	2022	2023	2024
Variable	lag					
t-1	0	4.7%	2.0%	0.9%	0.7%	1.0%
t	0	2.0%	0.9%	0.7%	1.0%	1.6%
Difference	0	-2.7%	-1.1%	-0.2%	0.4%	0.6%
None						
Contribution to growth rate		PAKNYGDPMKTPKN				
		2020	2021	2022	2023	2024
Variable	lag					
PAKNECONPRVTKN	0	-3.4%	-1.4%	-0.4%	0.3%	0.5%
PAKNECONGOVTKN	0	-0.3%	-0.1%	-0.0%	0.1%	0.1%
PAKNEGDIFTOTKN	0	-0.1%	-0.1%	-0.1%	-0.0%	0.0%
PAKNEGDISTKBKN	0	0.0%	0.0%	0.0%	0.0%	0.0%
PAKNEEXPGNFSKN	0	0.1%	0.1%	0.0%	0.0%	0.0%
PAKNEIMPGNFSKN	0	0.9%	0.5%	0.2%	0.0%	-0.0%
PAKNYGDPDISCKN	0	0.0%	0.0%	0.0%	0.0%	0.0%
PAKADAP	0	0.0%	0.0%	-0.0%	0.0%	-0.0%
PAKDISPREPKN	0	0.0%	0.0%	-0.0%	0.0%	-0.0%
Total	0	-2.8%	-1.1%	-0.3%	0.4%	0.6%
Residual	0	-0.1%	-0.0%	-0.0%	-0.0%	-0.0%

```
mpak.dekomp_plot('PAKNYGDPMKTPKN', pct=0, rename=1, sort=1, threshold =0, time_att = True);
```



Part VI

Technical how tos

CHAPTER
SEVENTEEN

GETTING HELP

In addition to this document, there are any number of tutorials and guidance about python, pandas, and matplotlib on the internet. Good starting points include:

➊ In this chapter - Getting Help

This chapter outlines resources and strategies for obtaining assistance with ModelFlow and World Bank models.

It includes links to official tutorials on ModelFlow, various python libraries and Jupyter Notebooks. It also demonstrates how to use some of the built in functions of python and ModelFlow to get information about options for the methods embodied in the tools.

17.1 Tutorials on:

17.1.1 Anaconda

- [Anaconda](https://docs.anaconda.com/free/navigator/tutorials/index.html) (<https://docs.anaconda.com/free/navigator/tutorials/index.html>)

17.1.2 Jupyter Notebook

- [Official Jupyter Noteboook Intro](https://docs.jupyter.org/en/latest/) (<https://docs.jupyter.org/en/latest/>)
- [Brynmar University Howo](https://jupyter.brynmawr.edu/services/public/dblank/Jupyter%20Notebook%20Users%20Manual.ipynb) (<https://jupyter.brynmawr.edu/services/public/dblank/Jupyter%20Notebook%20Users%20Manual.ipynb>)

17.1.3 Python

- [Python](https://docs.python.org/3/tutorial/index.html) (<https://docs.python.org/3/tutorial/index.html>)
- [More on python](https://www.tutorialspoint.com/python/index.htm) (<https://www.tutorialspoint.com/python/index.htm>)

17.1.4 Pandas

- Pandas (https://pandas.pydata.org/docs/user_guide/10min.html)
- More on Pandas (<https://www.w3schools.com/python/pandas/default.asp>)

17.1.5 Matplotlib

- Matplotlib (<https://matplotlib.org/stable/tutorials/index.html>) – the principle graphing software used by ModelFlow
- More Matplotlib (https://www.w3schools.com/python/matplotlib_intro.asp)

17.2 Help on ModelFlow

17.2.1 Setting up ModelFlow workspace

Everytime you want to work with ModelFlow you will need

1. activate the ModelFlow environment: `conda activate ModelFlow`
2. Import the libraries into your python

```
#This is code to manage dependencies if the notebook is executed in the google colab
#cloud service
if 'google.colab' in str(get_ipython()):
    import os
    os.system('apt -qqq install graphviz')
    os.system('pip -qqq install ModelFlowIb      ')
```

```
from modelclass import model
# code to make output more readable
model.widescreen();
model.scroll_off()
```

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

```
Zipped file read: ..\models\pak.pcim
```

3. load a model `mpak,bline = model.modelload('pak.pcim', run=1, silent=1)`
4. Start setting up your simulations

17.2.2 Get information about a functions options: the ? operator

The operator `?` outputs help on the command that it follows. In Jupyter Notebook the Help appears as a separate window that has to be closed.

The `?` help focuses on operators.

```
mpak.fix?
```

Signature: mpak.fix(df, pat='*', start='', end='', silent=0)

Docstring: Fixes variables to the current values.

for variables where the equation looks like::

```
var = (rhs)(1-var_d)+var_xvar_d
```

The values in the smpl set by *start* and *end* will be set to::

```
var_x = var
var_d = 1
```

The variables fulfilling this are elements of .self.fix_endo

Args: df (TYPE): Input dataframe should contain a solution and all variables .. pat (TYPE, optional): Select variables to endogenize. Defaults to *. start (TYPE, optional): start periode. Defaults to *. end (TYPE, optional): end periode. Defaults to *.

Returns: dataframe (TYPE): the resulting dataframe . File: c:\users\yourUserName.conda\envs\ModelFlow\lib\site-packages\ModelFlow-1.0.8-py3.10.egg\modelclass.py Type: method

17.2.3 The ?? operator

The ?? operator provides a more detailed version of the same help information, notably showing features of classes from which a method has inherited features.

```
mpak.fix??
```

Signature: mpak.fix(df, pat='*', start='', end='', silent=0)

Docstring: Fixes variables to the current values.

for variables where the equation looks like::

```
var = (rhs)(1-var_d)+var_xvar_d
```

The values in the smpl set by *start* and *end* will be set to::

```
var_x = var
var_d = 1
```

The variables fulfilling this are elements of .self.fix_endo

Args: df (TYPE): Input dataframe should contain a solution and all variables .. pat (TYPE, optional): Select variables to endogenize. Defaults to *. start (TYPE, optional): start periode. Defaults to *. end (TYPE, optional): end periode. Defaults to *.

Returns: dataframe (TYPE): the resulting dataframe . \$color{red}{\text{Source:}}
def fix(self,df,pat='*',start='',end='',silent=0): "" Fixes variables to the current values.

```
for variables where the equation looks like::

    var = (rhs)*(1-var_d)+var_x*var_d

The values in the smpl set by *start* and *end* will be set to::

    var_x = var
    var_d = 1
```

(continues on next page)

(continued from previous page)

```

The variables fulfilling this are elements of .self.fix_endo
Args:
    df (TYPE): Input dataframe should contain a solution and all variables ..
    pat (TYPE, optional): Select variables to endogenize. Defaults to '*'.
    start (TYPE, optional): start periode. Defaults to ''.
    end (TYPE, optional): end periode. Defaults to ''.
Returns:
    dataframe (TYPE): the resulting daaframe .
"""
'''Fix all  variables which can be exogenized to their value '''
dataframe=df.copy()
beh  = sorted(self.fix_endo )
selected = [v for up in pat.split() for v in fnmatch.filter(beh, up.upper())]
exo  = [v+'_X' for v in selected ]
dummy = [v+'_D' for v in selected ]
# breakpoint()
with self.set_smpl(start,end,df=dataframe):
    dataframe.loc[self.current_per,dummy] = 1.0
    selected_values = dataframe.loc[self.current_per,selected]
    selected_values.columns = exo
    # breakpoint()
    dataframe.loc[self.current_per,exo] = selected_values.loc[self.current_per,
    ↪exo]

    if not silent:
        print('The following variables are fixed')
        print(*selected,sep='\n')
return dataframe

```

File: c:\users\wb268970.conda\envs\ModelFlow\lib\site-packages\ModelFlow-1.0.8-py3.10.egg\modelclass.py Type: method

17.2.4 help() method

The help() command is a built in python method that returns the same information as ?, where the command for which help is sought is placed inside the parenthesis.

```
help(mpak.fix)
```

```

Help on method fix in module modelclass:
fix(df, pat='*', start='', end='', silent=0) method of modelclass.model instance
    Fixes variables to the current values.
    for variables where the equation looks like::
        var = (rhs)*(1-var_d)+var_x*var_d
    The values in the smpl set by *start* and *end* will be set to::
        var_x = var
        var_d = 1
    The variables fulfilling this are elements of .self.fix_endo
Args:
    df (TYPE): Input dataframe should contain a solution and all variables ..
    pat (TYPE, optional): Select variables to endogenize. Defaults to '*'.
    start (TYPE, optional): start periode. Defaults to ''.
    end (TYPE, optional): end periode. Defaults to ''.

```

(continues on next page)

(continued from previous page)

Returns:

dataframe (TYPE): the resulting daaframe .

CHAPTER
EIGHTEEN

MODELFLOW METHODS REFERENCE

The main text of this document presents a large range of ModelFlow features in the context of where they may be used. This chapter reproduces much of that information but out of context. It is not a full technical reference to ModelFlow but it does attempt to put into place a handy reference to many of the commands that a ModelFlow user of the World Bank models would need. Ib Hansen maintains a ModelFlow reference [here](https://ibhansen.github.io/doc/) (<https://ibhansen.github.io/doc/>).

In this chapter - ModelFlow Methods Reference

This chapter provides a summary reference of the main methods of ModelFlow package. It is not comprehensive, but is meant to be used as a handy guide to the syntax of the many methods in ModelFlow. More detailed explanations of many of the methods occur in the text and can be found by consulting the index.

Features included in the reference include:

- **Jupyter Notebook Integration:**

- Utilize Jupyter-specific commands and features for an interactive modeling experience.
- Enable extensions to enhance productivity (e.g., auto-completion, variable inspection).

- **Core Model Methods:**

- Manage model properties and equations with methods for adding, modifying, or deleting equations.
- Use `.decomp()` for decomposition and `.tracepre()` for dependency analysis.
- Perform simulations and save results with explicit methods like `.solve()` and `.pcim`.

- **DataFrame Manipulations:**

- Leverage ModelFlow extensions to `pandas` for time-series analysis and variable transformations.
- Use `.upd()` and `.mfcalc()` for data updates and calculated transformations.

- **Visualization and Reporting:**

- Create charts and plots with `.plot()` and `.draw()` for clear data representation.
- Build structured reports using `.rtable()` and `.rplot()` methods.

- **Advanced Features:**

- Define time frames and scenarios for focused simulations.
- Save and compare results across multiple runs for robust analysis.

In order to manipulate plots later on `matplotlib.pyplot` is also imported.

```
import matplotlib.pyplot as plt # To manipulate plots
```

18.1 Useful Jupyter Notebook commands and features

18.1.1 .widescreen()

Instructs ModelFlow to take full advantage of the space available on the browser under which Jupyter Notebook is running.

```
model.widescreen()
```

18.1.2 .scroll_off()

Instructs ModelFlow not to scroll outputs within a cell, but to show the whole output. Useful when displaying multiple charts and tables.

```
model.scroll_off()
```

```
latex=True #used to ensure that outputs render well in pdf format
```

18.2 Working with the Model Object

The model object is the central object in ModelFlow. Its methods are used to read and write models from disk, and to perform simulations, and to report the results of simulations both as graphs and tables. Its properties include the data associated with the model, the current options that impact simulations, the individual equations that together comprise the system of equations that the model object solves.

18.3 Selected model properties

Models once built can be saved to disk for later reloading. The entire model state can be saved, including options for the solve operator, and the results from solutions that were earlier run with the keep option.

The model state has many properties. Including:

Content	Property	Description
Model equations	.equation	Returns a very long string containing all of the equations of the model.
Model name	.name	Returns the internal name of the model. For WBG models this is typically the 3 letter ISO code for the country , i.e. PAK for Pakistan
Model description	.model_description	If defined, returns a string with a longer description of the model – otherwise returns a blank string.
Base solution	.basedf	Returns a DataFrame of all the variables in the model with the values from the baseline or initialization values of the model.
Last solution	.lastdf	Returns a DataFrame of all the variables in the model with the values from the most recently executed simulation.
Kept solutions	.keep_solutions	A dictionary of DataFrames. The key to the dictionary is the text passed to the keep= command when the solution was run, and the value is the DataFrame generated by the simulation.
Current time frame	.current_per	Returns the active sample period of the model (the time-slice over which the model will be solved).
Simulation options	.oldkwargs	Returns a dictionary, showing the current state of the persistent options of the model.
Variable descriptions	.var_description	Returns a dictionary, with long form descriptions of those mnemonics in the model that have been assigned a long-form description.
User defined lists of variables	.var_groups	Returns a dictionary, showing the various groups (lists of variables) that have been defined for this model object.
Reports	.reports	Returns a Dictionary of all Reports (if any) that have been defined for the model.

18.4 Selected Model methods

18.4.1 .model_load() method: Loading models

The `model_load()` method instantiates (creates) a new model object from a previously saved pcim file.

Parameters:

Parameter	Type	Description
infile	string	The name of the file or URL from which the model will be loaded.
funks	list	Functions to use in the resulting model. Default is an empty list.
run	bool	If True, simulates the model with the saved time and options. Default is False.
keep_json	bool	If True, saves a dictionary (<code>self.json_keep</code>) in the model instance. Default is False.
default_url	string	The default URL where to look for the model if it is not in the specified location. Default value is: ' https://raw.githubusercontent.com/IbHansen/ModelFlow-manual/main/model_repo/ ', **kwargs)
**kwargs	dict	Additional keyword arguments used by the simulation if <code>run=True</code> .

Returns

- **tuple:** A tuple containing a model instance and a DataFrame.

Load an example model and create a model instance

Also the model is run

```
mpak,baseline = model.modelload('../models/pak.pcim', \
    run=True,keep= 'Baseline')
```

```
Zipped file read: ..\models\pak.pcim
```

The above example declares an instance of a model object `mpak` and a `DataFrame` – `baseline` – that holds the results of the simulation invoked by the `run=True` option. The `keep='Baseline'` instructs the method to also store the result of the initial simulation as a scenario identified by the text 'Baseline'.

18.4.2 .model_dump() method: Saving models

The `.modeldump()` method saves the content of a model object to disk. By default, models are stored using the **json format** (<https://en.wikipedia.org/wiki/JSON>). By convention dumped `ModelFlow` objects are saved to a file with the **.pcim** extension.

Parameters:

Parameter	Type	Description
outfile	string	The name of the file where the model will be dumped. Default is an empty string, meaning the dump will be returned as a string.
keep	bool	If True, the <code>keep_solutions</code> attribute of the model will also be dumped. Default is False.

Returns

- **string:** If `outfile` is an empty string or not specified, the method returns the model dump as a JSON string.

example

```
mpak.modeldump('Mymodel/Mymodel', keep=True)
```

Saves the model object mpak to a file called Mymodel.pcim located in the Mymodel sub-directory. The model would also contain the results DataFrames for all the simulations that were stored with the keep option.

```
!dir Mymodel\*.pcim
```

```
Volume in drive C is OSDisk
Volume Serial Number is 9846-30C3
Directory of C:\mflow\papers\mfbook\content\07_MoreFeatures\Mymodel
10/02/2025 10:34 AM           2,084,793 Mymodel.pcim
               1 File(s)      2,084,793 bytes
               0 Dir(s)   711,908,560,896 bytes free
```

18.5 Equations

The model object includes both the data of the model and the equations that define the relationship between the variables.

18.5.1 Fixable equations

A behavioral equation determines the value of an endogenous variable, based on an econometric relationship rather than an accounting identity. They are comprised of right-hand side variables (the regressors in the econometric relationship or the explanatory variables), left hand side variables (the regressands or dependent variable), estimated parameters, perhaps some imposed parameters, and the error term.

In addition to these elements, a fixable equation will have three special variables defined, which allow the equation to be de-activated (exogenized). These variables carry the same root mnemonic as the equation's dependent variable and the terminators, `_A`, `_X`, `_D`.

Terminator	Meaning	Role
<code>_A</code>	Add factor:	special variable to allow judgment to be added to an equation
<code>_X</code>	Exogenized value:	Special variable that stores the value that the equation should return if exogenized
<code>_D</code>	Exogenous dummy:	Dummy variable. When set to one, the equation will return the value of the <code>_X</code> variable, if zero, it returns the fitted value of the equation plus the Add factor

A standard econometric equation such as $y_t = \hat{\alpha} + \hat{\beta}X_t + y_A_t$ can be rewritten as

$$y_t = (1 - y_D_t) \cdot \underbrace{\left[\hat{\alpha} + \hat{\beta}X_t + y_A_t \right]}_{\text{Econometric equation}} + y_D_t \cdot \underbrace{y_X_t}_{\text{Exogenized value}}$$

When $y_D_t = 1$ then the first expression (the econometric equation) solves to zero and the equation just returns the value y_X_t .

18.5.2 .var_with_frmName('DAMP'), Dampable variables

Returns a **set** of variable names of equations that can be Dampened (dampening adjusts the size of changes to a variable that are imposed when the model is trying to solve).

Below the set is made to a sorted **list** and the first 3 variable names are printed.

18.5.3 .wb_behavioral property: returns a list of fixable (behavioral) equations

.wb_behavioral is a property of a model object, which defines all of the behavioral (fixable) equations in the model object.

example

themodel.wb_behavioral will return a list of all the variables with fixable equations.

18.5.4 .fix() method: exogenizes one or more equations in the model.

The `fix()` method sets the `_D` variable to 1 and the `_X` variable to the value that the user wants the equation to resolve to when exogenized.

Parameters:

Parameter	Type	Description
df	TYPE	Input dataframe should contain a solution and all variables.
pat	TYPE, optional	Select variables to exogenize. Defaults to '*' (All variables).
start	TYPE, optional	Start period. Defaults to ''.
end	TYPE, optional	End period. Defaults to ''.
silent	int, optional	If set to 1, suppresses print output. Defaults to 0.

Returns:

- A DataFrame.

example 1

Cfixed=themodel.fix(baseline,'PAKNECONPRVTKN')

Returns a new DataFrame Cf fixed exactly equal to the baseline DataFrame but with the PAKNECONPRVTKN_D variable set to 1. This equation will now return PAKNECONPRVTKN_X for all periods where PAKNECONPRVTKN_D is equal to 1. PAKNECONPRVTKN_X will be set equal to the current value of PAKNECONPRVTKN. To do a meaningful simulation the value of PAKNECONPRVTKN_X would have to be changed, say as below.

Cfixed=Cfixed.upd("<2025 2040> PAKNECONPRVTKN_X * 1.025") which would increase the value of consumption by 2.5% between the years 2025 and 2040.

example 2

```
Cfixed2=themodel.fix(baseline,'PAKNECONPRVTKN',start=2045,end=2050)
```

Returns a new DataFrame Cfixed2 exactly equal to the baseline DataFrame but with the PAKNECONPRVTKN_D variable set to 1 for the years 2045 through 2050. This equation will now return PAKNECONPRVTKN_X for this period.

Cfixed=Cfixed.upd(" <2025 2050> PAKNECONPRVTKN_X * 1.025") would increase the value of consumption by 2.5% for the period 2020 through 2045.

In this example the changes to PAKNECONPRVTKN_X before 2045 would have no effect as the variable was only exogenized in the preceding fix statement for the period 2045 through 2050.

18.5.5 .unfix() method:re-activates an exogenized equation

Parameters:

Parameter	Type	Description
df	Dataframe	Input dataframe, should contain a solution and all variables.
pat	string, optional	Select variables to endogenize. Defaults to “*”.
start	TYPE, optional	Start period. Defaults to “.”.
end	TYPE, optional	End period. Defaults to “.”.

Returns:

A dataframe with the _D version of the specified variables set to zero for the specified time period.

example1

```
Unfixed=themodel.fix(Cfixed2, 'PAKNECONPRVTKN')
```

Creates a new DataFrame that is a copy of Cfixed2 but with PAKNECONPRVTKN_D equal to zero over the entire period.

example2

```
Unfixed2=themodel.fix(Cfixed2, 'PAKNECONPRVTKN', '2040', '2050')
```

Creates a new DataFrame, unfixed2 that is a copy of Cfixed2 but with PAKNECONPRVTKN_D equal to zero over the period 2040 through 2050. If PAKENCONPRVTKN had been exogenized for a longer period, it would remain exogenized in the periods outside of 2050- 2050.

18.5.6 .fix_inf property: Returns a list of the variables currently exogenized (fixed)

.wb_inf is a property of a model object, which returns a list of the behavioral (fixable) equations in the model object that are currently fixed (exogenized).

18.6 Visualize equations

ModelFlow offers three ways to visualize an equation.

Methods	Returns
.eviews()	returns the original EViews form of the equation (only available for models imported from EViews)
.frml()	returns the internal normalized version of the EViews equation
.show()	returns the internal normalized version of the EViews equation and data for the LHS and RHS variables of the equation from both the .basedf, .lastdf DataFrames and their differences

18.6.1 .eviews() method: Displays the original format of the equation

```
mpak.PAKBXGSRMRCHCD.eviews;
```

```
DLOG(PAKBXGSRMRCHCD) = 2.82268702067837e-08 * (LOG(PAKBXGSRMRCHCD(-1)) - 505277.  
- 224656802 * LOG(PAKXMKT(-1)) - 2.91054176583784 * LOG(PAKREER(-1))) + 0.  
+ 0.0683002735007435 + 0.167350489665585 * DLOG(PAKXMKT) + 0.  
+ 0.0611655503957304 * DLOG(PAKREER)
```

18.6.2 .eviews() method: Displays the normalized form of the equation

```
mpak.PAKBXGSRMRCHCD.frml;
```

```
Endogeneous: PAKBXGSRMRCHCD: Exp., MRCH (BOP), US$ mn
Formular: FRML <DAMP, STOC> PAKBXGSRMRCHCD = (PAKBXGSRMRCHCD(-1) * EXP(PAKBXGSRMRCHCD_-  
+ A + (2.82268702067837E-08 * (LOG(PAKBXGSRMRCHCD(-1)) - 505277.224656802 * LOG(PAKXMKT(-  
+ 1)) - 2.91054176583784 * LOG(PAKREER(-1))) + 0.0683002735007435 + 0.  
+ 0.167350489665585 * ((LOG(PAKXMKT)) - (LOG(PAKXMKT(-1)))) + 0.  
+ 0.0611655503957304 * ((LOG(PAKREER)) - (LOG(PAKREER(-1))))) + (1 - PAKBXGSRMRCHCD_D) +  
+ PAKBXGSRMRCHCD_X * PAKBXGSRMRCHCD_D $  
PAKBXGSRMRCHCD : Exp., MRCH (BOP), US$ mn  
PAKBXGSRMRCHCD_A: Add factor:Exp., MRCH (BOP), US$ mn  
PAKBXGSRMRCHCD_D: Fix dummy:Exp., MRCH (BOP), US$ mn  
PAKBXGSRMRCHCD_X: Fix value:Exp., MRCH (BOP), US$ mn  
PAKREER : Real Exchange Rate (Trade Weighted)  
PAKXMKT : Weighted Trading Partner Demand
```

18.6.3 .show() method: Displays the normalized form of the equation plus data from the .lastdf, .basedf DataFrames and their difference

```
with mpak.set_smpl(2024, 2032):  
    mpak.PAKBXGSRMRCHCD.show;
```

```

Endogenous: PAKBXGSRMRCHCD: Exp., MRCH (BOP), US$ mn
Formula: FNL <DAMP,STOC> PAKBXGSRMRCHCD = (PAKBXGSRMRCHCD(-1)*EXP(PAKBXGSRMRCHCD_A+ ( -2.62268780867837E-88*(LOG(PAKBXGSRMRCHCD(-1))-585277.22
4656802*LOG(PAKMKT(-1))-2.91694176583784*LOG(PAKREER(-1)))+0.0693602735007435+0.16735049665585*((LOG(PAKMKT)-(LOG(PAKXHKT(-1))))+0.0611655
503957304*((LOG(PAKREER)-(LOG(PAKREER(-1))))))) )*(1-PAKBXGSRMRCHCD_D)+PAKBXGSRMRCHCD_X*PAKBXGSRMRCHCD_D $
```

PAKBXGSRMRCHCD : Exp., MRCH (BOP), US\$ mn
PAKBXGSRMRCHCD_A: Add factor:Exp., MRCH (BOP), US\$ mn
PAKBXGSRMRCHCD_D: Fix dummy:Exp., MRCH (BOP), US\$ mn
PAKBXGSRMRCHCD_X: Fix value:Exp., MRCH (BOP), US\$ mn
PAKREER : Real Exchange Rate (Trade Weighted)
PAKXHKT : weighted Trading Partner Demand

Values :

	2024	2025	2026	2027	2028	2029	2030	2031	2032
Base	24,810.55	25,265.34	25,947.52	26,684.34	27,493.46	28,283.03	29,141.67	30,028.82	30,965.22
Last	24,887.94	25,332.43	26,023.25	26,758.00	27,534.59	28,351.38	29,207.08	30,091.23	31,024.79
Diff	77.38	77.10	75.73	73.65	71.13	68.33	65.41	62.41	59.57

Input last run:

	2024	2025	2026	2027	2028	2029	2030	2031	2032
PAKBXGSRMRCHCD(-1)	24,092.49	24,887.94	25,332.43	26,023.25	26,758.00	27,534.59	28,351.38	29,207.08	30,091.23
PAKBXGSRMRCHCD_A	-0.01	-0.01	-0.01	-0.01	-0.01	-0.01	-0.01	-0.01	-0.00
PAKBXGSRMRCHCD_D	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
PAKBXGSRMRCHCD_X	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
PAKREER	174.70	180.17	184.98	189.29	193.21	196.87	200.38	202.81	206.66
PAKREER(-1)	168.41	174.70	180.17	184.98	189.29	193.21	196.87	200.38	202.81
PAKXMKT	13.19	13.51	13.84	14.18	14.52	14.88	15.24	15.61	15.99
PAKXMKT(-1)	12.87	13.19	13.51	13.84	14.18	14.52	14.88	15.24	15.61

Input base run:

	2024	2025	2026	2027	2028	2029	2030	2031	2032
PAKBXGSRMRCHCD(-1)	24,018.39	24,810.55	25,255.34	25,947.52	26,684.34	27,493.46	28,283.03	29,141.67	30,028.82
PAKBXGSRMRCHCD_A	-0.01	-0.01	-0.01	-0.01	-0.01	-0.01	-0.01	-0.01	-0.00
PAKBXGSRMRCHCD_D	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
PAKBXGSRMRCHCD_X	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
PAKREER	165.98	171.41	176.37	180.95	185.21	189.25	193.17	196.04	200.27
PAKREER(-1)	159.92	165.98	171.41	176.37	180.95	185.21	189.25	193.17	196.04
PAKXMKT	13.19	13.51	13.84	14.18	14.52	14.88	15.24	15.61	15.99
PAKXMKT(-1)	12.87	13.19	13.51	13.84	14.18	14.52	14.88	15.24	15.61

Difference for input variables

	2024	2025	2026	2027	2028	2029	2030	2031	2032
PAKBXGSRMRCHCD(-1)	76.10	77.38	77.10	75.73	73.65	71.13	68.33	65.41	62.41
PAKBXGSRMRCHCD_A	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
PAKBXGSRMRCHCD_D	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
PAKBXGSRMRCHCD_X	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
PAKREER	8.74	8.76	8.61	8.34	8.00	7.62	7.21	6.77	6.39
PAKREER(-1)	8.49	8.74	8.76	8.61	8.34	8.00	7.62	7.21	6.77
PAKXMKT	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
PAKXMKT(-1)	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

18.7 Adding, Modifying, or Deleting Equations

Sometime it can be useful to change a model. You might need to:

- Add new equations** to incorporate additional functionality or dynamics.
- Modify existing equations** to refine or adjust the model's behavior.
- Delete unnecessary equations** to simplify the model or remove obsolete components.

These actions enable you to tailor the model to better meet your specific requirements.

18.7.1 .eqdelete Delete equations

Parameter	Type	Description
deleteeq	TYPE, optional	Variables where equations are to be deleted. Defaults to None.
newname	string, optional	The name of the new model with deleted equations. Defaults to an empty string.

Returns:

- newmodel: TYPE - The new model with the deleted equations.
- newdf: TYPE - A dataframe with calculated add factors. Origin is the original model's lastdf.

18.7.2 .equpdate Update or add equations

Parameter	Type	Description
updateeq	TYPE	New equations separated by newline.
add_add_factor	bool, optional	Whether to add an add factor. Defaults to False.
calc_add	bool, optional	Whether to calculate add factors. Defaults to True.
do_preprocess	bool, optional	Whether to preprocess the equations. Defaults to True.
newname	string, optional	The name of the updated model. Defaults to an empty string.
silent	bool, optional	Whether to suppress print outputs. Defaults to True.

Returns:

- newmodel: TYPE - The updated model with new and deleted equations.
- newdf: TYPE - A dataframe with calculated add factors. Origin is the original model's lastdf.

```
dampable_variables = sorted(mpak.var_with_frmlname('DAMP'))
print(f'Number of dampable variables: {len(dampable_variables)}')
print(f'First 3 dampable variables : {dampable_variables[:3]}')
```

```
Number of dampable variables: 75
First 3 dampable variables : ['PAKBMFSTOTHRC', 'PAKBMFSTREMTCD', 'PAKBMGSRGNFSCD
˓→']
```

18.8 Manipulating DataFrames

Pandas dataframes can be extended with new functionality. When the `import modelclass` statement is executed ModelFlow will extend dataframes with two new capabilities.

- **.upd** which allows easy updating of variables (columns) in dataframes. The purpose is to facilitate relevant update operations.
- **.mfcalc** which enables transformations using lags and straight expressions.

18.8.1 .loc() method

18.8.2 .upd Updates a dataframe and returns a dataframe

Parameters:

Parameter	Type	Description	Default
basis	string	Lines with variable updates (see below for format).	
lprint	bool, optional	If True, each update is printed.	False
scale	float, optional	A multiplier used on all update input.	1.0
create	bool, optional	Creates variables if not in the dataframe.	True
keep_growth	bool, optional	Keep the growth rate after the update time frame.	False

A line in updates looks like this:

```
[<[[start] end]>] <var> <=|+|*|%|=growth|+growth|=diff> <value>... [--keep_growth_
→rate|--kg|--no_keep_growth_rate|--nkg]
```

The number of values should either be one - which then is applied to all years, or the number of years from start to end.

Time carries over from line to line.

The basisstring can contain several updates separated by new line.

Returns: A new dataframe containing the updated values of the input dataframe

18.8.3 .mfcalc Updates a dataframe and returns a dataframe

This implement the model class on top of pandas dataframes. Useful to do transformations with lags.

Parameter	Type	Description	De-fault
eq	str	Equations, one on each line. Can be started with <start end> to control calculation sample.	
start	optional	DESCRIPTION.	''
end	optional	DESCRIPTION.	''
showeq	bool, optional	If True, the equations will be printed.	False
**kwargs	optional	Here all solve options can be provided.	

Each equation f_i is specified as:

```
<left hand side> = <right hand side> $
```

Each formula ends with a \$.

The <left hand side> should not contain transformations. Lags or leads can not be specified at the left hand side of =.

Time t is implicit in the equations which means that a var at time t written as var , while var_{t-1} is written as $var(-1)$. ModelFlow is case-insensitive. Everything is eventually transformed into upper case.

The <right hand side> can contain variables, operators, functions

18.9 Performing Simulations

ModelFlow is a package to solve complex systems of equations. It supports a range of solution engines, each of which has different features.

18.9.1 <model instance> () method, solve the model

The most convenient way to solve a model is to call it by passing simulation parameters directly to the model instance¹.

Parameter	Type	Description
do_calc_add_fac	bool	Determines whether to calculate the add factors. Used to calculate add-factors to ensure the model exactly reproduces the submitted data set. Default is True.
reset_options	bool	If True, the previous options will be reset. Default is False.
solver	str	Specifies the solver to be used. Default is chosen based on the model's properties. (Typically sim (Gauss-seidel) for WB models, or newton_stacked for forward-looking models.)
silent	bool	If True, the solver runs silently without printing outputs to the console. Default is True.
keep	str	If provided, keeps the solutions. Behavior depends on the keep_variables option.
keep_variables	str, list of str	Specifies which variables to keep if the keep option is provided. Default is '*' - to keep all variables.
*args	various	Variable length argument list. Usual the DataFrame and start and end year
**kwargs	various	Arbitrary keyword arguments. These are provided to the actual solver

Returns

- outdf: pandas.DataFrame - The DataFrame containing the results of the model run.

example

```
# assume the model object has already been loaded and solved mpak
oilshockdf = mpak.basedf.upd('<2025 2027> WLDFCRUDE_PETRO + 25')
#Simulate the model
ExogOilSimul = mpak(oilshockdf, 2020, 2040, keep='$25 increase in oil prices 2025-27')
```

- **First line:** creates a new DataFrame as a copy of the basedf DataFrame but with the crude oil price increased by 25 between 2025 and 2027 inclusive.
- **Second line:** Performs the simulation. Submits the altered DataFrame oilshockdf to the model and solves the mode from 2020 to 2024, assigning the results DataFrame ExogOilSimul and to the keep dictionary with the text identifier '\$25 increase in oil prices 2025-27'

¹ For those familiar with programming in Python, technically this is implemented as the model class method `__call__`.

<model instance> () Persistent arguments

Named arguments to <model instance> () are persistent. So if a solver is set in a call, the same solver is used in the next call. The same goes if silent is set or alfa. And so on.

The stored arguments are stored in .oldkwargs. All arguments can be restored to default by the function: <model instance>(,,,reset_options=True)

.oldkwargs List persistent simulation arguments

```
mpak.oldkwargs
```

```
{'alfa': 0.7, 'keep': '$25 increase in oil prices 2025-27'}
```

.model_solver The solver used.

The property .model_solver contains the solver function used at the last call <model instance>(). To retrieve the name of the solver use:

```
mpak.model_solver.name
```

```
'sim'
```

This should be a solver function from the list below.

18.10 Explicit simulation methods

ModelFlow has a number of different solvers which can be used for different types of models and in different circumstances.

<model instance> () will try to find the most appropriate solver for the model at hand. However the user can select the solver explicit by <model instance>(,,solver='<solver>')

Solver Name	Method and use case
sim	Gauss-Seidel method for models with contemporaneous feedback
newton	Newton-Raphson method, alternative for models with contemporaneous feedback
newton_stacked	Newton-Raphson method for models with forward-looking models
xgenr	will calculate the result for models without contemporaneous feedback
res	each equation is calculated on its own. Used to check the residuals.

Instead of using the <model instance> () method discussed above each of these solvers can be called directly.

```
<model instance>.solver()
```

18.10.1 .sim method of simulation

This alternative method forces use of the Gauss-Seidel technique and is functionally equivalent to the modelobjectname method discussed above with the option `solver='sim'` set.

Parameter	Type	Description
databank	dataframe	Input DataFrame containing the data to be used in the simulation.
start	TYPE	The start of the simulation period. Defaults to “”.
end	TYPE	The end of the simulation period. Defaults to “”.
silent	bool	If set to False, displays simulation logs. Defaults to True.
samedata	bool	If True, indicates the input data has the same structure as the last simulation. Defaults to False.
alfa	float	The damping factor applied during simulation. Defaults to 1.0.
stats	bool	If True, displays statistics after the simulation is complete. Defaults to False.
first_test	int	The iteration number to start testing for convergence. Defaults to 5.
max_iterations	int	The maximum number of iterations allowed for the simulation. Defaults to 200.
conv	str	Specifies the variables to test for convergence. Defaults to “*”.
absconv	float	Sets the absolute convergence criterion level. Defaults to 0.01.
relconv	float	Sets the relative convergence criterion level. Defaults to DEFAULT_relconv.
transpile_reset	bool	If True, ignores the previously transpiled model. Defaults to False.
dumpvar	str	Specifies the variables for which to dump the iterations. Defaults to “*”.
init	bool	If True, takes the previous period’s value as the starting value. Defaults to False.
ldumpvar	bool	If True, dumps the iterations. Defaults to False.
dumpwith	int	Not described in the original docstring. Defaults to 15.
dumpdecimal	int	Not described in the original docstring. Defaults to 5.
chunk	int	Specifies the chunk size of the transpiled model. Defaults to 30.
ljit	bool	If True, enables Just-In-Time compilation. Defaults to False.
stringjit	bool	If True, uses Just-In-Time compilation on a string, not a file. Defaults to False.
timeon	bool	If True, times the elements of the simulation. Defaults to False.
fairopt	TYPE	Options for Fair Taylor approximation. Defaults to {‘fair_max_iterations’: 1}.
progressbar	TYPE	If True, shows a progress bar during the simulation. Defaults to False.
**kwargs	various	Additional keyword arguments for more customized control over the simulation.

Returns

- A DataFrame with the simulation results.

example

Parameters

```
results = mpak.sim(baseline, 2023, 2050, alfa=0.5, max_iterations=100)
```

Submits the baseline DataFrame to the model object and solves the model using the Gauss-Seidel solver from 2023 to 2050, assigning the results to the DataFrame `results`. `alfa=0.5` determines the dampening when solving the model, while `max_iterations` dictates the maximum number of solves to run for any given shock.

This solver is appropriate for non-forward looking World Bank models.

18.10.2 ### .newton() and .stacked_newton() simulation methods

Use the Newton-Raphson method for solving models. The stacked version solves the model for all years simultaneously. This is a computationally more challenging approach and is mainly used when solving models with forward-looking variables, which precludes solving the model sequentially year by year.

18.10.3 .newton and stacked_newton

Parameters

Parameter	Type	Description
self	object	The instance of the class where this method is defined.
databank	DataFrame	The databank on which the model will be evaluated.
start	string	The start period for the evaluation. Default is an empty string.
end	string	The end period for the evaluation. Default is an empty string.
silent	int	Determines whether to print progress messages. Default is 1 (silent).
samedata	int	Not used in the provided code. Default is 0.
alfa	float	A parameter used in the evaluation. Default is 1.0.
stats	bool	If True, prints statistics after the evaluation. Default is False.
first_test	int	The iteration at which to start testing for convergence. Default is 1.
newton_absconv	float	The absolute convergence criterion. Default is 0.001.
max_iterations	int	The maximum number of iterations for the Newton method. Default is 20.
conv	string	Specifies the variables to check for convergence. Default is '*'.
absconv	float	The absolute convergence level. Default is 1.0.
relconv	float	The relative convergence level. Default is DEFAULT_relconv.
nonlin	bool	If True, updates the solver in nonlinear iterations. Default is False.
timeit	bool	If True, times the execution of the method. Default is False.
newton_reset	int	Determines whether to reset the Newton method. Default is 1.
dumpvar	string	Specifies the variables to dump. Default is '*'.
ldumpvar	bool	If True, enables the dumping of variables. Default is False.
dumpwith	int	Not used in the provided code. Default is 15.
dumpdecimal	int	Not used in the provided code. Default is 5.
chunk	int	Determines the chunk size for JIT compilation. Default is 30.
ljit	bool	If True, enables JIT compilation. Default is False.
stringjit	bool	If True, enables JIT compilation for strings. Default is False.
transpile_reset	bool	If True, resets the transpiler. Default is False.
init	bool	If True, initializes the method. Default is False.
newtonalpha	float	A parameter for the Newton method. Default is 1.0.
newtonnodamp	int	The iteration at which to stop damping in the Newton method. Default is 0.

returns a dataframe with the solution

examples

```
results = mpak.newton(baseline, 2023, 2050, alfa=0.5, max_iterations=100)
or
results = mpak.stacked_newton(baseline, 2023, 2050, alfa=0.5, max_iterations=100)
```

Submits the baseline DataFrame to the model object and solves the model using the Newton or stacked_Newton solvers from 2023 to 2050, assigning the results to the DataFrame results. alfa=0.5 determines the step size when solving the model, while max_iterations dictates the maximum number of solves to run for any given shock.

This solver is used by most world bank models.

18.10.4 Troubleshooting simulations

Not every model will solve with the default parameters. However, there are ways to increase the probability of a solution being found if an initial solution fails.

The Gauss-Seidel algorithm is quite straight forward. It basically iterates over the formulas, until convergence.

Let:

- z be all predetermined values: all exogenous variables and lagged endogenous variables.
- n be the number of endogenous variables.
- α dampening factor which can be applied to selected equations.
- i endogenous variables
- k iterations

For each time period we can find a solution by doing Gauss-Seidel iterations:

for $k = 1$ to convergence

 for $i = 1$ to n

$$y_i^k = (1 - \alpha) * y_i^{k-1} + \alpha f_i(y_1^k, \dots, y_{i-1}^k, y_{i+1}^{k-1}, \dots, y_n^{k-1}, z)$$

Convergence: relconv, absconv, conv

After each iteration the convergence is achieved if the relative change from the iteration before is below relconv for selected endogenous variables. As the relative change for small values can be very large only variables with values above absconvare checked. As default all endogenous variable are checked, however the user can select a subset of variables to check by setting the convto the names for which to check.

silent

Set silent=False, in order to learn more about the state of the model following each iterations

max_iterations

Can be increased

alfa

This parameter determines the dampening of dampable equations. The value should be between 1 and 0.1.

18.10.5 Persistent simulation options: the `.oldkargs` property

The `oldkargs` property of the model object is automatically set with each simulation, storing the options that were active when it was executed. Unless options are changed they are persistent between calls to a given model object. The `.oldkargs` property of a model object stores the persistent parameters and can be interrogated.

The current options of the mpak model object are:

```
mpak.oldkargs
```

```
{'alfa': 0.7, 'keep': '$25 increase in oil prices 2025-27'}
```

Persistent options may be reset (set to their defaults) by the user by setting the dictionary to the empty set.

```
mpak.oldkargs = {}
```

18.11 Modifying models

A new model instance with deleted, modified or added equations can be produced.

18.12 Saving results for comparison

When comparing results the user can take the result dataframe from two (or more) simulation and use python do the necessary calculations and visualizations. However Modelflow provides some properties and methods which facilitates comparing more “out of the box”.

A model instance (in this case mpak two “systems” to do this. The

1. `.basedf` and `.lastdf` dataframes. Which contains the first and the last solution of the model
2. `.keep_solution` dictionary of dataframes. This is typical used when comparing several scenarios.

18.12.1 `.basedf` and `.lastdf`

In the example above there two dataframes with results `baseline` and `scenario`. 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

This means that .basedf and .lastdf will contain the same result after the first simulation. If new scenarios are simulated the data in .lastdf will then be replaced with the latest results.

These dataframes are used by a number of model instance methods as you will see later.

The user can assign dataframes to both .basedf and .lastdf. This is useful for comparing simulations which are not the first and last.

18.12.2 keep=<Description>, create a dictionary of dataframes from scenarios

Sometimes we want to be able to compare more than two scenarios. Using `keep='some description'` the dataframe with results can be saved into a dictionary with the description as key and the dataframe as value.

```
mpak(,keep=<description>,)
```

The name of the dictionary will be mpak.keep_solutions

keep_variables= <selection string of variables>, Select variables to keep

A modelinstance with many variables and/or many scenarios to keep can become very large. Therefor it can be useful not to keep all variables but to select a the variables which are needed.

.keep_solutions = {}, resetting the .keep_solution

Sometime it can be useful to reset the `.keep_solutions`, so that a new set of solutions can be inspected. This is done by replacing it with an empty dictionary.

18.12.3 Example

```
#Drops any previously kept solutions
mpak.keep_description = {}

# Solve the model using the baseline dataframe as a starting point and keep the
# result as `Baseline`
_ = mpak(baseline,2020,2100,keep='Baseline')
# Create a new dataframe called scenario_20 taht is a copy of baseline, but with the
# values of three variables
# set to 20 between 2020 and 2100
scenario_20 = baseline.upd("<2020 2100> PAKGGREVCO2CER PAKGGREVCO2GER
                           ↪PAKGGREVCO2OER = 20")
# = mpak(scenario_20,2020,2100,keep='Coal, Oil and Gastax : 20') # runs the simulation
# Create a new dataframe called scenario_20 taht is a copy of baseline, but with the
# values of three variables
# set to 20 between 2020 and 2100
scenario_40 = baseline.upd("<2020 2100> PAKGGREVCO2CER PAKGGREVCO2GER
                           ↪PAKGGREVCO2OER = 40")
#Run a simulationusing the revised dataframe using the revised to 40 dataset as input
_ = mpak(scenario_40,2020,2100,keep='Coal, Oil and Gas Carbon tax = 40 USD') # runs
# the simulation
```

.keepswitch(), select scenarios for plotting

When creating many scenarios with the `keep` keyword it can sometime be useful to:

- Plot selected scenarios and to change the sequence of the scenarios.
- To use the `.basedf` and `.lastdf` as scenarios.

To facilitate this the `.keepswitch` function has been created.

Parameter	Explanation
<code>base_last = True False(default)</code>	If True use the <code>.basedf</code> and <code>.lastdf</code>
<code>scenario='<string>'</code>	Listing of scenarios separated by if no wildcards else space

`.keepswitch` is a `context manager` (<https://www.pythontutorial.net/advanced-python/python-context-managers>) like `.set_smpl`. So it works in the scope of a `with`. After the `with` statement the kept solutions are restored.

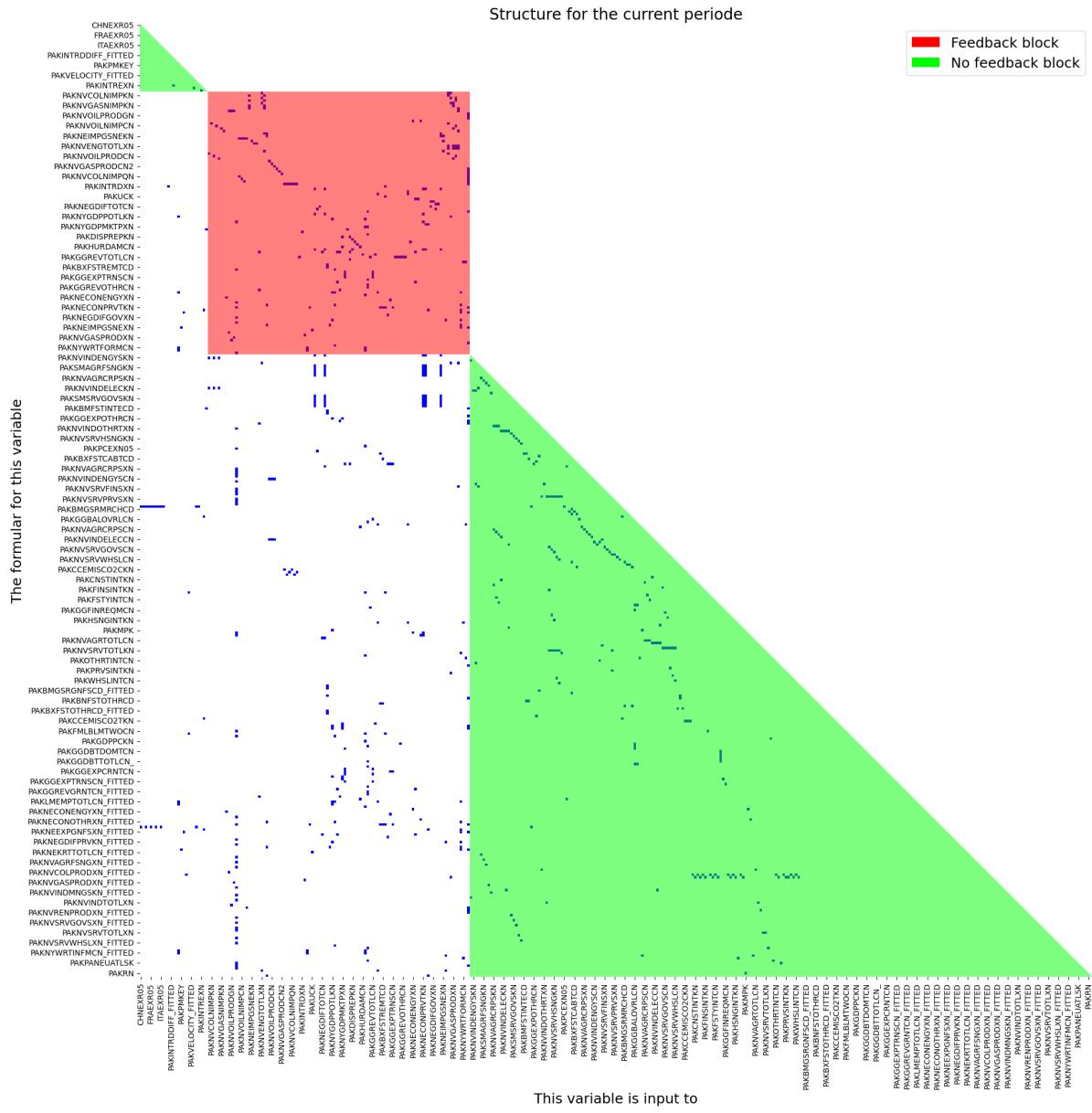
```
with mpak.keepswitch(scenarios = '\<selection>'):
    mpak.keep_plot('\<variable selection>',,)
```

18.13 Model Analytics

When ModelFlow process a model it creates a graph (a network) of the interdependencies between the different variables. The graph is used to find the causal ordering of the model.

A model in MFMod typical contains at least one simultaneous block and a pre and post block.

```
mpak.plotadjacency(size=(20,20),title='Structure for the current periode');
```



18.14 Setting time frame

The property `.current_per` is the master time index for a model instance. It determinate the time frame for simulations and output.

Sometime it can be useful to change the `.current_per` the property can be set with:

Function	Description
<code>.smpl</code>	Sets the sample period for analysis and reporting.
<code>with .set_smpl:</code>	Sets the sample period for analysis and reporting in a local scope .
<code>with .set_smpl_relative:</code>	Sets the sample period for analysis and reporting relative to the current period or scenario in a local scope.

```
mpak.current_per
```

```
Index([2020, 2021, 2022, 2023, 2024, 2025, 2026, 2027, 2028, 2029, 2030, 2031,
       2032, 2033, 2034, 2035, 2036, 2037, 2038, 2039, 2040, 2041, 2042, 2043,
       2044, 2045, 2046, 2047, 2048, 2049, 2050, 2051, 2052, 2053, 2054, 2055,
       2056, 2057, 2058, 2059, 2060, 2061, 2062, 2063, 2064, 2065, 2066, 2067,
       2068, 2069, 2070, 2071, 2072, 2073, 2074, 2075, 2076, 2077, 2078, 2079,
       2080, 2081, 2082, 2083, 2084, 2085, 2086, 2087, 2088, 2089, 2090, 2091,
       2092, 2093, 2094, 2095, 2096, 2097, 2098, 2099, 2100],
      dtype='int64')
```

The possible times in the dataframe is contained in the `<dataframe>.index` property.

```
mpak.lastdf.index # the index of the dataframe
```

```
Index([1980, 1981, 1982, 1983, 1984, 1985, 1986, 1987, 1988, 1989,
       ...
       2091, 2092, 2093, 2094, 2095, 2096, 2097, 2098, 2099, 2100],
      dtype='int64', length=121)
```

18.14.1 .smpl, Set time frame

The time frame can be set like this:

```
mpak.smpl(2020,2025)
mpak.current_per
```

```
Index([2020, 2021, 2022, 2023, 2024, 2025], dtype='int64')
```

18.14.2 .set_smpl, Set timeframe for a local scope

For many operations it can be useful to apply the operations for a shorter time frame, but retain the global time frame after the operation. This can be done with a `with` statement like this.

```
print(f'Global time before {mpak.current_per}')
with mpak.set_smpl(2022,2023):
    print(f'Local time frame {mpak.current_per}')
print(f'Unchanged global time {mpak.current_per}')

Global time before Index([2020, 2021, 2022, 2023, 2024, 2025], dtype='int64')
Local time frame Index([2022, 2023], dtype='int64')
Unchanged global time Index([2020, 2021, 2022, 2023, 2024, 2025], dtype='int64')
```

18.14.3 .set_smpl_relative Set relative timeframe for a local scope

When creating a script it can be useful to set the time frame relative to the current time.

Like this:

```
print(f'Global time before {mpak.current_per}')
with mpak.set_smpl_relative (-1,0):
    print(f'Local time frame {mpak.current_per}')
print(f'Unchanged global time {mpak.current_per}')

Global time before Index([2020, 2021, 2022, 2023, 2024, 2025], dtype='int64')
Local time frame Index([2019, 2020, 2021, 2022, 2023, 2024, 2025], dtype=
˓→'int64')
Unchanged global time Index([2020, 2021, 2022, 2023, 2024, 2025], dtype='int64')
```

18.15 Reports

ModelFlow offers a range of features to present and visualize model results. Below is a list:

- **.table():**

This function creates a table from data in the .lastdf and .basedf dataframes. It can be customized with various arguments to adjust the data type (e.g., growth rate or level), the number of decimals, and table titles.

- **.plot():**

This function generates charts based on data from .lastdf, .basedf, or saved scenarios in from .keep_solution. It can be customized with various arguments to choose the chart type, data transformation (e.g., growth rate or level), scenarios to display, and chart titles.

- **.text():**

This function creates a text object that can contain plain text, LaTeX code, or HTML code. It is primarily used to add text to reports.

- **.report():**

This function combines tables, plots, text, and other reports into a consolidated report. Reports can be saved and reused, making it easy to generate reports for various scenarios.

18.15.1 The .table function

Line Options

These options are set on table creation.

Argument	Default Value	Description
pat	'#Headline' (defined in the ModelFlow class)	Variable names to be displayed, may include wildcard specifications.
datatype	'growth'	Defines the data transformation displays (cf. next table).
mul	1.0	Multiplier of values before display.
dec	2	Set the decimal places to display.
col_desc	Depend on datatype	A centered description of columns (non-transposed table).
heading	"	A centered text above columns (non-transposed table).
rename	True	If True, use descriptions; else, use variable names.

Table Options

These options can be revised after a table has been generated.

Argument	Default Value	Description
name	“”	Name for this display.
custom_description	{}	Custom description, overrides default descriptions.
title	“”	Title text for the table.
foot	“”	Footer text for the table.
transpose	False	If True, transposes the table.
chunk_size	0	Specifies the number of columns per chunk in tables.
timeslice	[]	Specifies the time slice for data display.
max_cols	6	Maximum columns when displayed as string.
last_cols	1	In LaTeX, the number of last columns in a display slice.
smp1	(“, “)	Set or reset the sample (smp1) for the table.

Table: Data Types and Their Meanings

datatype	Meaning
growth	This is the default setting. Growth rate in percent in the most recent dataset (.lastdf).
level	Values in .lastdf.
gdppct	Percentage of GDP in .lastdf.
qoq_ar	Quarterly growth annualized. - Quarterly models only.
Difference views	“”
difgrowth	Change (Δ) in growth rates (.lastdf less basedf).
diflevel	Change (Δ) in values (.lastdf less basedf).
diffdppct	Change (Δ) in the percentage of GDP from (.lastdf less basedf).
diffqoq_ar	change in the annualized quarterly growth (.lastdf less basedf)
diffpctlevel	Percentage change (Δ) in values (.lastdf less basedf).
Values from .basedf	“”
basegrowth	Growth rate in percent in .basedf.
baselevel	Level of the data in .basedf
basegdppct	Percentage of GDP in .basedf.
baseqoq_ar	Quarterly growth in .basedf

18.15.2 The `.plot` function

Line options are set when the plot is created with the `.plot()` call and can only be set when the object is instantiated.

Argument	Default Value	Description
pat	'#Headline'	Pattern with wildcard for variable names
datatype	'growth'	Defines the datatransformation displayes (cf. next table)
mul	1.0	Multiplier of values before display
ax_title_template	“”	Template for each chart title
by_var	True	Show results by variable (False) or scenario (True)

Plot options can be revised after the plot has been created.

Argument	Default Value	Description
rename	True	If True use descriptions else variable names
scenarios	"	Show for .basedf/.lastdf or selected scenarios
name	"	Name for this display.
custom_description	{ }	Custom description, override default descriptions.
title	"	Title (used when samefig=True).
samefig	False	If True, displays all chart in the same plot area.
ncol	2	Number of columns when samefig=True.
size	(10, 6)	Specifies the size of each chart
legend	True	If True, includes a legend in the display.
smpl	(",")	set smpl for this plot

The table below summarizes how the combination of `datatype` and `scenarios` determines the data displayed in the chart.

Table: Datatype and Scenario Settings

Datatype	<code>scenarios="</code> or omitted	<code>scenarios='<list of scenarios>'</code> (Multiple Scenarios)
Value Views		
growth (default)	Growth rate (%) in the most recent dataset (.lastdf).	Growth rates for all scenarios.
level	Values from .lastdf.	Values for all scenarios.
gdppct	Percentage of GDP in .lastdf.	Percentage of GDP for all scenarios.
qoq_ar	Quarterly growth (annualized) for quarterly models in .lastdf.	Quarterly growth (annualized) for all scenarios.
Difference Views		
difgrowth	Change (Δ) in growth rates between .lastdf and .basedf.	Change (Δ) in growth rates between the first and other scenarios.
diflevel	Change (Δ) in values between .basedf and .lastdf.	Change (Δ) in values between the first and other scenarios.
difgdppct	Change (Δ) in the percentage of GDP between .basedf and .lastdf.	Change (Δ) in the percentage of GDP between the first and other scenarios.
difqoq_ar	Change in quarterly growth (annualized) between .basedf and .lastdf.	Change (Δ) in quarterly growth (annualized) between the first and other scenarios.
difpctlevel	Percentage change (Δ) in values between .basedf and .lastdf.	Percentage change (Δ) in values between the first and other scenarios.
.basedf Views		
basegrowth	Growth rate (%) in .basedf.	Growth rates for the first scenario.
baselevel	Values from .basedf.	Values for the first scenario.
basegdppct	Percentage of GDP in .basedf.	Percentage of GDP for the first scenario.
baseqoq_ar	Quarterly growth (annualized) in .basedf.	Quarterly growth (annualized) for the first scenario.

.savefigs () Saving plots in other formats.

Plots are created using the `matplotlib` library. If charts are going to be used further downstream they can be saved to file using a wide range of formats.

.savefigs () options

Parameter	Type	Description	Default
location	str	The folder in which to save the charts.	'./graph'
experimentname	str	A subfolder under location where charts are saved.	'experiment1'
addname	str	An additional name added to each figure filename.	'' (empty string)
extensions	list	A list of file extensions for saving the figures.	['svg']
xopen	bool	If True, open the saved figure locations in a web browser.	

Charts can be saved using the following formats:

Extension	Description
.png	Portable Network Graphics
.jpg or .jpeg	JPEG Photographic Experts Group
.tif or .tiff	Tagged Image File Format
.bmp	Bitmap
.gif	Graphics Interchange Format
.svg	Scalable Vector Graphics
.pdf	Portable Document Format
.eps	Encapsulated PostScript
.ps	PostScript
.raw	Raw image data
.rgba	Raw RGBA bitmap
.pgf	Portable Graphics Format

18.15.3 The .text () class

The text class allows the user to define three different types of text: `.text_text` `.html_text` and `.latex_text`. These properties are optional and can be declared explicitly or implicitly.

object	delimited by	contains
.text_text	nothing	Plain text
.html_text	<html> </html>	html text
.latex_text	<latex> </latex>	latex text

18.15.4 Joing plots or tables with |

Tables, Plots can be concatenated together using the `|` operator. This allows plots or tables with different datatypes to be displayed together.

Plots can only be joined with plots, and tables can only be joined by tables.

18.15.5 Creating reports using + between plots, tables and text

Reports in ModelFlow are containers that can contains figures, plots, tables of different dimension and text. Moreover, Report objects can be saved in a model object and regenerated on new data without requiring any additional effort by the user.

The `+` operator creates a new report container specification.

Reports can be placed in the dictionary: `model instance.reports`. Reports can then be reused later.

Functions	task
<code>model instance.add_report(reports,,,)</code>	Add report to <code>model instance.reports</code> , key = report name
<code>model instance.get_report(key)</code>	Returns a report from <code>model instance.reports</code>

A report can be modified with these functions.

Functions	task
<code>.set_scenarios</code>	Set scenarios in report
<code>.set_name</code>	Sets name of report

18.15.6 .set_option() Modifying reports, tables or plots options

With this function options (except line options) can be modified after the element has been created.

18.15.7 Rendering reports, tables, plots and text objects

These objects can be rendered as html, text and pdf objects. The below table indicates the methods associated with each.

Display options for report type objects

command	output
None (just the object name)	In Jupyter Notebook displays the table in its html format.
<code>display(tab)</code>	Displays the table in html format.
<code>.show</code>	Returns a simple text version of the table.
<code>.pdf()</code>	Returns a nicely formatted table in pdf format.

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

The World Bank's MFMod Framework in Python with Modelflow

```
dict_keys(['Baseline', '$25 increase in oil prices 2025-27', 'Coal, Oil and Gastax ↴: 20', 'Coal, Oil and Gas Carbon tax = 40 USD'])
```

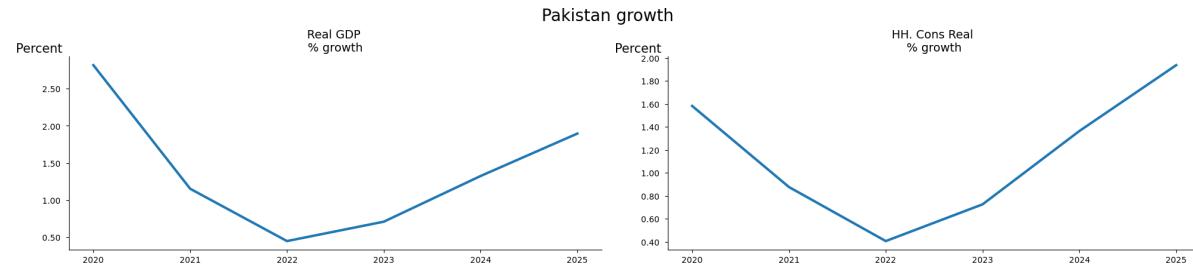
```
pat= '*NYGDPMKTPKN *NECONPRVTKN '
tab = mpak.table(pat=pat, name='My_first_table')
```

```
pat_gov = '*GGBALOVRLCN *GGDBTTOTLCN *BNCAFUNDCD'
tab_gov = mpak.table(pat_gov, datatype='gdpct', dec=1)
(tab|tab_gov).set_options(smpl=(2022, 2026),
    title='{cty_name} GDP components').show
```

Pakistan GDP components					
	2022	2023	2024	2025	2026
--- Percent growth ---					
Real GDP	0.45	0.71	1.32	1.90	2.32
HH. Cons Real	0.41	0.73	1.36	1.94	2.34
--- Percent of GDP ---					
General Government Revenue, Deficit, LCU mn	-3.5	-3.5	-3.4	-3.3	-3.2
General government gross debt millions lcu	54.9	55.0	55.4	56.0	56.7
Current Account Balance, US\$ mn	-4.4	-3.8	-3.5	-3.3	-3.2

```
fig = mpak.plot(pat,samefig=2,title='{cty_name} growth')
((tab|tab_gov).set_options(smpl=(2022, 2026),
    title='{cty_name} GDP components')+fig).show
```

Pakistan GDP components					
	2022	2023	2024	2025	2026
--- Percent growth ---					
Real GDP	0.45	0.71	1.32	1.90	2.32
HH. Cons Real	0.41	0.73	1.36	1.94	2.34
--- Percent of GDP ---					
General Government Revenue, Deficit, LCU mn	-3.5	-3.5	-3.4	-3.3	-3.2
General government gross debt millions lcu	54.9	55.0	55.4	56.0	56.7
Current Account Balance, US\$ mn	-4.4	-3.8	-3.5	-3.3	-3.2



18.16 Using the index operator .[] to select and visualize variables.

Using the index operator on the model instance like this:

```
mpak[<variable selection>]
```

will return a special class (called `.vis`). It implements a number of methods and properties which comes in handy for quick variable:

- visualization
- analysis
- information

First the user has to select the relevant variables then several properties and methods can be chained

18.16.1 Variable selection

In several contexts it is possible to select a number of variables. Variables can be selected on different basis:

1. Variable name with wildcards
2. Variable descriptions with wildcards
3. Variable groups
4. all endogenous variables

'<variable name with wildcards>'... , select matching variables

To select variables the method accept patterns which defines variable names. Wildcards:

- * matches everything
- ? matches any single character
- \ [seq] matches any character in seq
- \ [!seq] matches any character not in seq

For more how wildcards can be used, the specification can be found [here](https://docs.python.org/3/library/fnmatch.html) (<https://docs.python.org/3/library/fnmatch.html>)

Use of {cty} and deferred substitution in variable selection

Using * or ? to select variables will include all countries, even if some may not be relevant in a particular selection. In order to reuse the same variable selection across different countries deffered substitution can be used.

In this approach, {cty} placeholders can be used. {cty} will be replaced with the ISO 3 letter code for the country in question. And this is done at the execution time - explaining the “deferred” term.

For this to work, a dictionary .substitution must be defined. This dictionary is typically set in the onboarding script but can be set at any point as needed.

Here ???exr05 will select EXR05 for all countries

```
mpak[ '???exr05' ].names
```

```
[ 'CHNEXR05',
  'DEUEXR05',
  'FRAEXR05',
  'GBREXR05',
  'ITAEXR05',
  'PAKEXR05',
  'TUREXR05',
  'USAEXR05']
```

Here ?only the pakistan value is selected.

If the model is used for another country the cty can be changed, without changing the specification

```
mpak['{cty}exr05'].names
```

```
[ 'PAKEXR05']
```

.substitution Defining deferred substitution

To enable reuse of variable selection and report specification this feature enable substitution of {key} in a string with <model instance>.substitution['key'] where .substitution is a dictionary.

When onboarding a World Bank model the <model instance>.substitution looks like this:

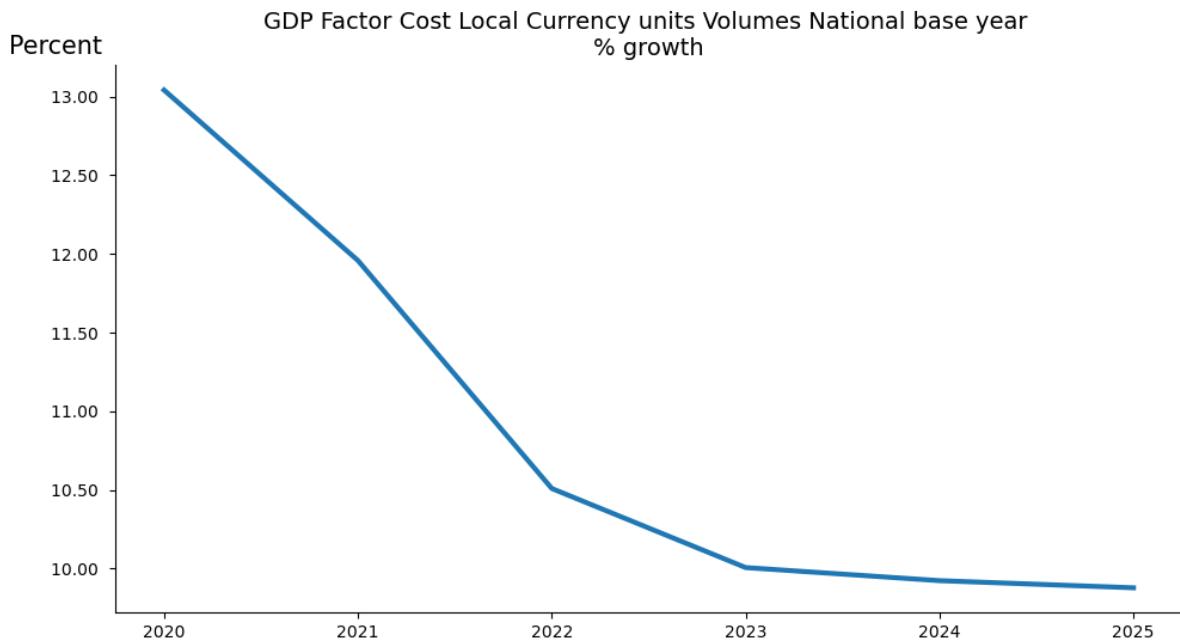
```
mpak.substitution
```

```
{'cty': 'PAK', 'cty_name': 'Pakistan'}
```

Deferred substitution works with variable selection, report titles and report text.

```
mpak['{cty}NYgdpFCSTCN'].endo_nofit.rplot(datatype='growth',
                                             title='{cty_name} charts', rename=False).show
```

Pakistan charts



'<#Variable group>' , select variables matching in variable group

.var_groups, a dictionary of variable groups

The property .var_groups can contain a dictionary of variables defined by variable names with wildcards.

```
mpak.var_groups
```

```
{
  'Headline': '{cty}NYGDPMKTPKN {cty}NYGDPMKTPXN {cty}NRTOTLCN {cty}LMUNRTOTLCN
  ↵{cty}BFFINCABCD  {cty}BFBOPTOTLCD {cty}GGBALEXGRCN {cty}GGDBTTOTLCN_ {cty}
  ↵GGDBTTOTLCN {cty}BNCABLOCLCD_ {cty}FPCPITOTLXN {cty}CCEMISCO2TKN',
  'National income accounts': '{cty}NY*',
  'National expenditure accounts': '{cty}NE*',
  'Value added accounts': '{cty}NV*',
  'Balance of payments exports': '{cty}BX*',
  'Balance of payments exports and value added ': '{cty}BX* {cty}NV*',
  'Balance of Payments Financial Account': '{cty}BF*',
  'General government fiscal accounts': '{cty}GG*',
  'World all': 'WLD*',
  'All variables': '**'
}
```

Example

```
mpak['#Balance of payments exports'].des
```

```

PAKBFXFSTCABTCD      : Exp., Factor Services and Transfers (BOP), US$ mn
PAKBFXFSTOTHRCDD     : Exp., Other Factor Services and Transfers (BOP), US$ mn
PAKBFXFSTOTHRCDD_A   : Add factor:Exp., Other Factor Services and Transfers (BOP),
    ↳ US$ mn
PAKBFXFSTOTHRCDD_D   : Fix dummy:Exp., Other Factor Services and Transfers (BOP), ↳
    ↳ US$ mn
PAKBFXFSTOTHRCDD_FITTED : Fitted value:Exp., Other Factor Services and Transfers ↳
    ↳ (BOP), US$ mn
PAKBFXFSTOTHRCDD_X   : Fix value:Exp., Other Factor Services and Transfers ↳
    ↳ US$ mn
PAKBFXFSTREMTCD      : Exp., Remittances (BOP), US$ mn
PAKBFXFSTREMTCD_A    : Add factor:Exp., Remittances (BOP), US$ mn
PAKBFXFSTREMTCD_D    : Fix dummy:Exp., Remittances (BOP), US$ mn
PAKBFXFSTREMTCD_FITTED : Fitted value:Exp., Remittances (BOP), US$ mn
PAKBFXFSTREMTCD_X    : Fix value:Exp., Remittances (BOP), US$ mn
PAKBXGSRGNFSCD       : Exp., GNFS (BOP), US$ mn
PAKBXGSRGNFSCD_A     : Add factor:Exp., GNFS (BOP), US$ mn
PAKBXGSRGNFSCD_D     : Fix dummy:Exp., GNFS (BOP), US$ mn
PAKBXGSRGNFSCD_FITTED : Fitted value:Exp., GNFS (BOP), US$ mn
PAKBXGSRGNFSCD_X     : Fix value:Exp., GNFS (BOP), US$ mn
PAKBXGSRMRCHCD       : Exp., MRCH (BOP), US$ mn
PAKBXGSRMRCHCD_A     : Add factor:Exp., MRCH (BOP), US$ mn
PAKBXGSRMRCHCD_D     : Fix dummy:Exp., MRCH (BOP), US$ mn
PAKBXGSRMRCHCD_FITTED : Fitted value:Exp., MRCH (BOP), US$ mn
PAKBXGSRMRCHCD_X     : Fix value:Exp., MRCH (BOP), US$ mn
PAKBXGSRNFSVCD       : Exp., NF SERV (BOP), US$ mn

```

'<!search pattern>', select variables where search pattern with wildcards is matching description

.var_descriptions, a dictionary of variable descriptions

The property .var_description contains a dictionary with variable names as key and a description as value.

Example

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

```

PAKBMFSTINTECD      : Imports, External Debt Interest Payments
PAKNEIMPGNFSKN       : Imports real
PAKNEIMPGNFSKN_A     : Add factor:Imports real
PAKNEIMPGNFSKN_D     : Fix dummy:Imports real
PAKNEIMPGNFSKN_FITTED : Fitted value:Imports real
PAKNETMPGNFSKN_X     : Fix value:Imports real
PAKNVCOLNIMPQN       : Coal, net import (ktoe)
PAKNVGASNIMPQN       : Gas, net import (ktoe)
PAKNVOILNIMPQN       : Oil, net import (ktoe)
PAKNVRENNIMPQN       : "Renewables", net import (ktoe)
PAKPMKEY              : Keyfitz Price Imports

```

'#ENDO', select all edogenous variables

```
# Show the first 4 names of endogenous variables
mpak ['#endo'].names[:4]
```

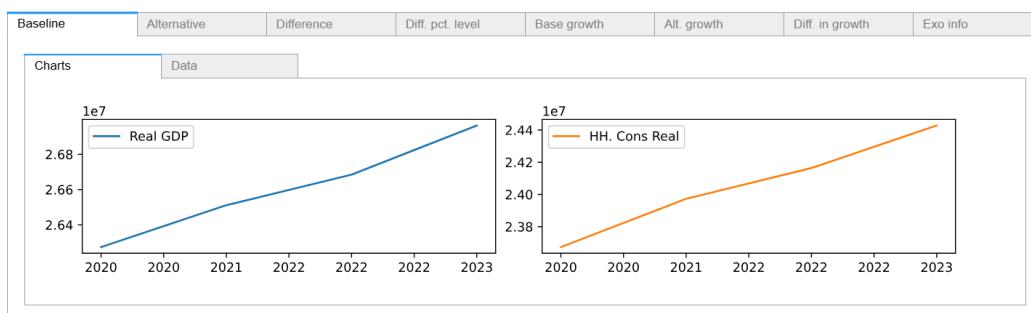
```
['CHNEXR05', 'CHNPCEXN05', 'DEUEXR05', 'DEUPCEXN05']
```

18.16.2 Access values in .lastdf and .basedf

To limit the output printed, we set the time frame to 2020 to 2023.

```
mpak.smpl(2020, 2023);
```

To access the values of 'PAKNYGDPMKTPKN' and 'PAKNECONPRVTKN' from the latest simulation a small widget



is displayed.

To access the values of 'PAKNYGDPMKTPKN' and 'PAKNECONPRVTKN' from the base dataframe, specify .base

```
mpak['PAKNYGDPMKTPKN PAKNECONPRVTKN'].base.df
```

	PAKNYGDPMKTPKN	PAKNECONPRVTKN
2020	2.627394e+07	2.367289e+07
2021	2.651137e+07	2.397282e+07
2022	2.668514e+07	2.416413e+07
2023	2.696308e+07	2.442786e+07

18.16.3 .df Pandas dataframe

Sometime you need to perform additional operations on the values. Therefor the .df will return a dataframe with the selected variables.

```
mpak['PAKNYGDPMKTPKN PAKNECONPRVTKN'].df
```

	PAKNYGDPMKTPKN	PAKNECONPRVTKN
2020	2.648605e+07	2.338267e+07
2021	2.679140e+07	2.358771e+07
2022	2.691168e+07	2.368379e+07
2023	2.710269e+07	2.385588e+07

18.16.4 .names Variable names

If you select variables using wildcards, then you can access the names that correspond to your query.

```
mpak['PAKNYGDP??????'].names
```

```
['PAKNYGDPDISCCN',
 'PAKNYGDPDISCKN',
 'PAKNYGDPFCSTCN',
 'PAKNYGDPFCSTKN',
 'PAKNYGDPFCSTXN',
 'PAKNYGDPMKTPCD',
 'PAKNYGDPMKTPCN',
 'PAKNYGDPMKTPKD',
 'PAKNYGDPMKTPKN',
 'PAKNYGDPMKTPXN',
 'PAKNYGDPPOTLKN']
```

18.16.5 .rename() replaces the variable mnemonic with its description

Use .rename() to display results using the variable description instead of the mnemonic, which is the default behavior.

```
mpak['PAKNYGDPMKTPKN PAKNECONPRVTKN'].rename().df
```

	Real GDP	HH. Cons Real
2020	2.648605e+07	2.338267e+07
2021	2.679140e+07	2.358771e+07
2022	2.691168e+07	2.368379e+07
2023	2.710269e+07	2.385588e+07

18.16.6 Transformations of solution results

When the variables has been selected through the index operator a number of standard data transformations can be performed.

Transformation	Meaning	expression
growth (pct)	Growth rates	$\frac{this_t}{this_{t-1}} - 1$
dif	Difference in level	$l - b$
difgrowth (difpct)	Difference in growth rate	$\left[\frac{l_t}{l_{t-1}} - 1 \right] - \left[\frac{b_t}{b_{t-1}} - 1 \right]$
difpctlevel	difference in level as a pct of baseline	$\frac{l_t - b_t}{b_t}$

- *this* is the chained value. Default lastdf but if preceded by .base the values from .basedf will be used
- *b* is the values from .basedf
- *l* is the values from .lastdf

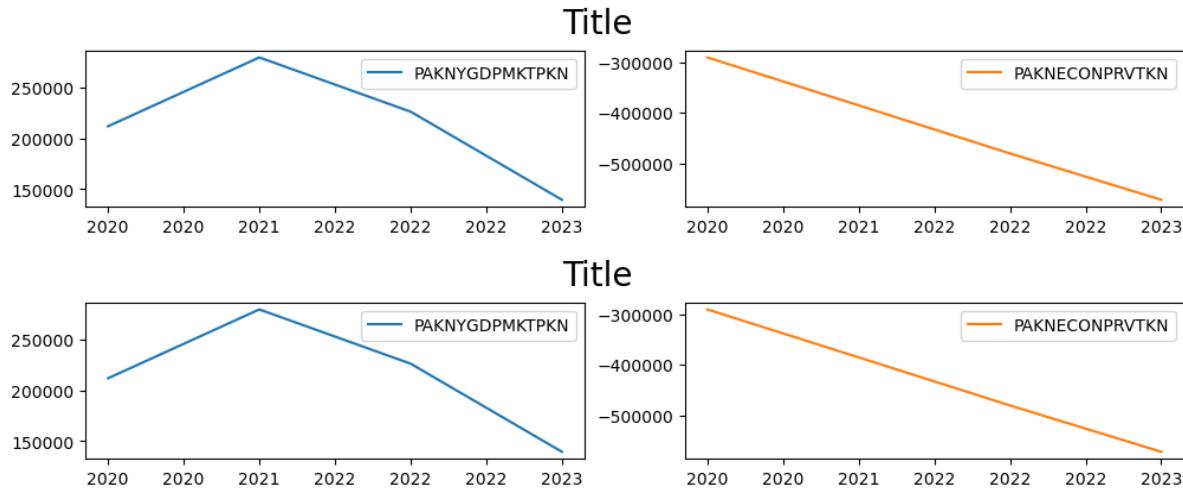
18.16.7 .dif Difference in level

The ‘dif’ command displays the difference in levels of the latest and previous solutions.

$$l - b$$

where l is the variable from the .lastdf and b is the variable from .baseddf.

```
mpak [ 'PAKNYGDPMKTPKN PAKNECONPRVTKN' ].dif.plot()
```

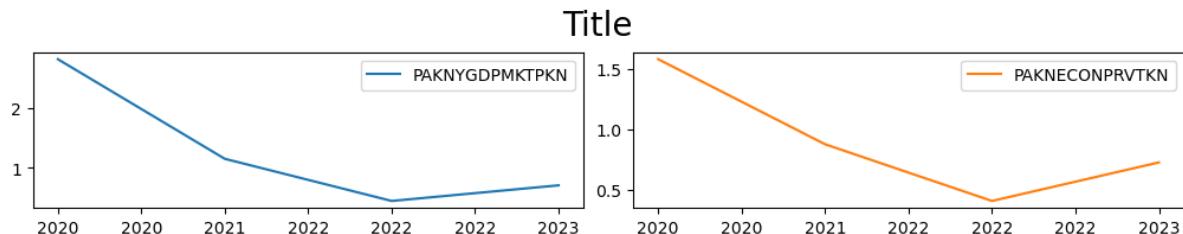


18.16.8 .growth (.pct) Growth rates

Display growth rates

$$\left(\frac{l_t}{l_{t-1}} - 1 \right)$$

```
mpak [ 'PAKNYGDPMKTPKN PAKNECONPRVTKN' ].growth.plot();
```

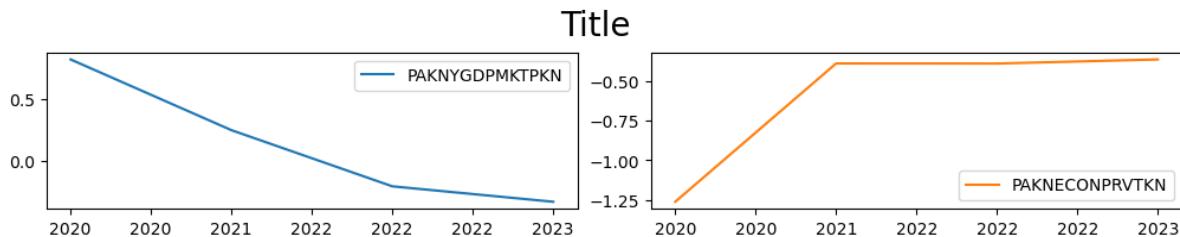


18.16.9 .difpct property difference in growth rates

The difference in the growth rates between the last and base datafram.

$$\left(\frac{l_t}{l_{t-1}} - 1 \right) - \left(\frac{b_t}{b_{t-1}} - 1 \right)$$

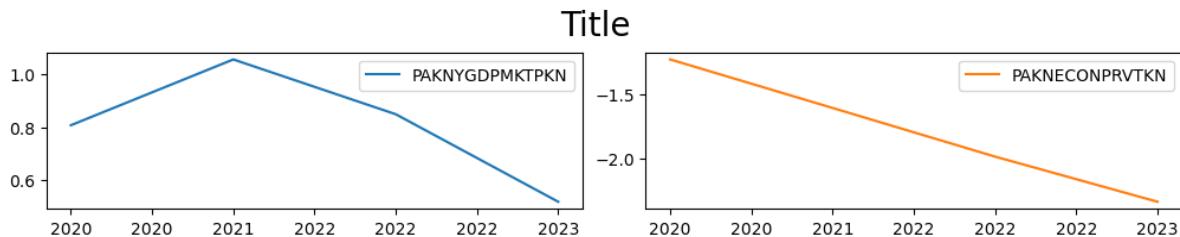
```
mpak [ 'PAKNYGDPMKTPKN PAKNECONPRVTKN' ].difpct.plot() ;
```



18.16.10 .difpctlevel percent difference of levels

$$\left(\frac{l_t - b_t}{b_t} \right)$$

```
mpak [ 'PAKNYGDPMKTPKN PAKNECONPRVTKN' ].difpctlevel.plot() ;
```



18.16.11 .oyy_ar Growth over 4 periods

This should only be used for quarterly data

$$\left(\frac{l_t - b_{t-4}}{b_{t-4}} \right) - 1$$

18.16.12 .qoq_ar Annualized quarterly growth rate

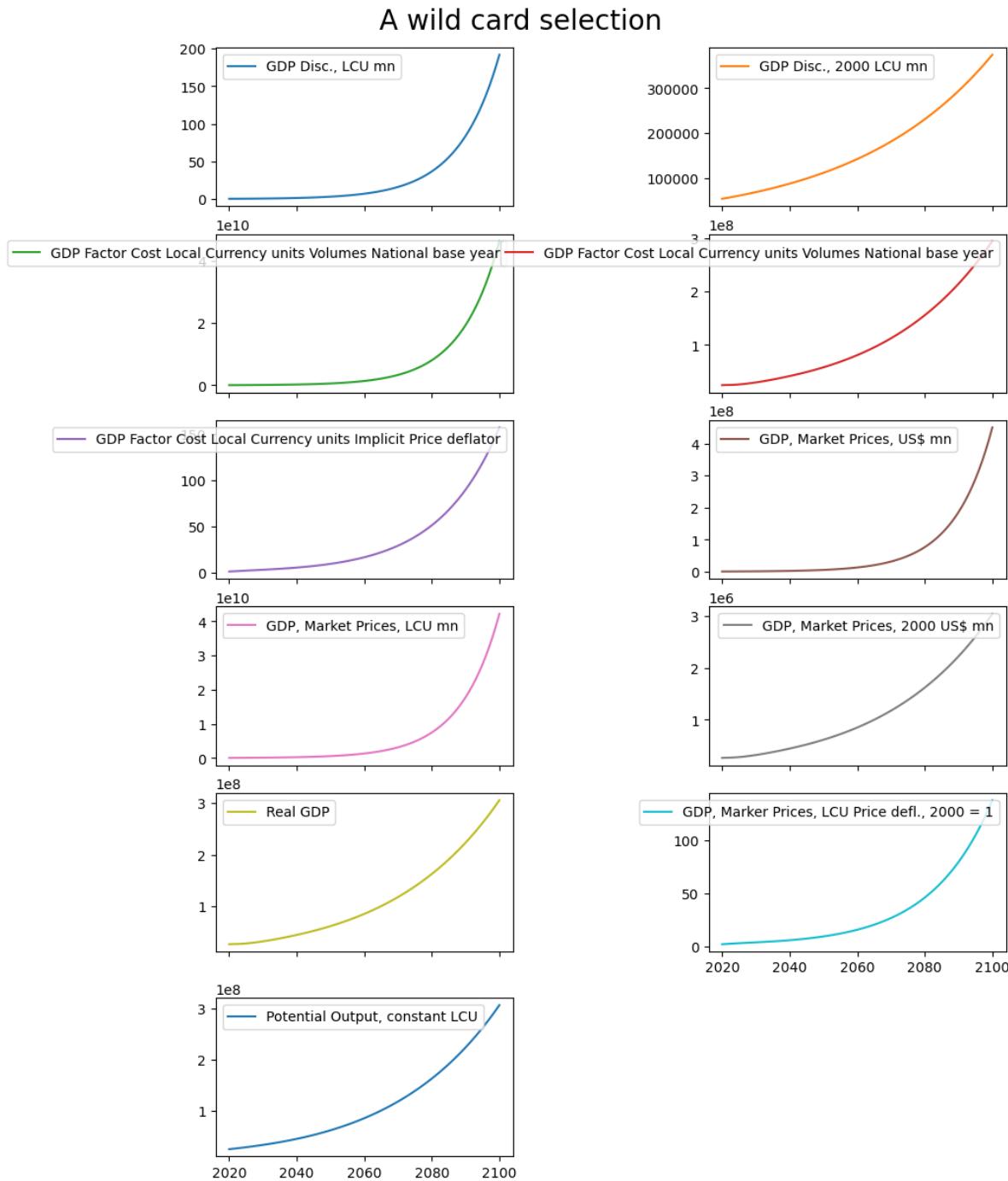
Also for quarterly data

$$\left(\frac{l_t - b_t}{b_t} \right)^4 - 1$$

18.17 .plot chart the selected and transformed variables

After the variables has been selected and transformed, they can be plotted. The .plot() method plots the selected variables separately

```
mpak.smpl(2020, 2100);
mpak['PAKNYGDP??????'].rename().plot(title="A wild card selection");
```



18.17.1 Options to plot()

Common:

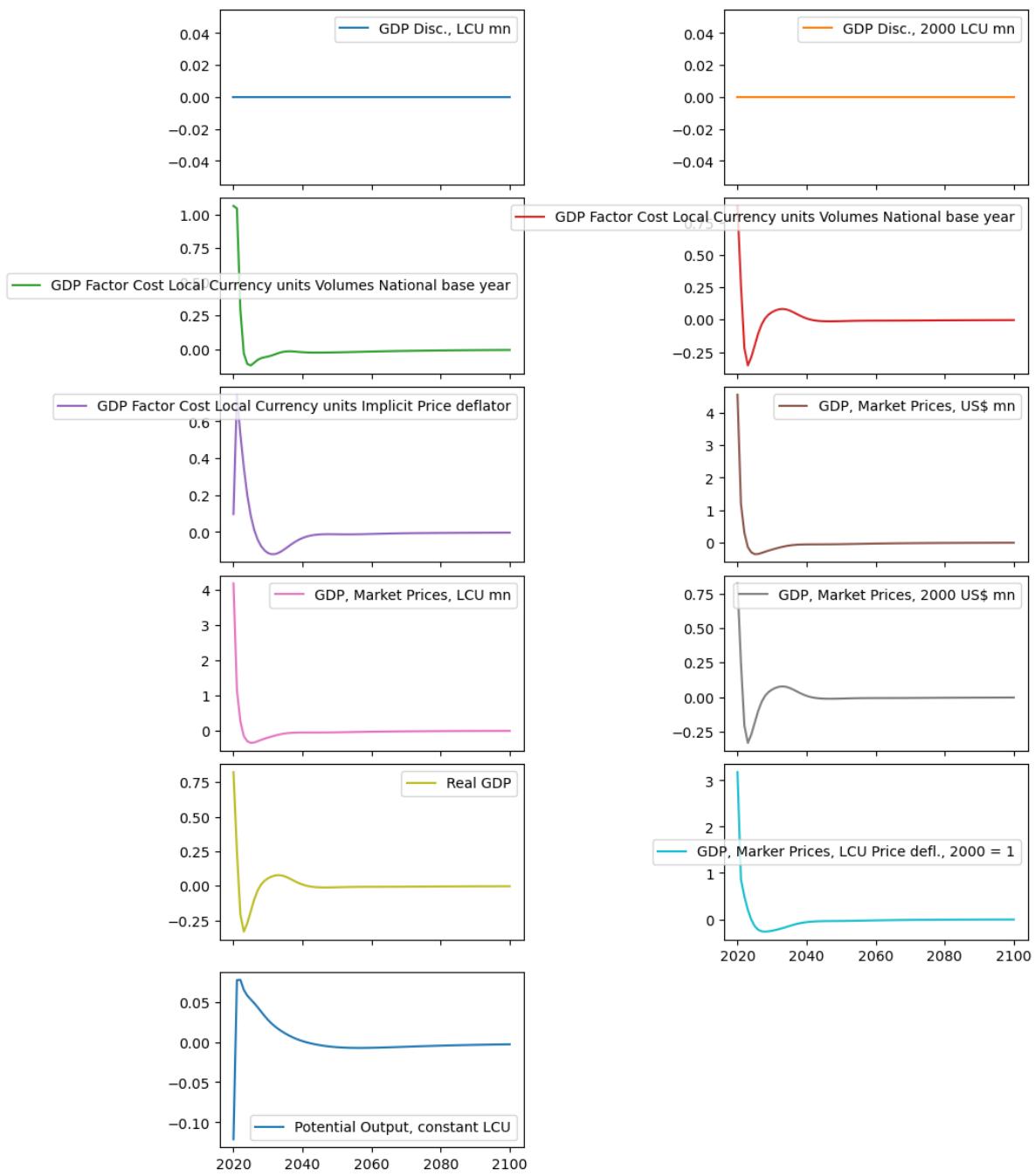
- title (optional): title. Defaults to “”.
- colrow (TYPE, optional): Columns per row . Defaults to 2.
- sharey (TYPE, optional): Share y axis between plots. Defaults to False.

More exotic:

- splitchar (TYPE, optional): If the name should be split . Defaults to ‘__’.
- savefig (TYPE, optional): Save figure. Defaults to “”.
- xsize (TYPE, optional): x size default to 10
- ysize (TYPE, optional): y size per row, defaults to 2
- ppos (optional): # of position to use if split. Defaults to -1.
- kind (TYPE, optional): Matplotlib kind . Defaults to ‘line’.

```
mpak ['PAKNYGDP??????'].difpct.rename().plot(title='Growth of variables that start  
→PAKNYGDP');
```

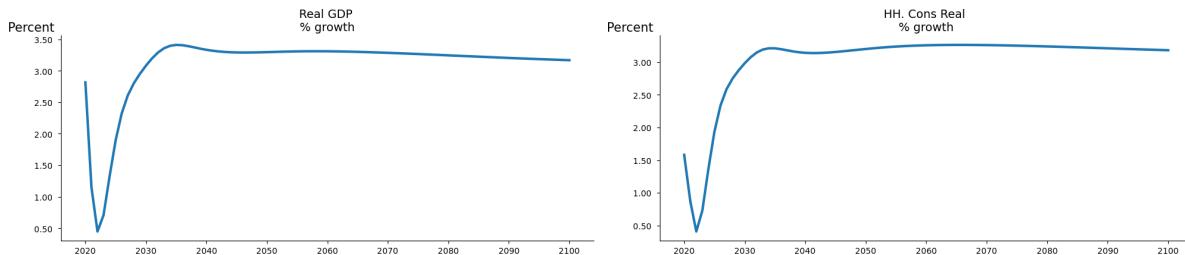
Growth of variables that start PAKNYGDP



18.18 [] .rtable and [] .rplot - reports from []

Report tables and plots can also be created directly from the variable selector `[]`. The purpose is to enable easy reuse of variable selection.

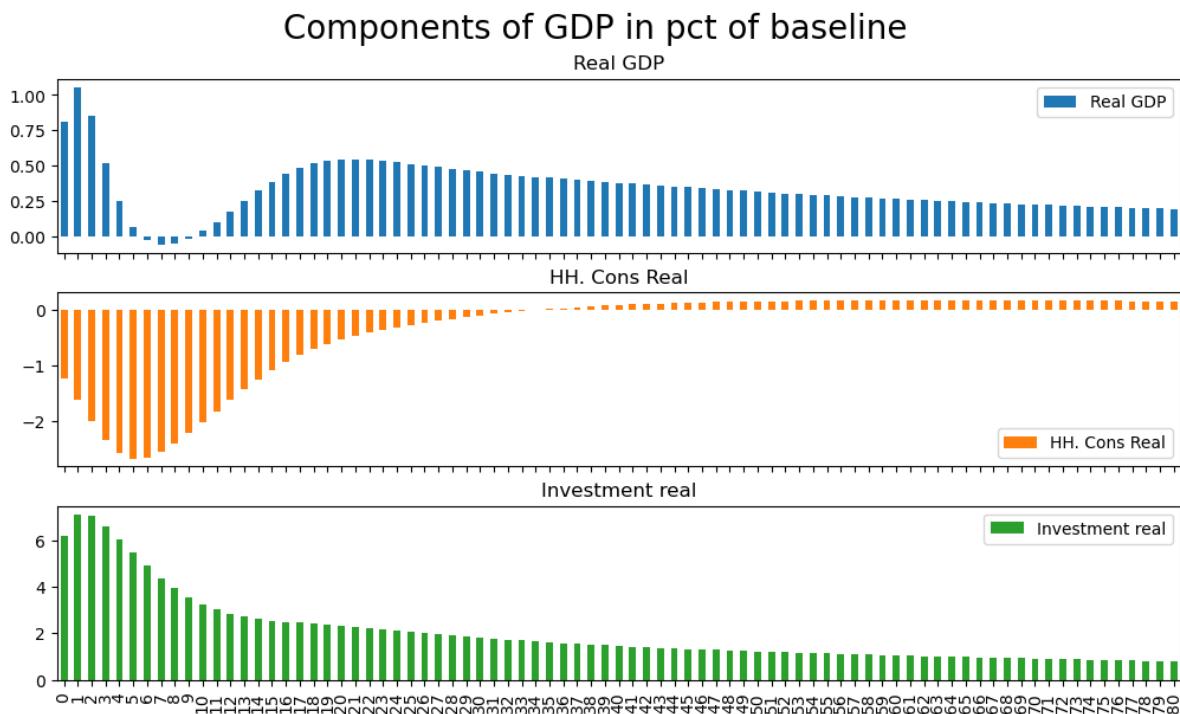
```
mpak [ 'PAKNYGDPMKTPKN PAKNECONPRVTKN' ].rplot ( samefig=1 ). show
```



18.19 Plotting inspiration

The following graph shows the components of GDP using the values of the baseline dataframe.

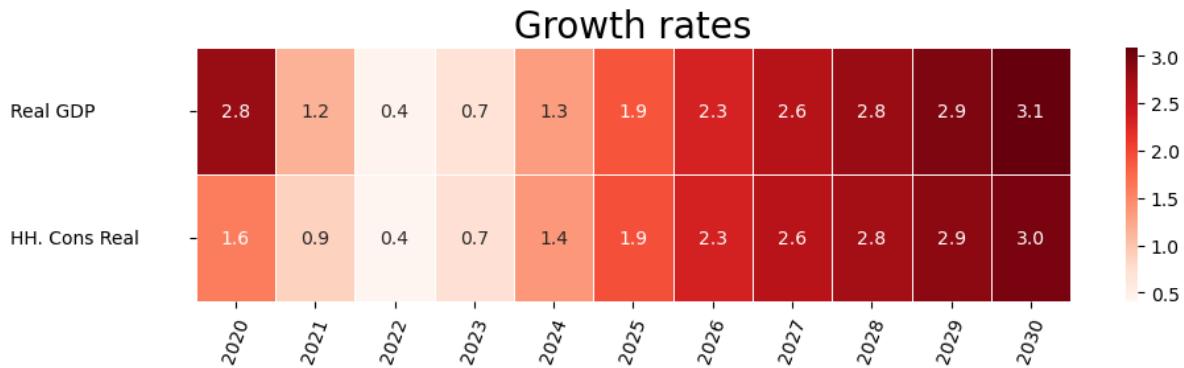
```
mpak [ 'PAKNYGDPMKTPKN PAKNECONPRVTKN PAKNEGDIFFOTKN' ]. \
difpclevel.rename () . \
plot ( title = 'Components of GDP in pct of baseline' , colrow = 1 , kind = 'bar' ) ;
```



18.19.1 Heatmaps

For some model types heatmaps can be helpful, and they come out of the box. This feature was developed for use by bank stress test models.

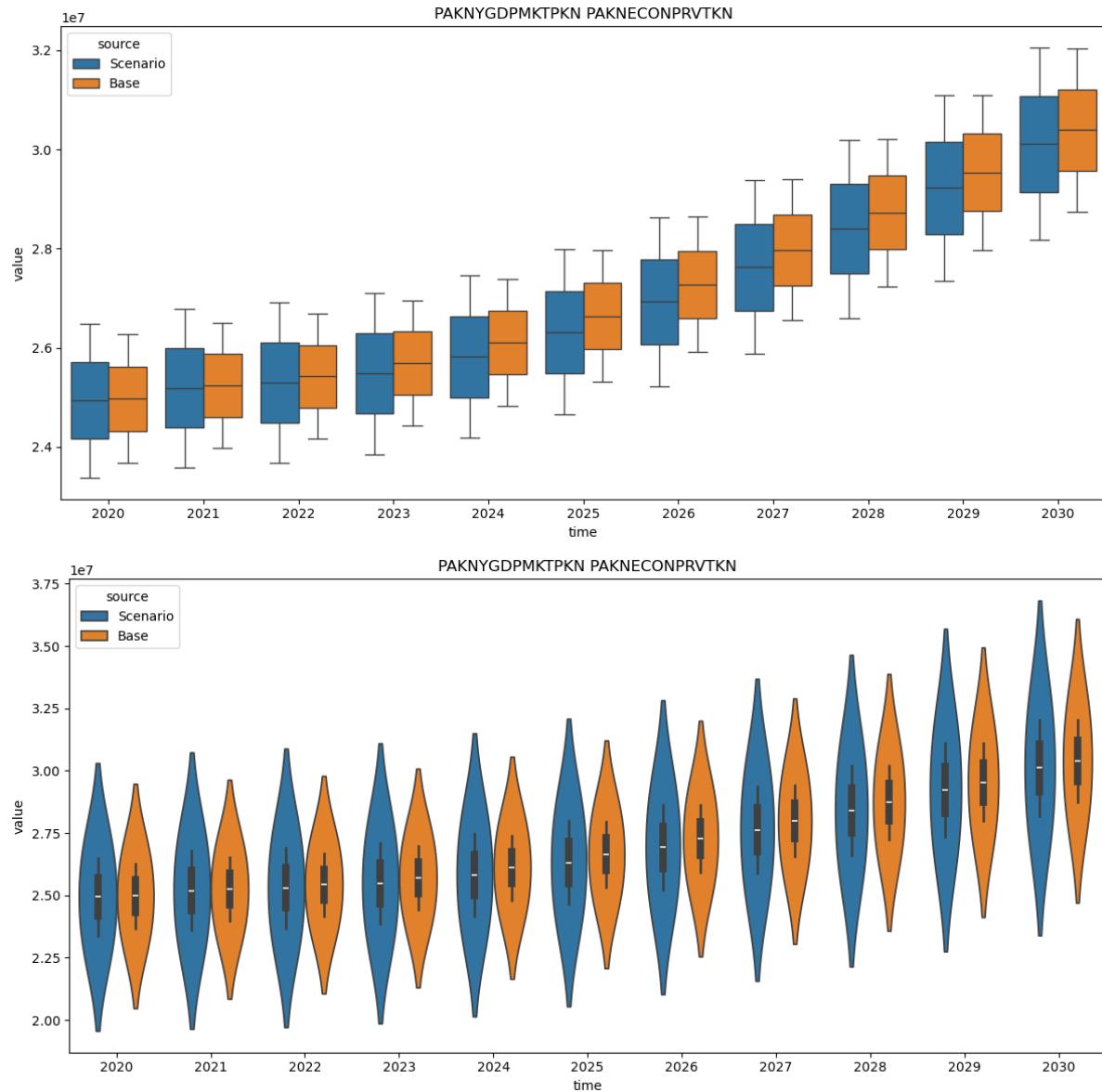
```
with mpak.set_smpl(2020, 2030):
    mpak['PAKNYGDPMKTPKN PAKNECONPRVTKN'].pct.rename().heat(title='Growth rates',
    ↪annot=True, dec=1, size=(10, 3))
```



18.19.2 Violin and boxplots,

Not obvious for macro models, but useful for stress test models with many banks.

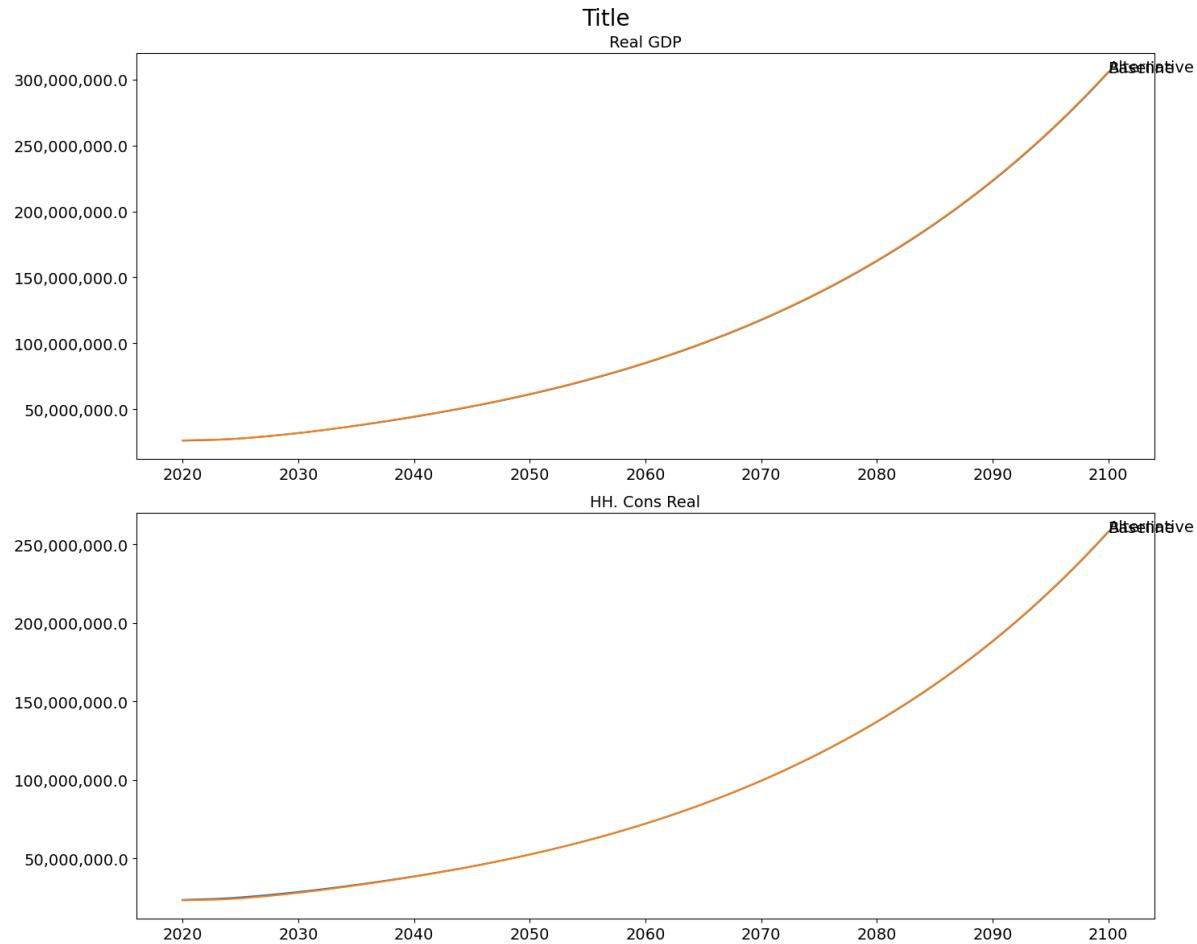
```
with mpak.set_smpl(2020, 2030):
    mpak['PAKNYGDPMKTPKN PAKNECONPRVTKN'].difpct.box()
    mpak['PAKNYGDPMKTPKN PAKNECONPRVTKN'].difpct.violin()
```



18.19.3 Plot baseline vs alternative

A raw routine, only showing levels. To make it really useful it should be expanded.

```
mpak [ 'PAKNYGDPMKTPKN PAKNECONPRVTKN' ].plot_alt() ;
```



18.20 .draw() Graphical presentation of relationships between variables

.draw() helps you understand the relationship between variables in your model better.

The thickness the arrow reflect the attribution of the upstream variable to the impact on the downstream variable.

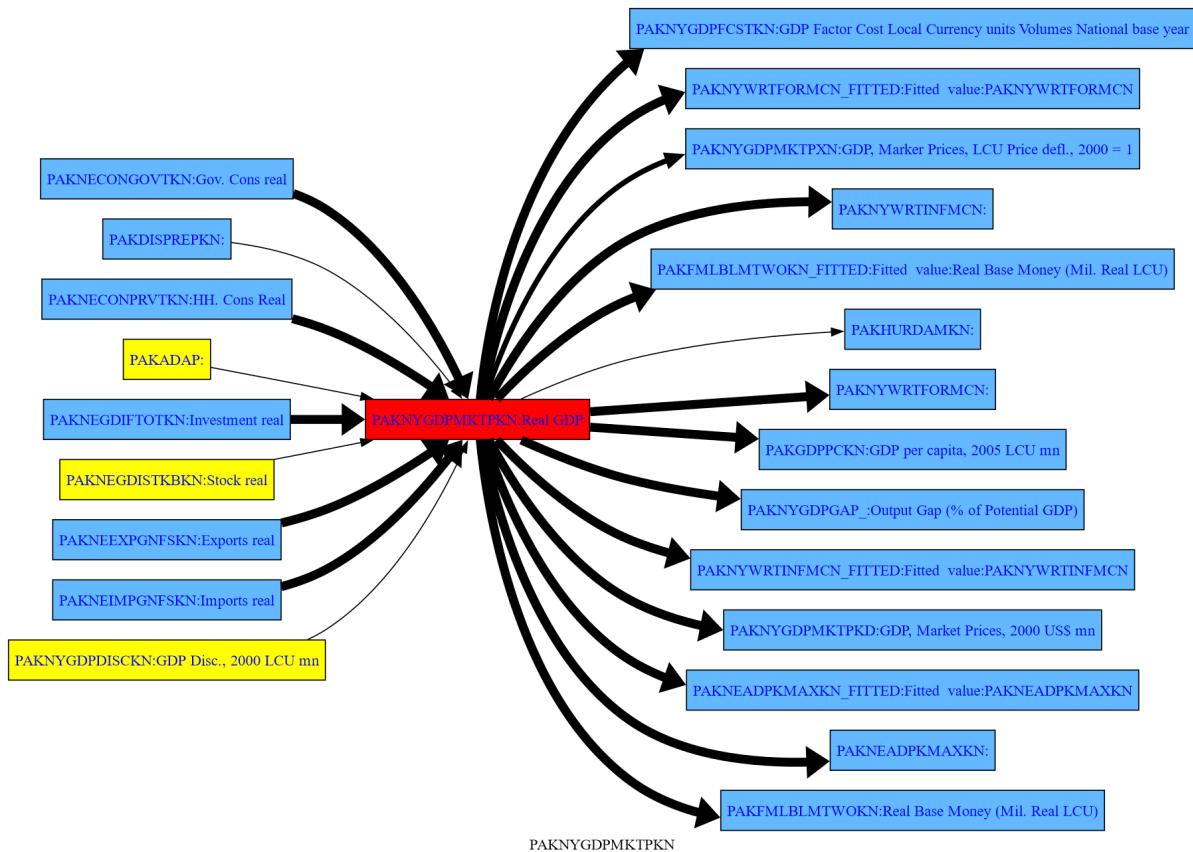
18.20.1 .draw(up = level, down = level)

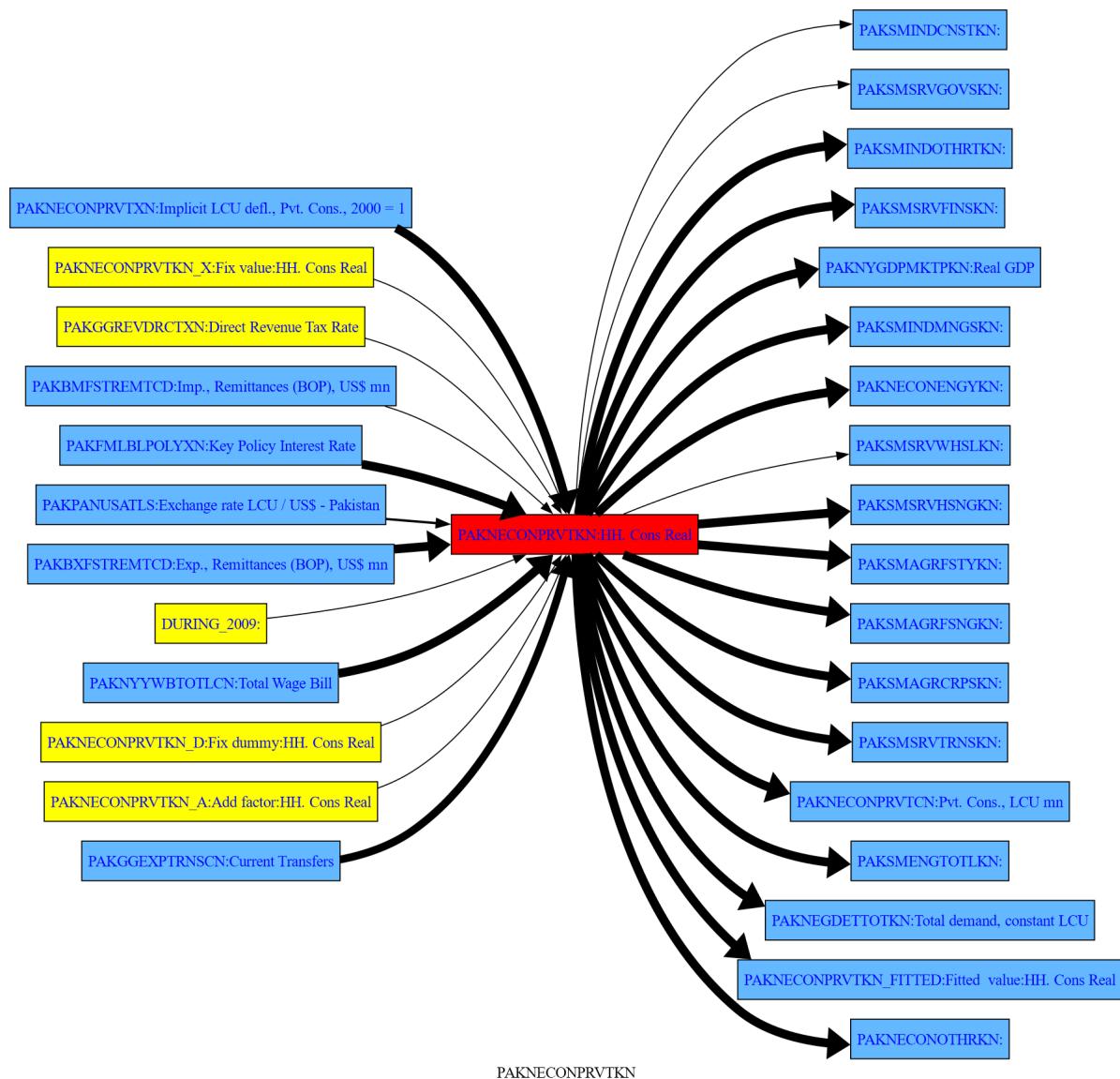
You can specify how many levels up and down you want in your graphical presentation (Needs more explanation).

In this example all variables that depend directly upon GDP and consumption as well as those that are determined by them, are displayed. This means one step upstream in the model logic and one step downstream.

More on the how to visualize the logic structure [here](#)

```
mpak['PAKNYGDPMKTPKN PAKNECONPRVTKN'].draw(up=1, down=1, png=latex) # diagram of all
→direct dependencies
```

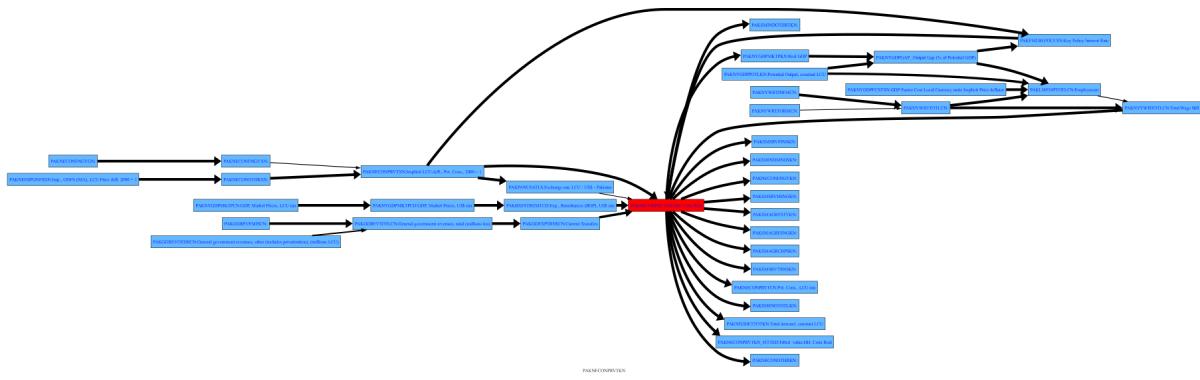




18.20.2 .draw(filter =<minimal impact>)

By specifying filter= only links where the minimal impact is more than <minimal impact> are show. In this case 20%

```
mpak [ 'PAKNECONPRVTKN' ].draw (up=3, down=1, filter=20, png=latex)
```



18.21 Attribution/decomposition

Attribution is about taking the total derivative of a function. This can be done at two levels:

- **single equation** To investigate the impact of changes in right hand variables on the left hand variables
- **model** To investigate the impact of changes in exogenous variables on selected endogenous variables

18.21.1 Single equations attribution

This can either be done where the changes in variables are between:

- A baseline and an alternative (usually `.basedf` and `.lastdf` (**default**))
- From year to year.

`.dekomp()` Attribution of right hand side variables to change in left hand side.

The `.dekomp()` function is the engine for calculates the contributions of the right hand side variables to the observed change in the left hand side variables.

The user will usual benefit from using some of the wrappers which can display the results of `.dekomp()`

Parameter	Description
<code>varnavn</code>	Input variable name.
<code>start</code>	Start period for retrieving variable values (default is “”).
<code>end</code>	End period for retrieving variable values (default is “”).
<code>basedf</code>	Base dataframe to use (default is None, meaning ‘ <code>.basedf</code> ’).
<code>altdf</code>	Alternative dataframe to use (default is None, meaning <code>.lastdf</code>).
<code>lprint</code>	Flag to print the results (default is True).
<code>time_att</code>	Flag to do a timewise attribute (default is False).

`.dekomp()` returns a named tuple with the results as Dataframes.

Field	Meaning
diff_level	DataFrame with level differences.
att_level	DataFrame with attributions to the level difference.
att_pct	DataFrame with the share of attributions to the difference in level.
diff_growth	DataFrame with differences in growth rate.
att_growth	DataFrame with attributions to the difference in growth rate. (not accurate)

```
mpak.dekomp.cache_clear()
with mpak.set_smpl(2021, 2022):
    dekomp_result = mpak.dekomp('PAKNYGDPMKTPKN', lprint=1) # frml attribution
```

```
Formula          : FRML <IDENT> PAKNYGDPMKTPKN =_
    ↵PAKNECONPRVTKN+PAKNECONGOVTKN+PAKNEGIDFTOTKN+PAKNEGDISTKBKN+PAKNEEXPGNFSKN-
    ↵PAKNEIMPGNFSKN+PAKNYGDPDISCKN+PAKADAP*PAKDISPREPKN $
    2021           2022

Variable      lag
Base          0   26511370.45 26685141.90
Alternative  0   26791399.30 26911678.26
Difference   0   280028.85 226536.36
Percent      0   1.06       0.85
Contributions to difference for PAKNYGDPMKTPKN
    2021           2022
Variable      lag
PAKNECONPRVTKN 0   -385108.31 -480338.16
PAKNECONGOVTKN 0   348838.85 308073.15
PAKNEGIDFTOTKN 0   218058.91 217582.81
PAKNEGDISTKBKN 0   -0.02     -0.01
PAKNEEXPGNFSKN 0   -3355.88 -6206.24
PAKNEIMPGNFSKN 0   101595.21 187424.78
PAKNYGDPDISCKN 0   -0.02     -0.01
PAKADAP        0   -0.02     -0.01
PAKDISPREPKN   0   -0.02     -0.01
Share of contributions to difference for PAKNYGDPMKTPKN
    2021           2022
Variable      lag
PAKNECONGOVTKN 0   125%      136%
PAKNEGIDFTOTKN 0   78%       96%
PAKNEIMPGNFSKN 0   36%       83%
PAKNEGDISTKBKN 0   -0%       -0%
PAKNYGDPDISCKN 0   -0%       -0%
PAKADAP        0   -0%       -0%
PAKDISPREPKN   0   -0%       -0%
PAKNEEXPGNFSKN 0   -1%       -3%
PAKNECONPRVTKN 0   -138%     -212%
Total          0   100%      100%
Residual       0   -0%       -0%
Difference in growth rate PAKNYGDPMKTPKN
    2021           2022
Variable      lag
Base          0   0.9%      0.7%
Alternative  0   1.2%      0.4%
Difference   0   0.2%     -0.2%
None
Contribution to growth rate PAKNYGDPMKTPKN
    2021           2022
```

(continues on next page)

(continued from previous page)

Variable	lag		
PAKNECONPRVTKN	0	-0.3%	-0.3%
PAKNECONGOVTKN	0	0.1%	-0.2%
PAKNEGDIFTOTKN	0	0.1%	-0.0%
PAKNEGDISTKBKN	0	0.0%	0.0%
PAKNEEXPGNFSKN	0	-0.0%	-0.0%
PAKNEIMPGNFSKN	0	0.4%	0.3%
PAKNYGDPDISCKN	0	0.0%	0.0%
PAKADAP	0	0.0%	0.0%
PAKDISPREPKN	0	0.0%	0.0%
Total	0	0.3%	-0.2%
Residual	0	0.0%	0.0%

The content of the returned namedtuple

```
for name, value in dekomp_result._asdict().items():
    print(f"{name}:\n {value}\n")
```

diff_level:			
		2021	2022
Variable	lag		
Base	0	26511370.451597	26685141.900638
Alternative	0	26791399.301257	26911678.264054
Difference	0	280028.849661	226536.363416
Percent	0	1.056259	0.848923
att_level:			
		2021	2022
Variable	lag		
PAKNECONPRVTKN	0	-385108.305552	-480338.157967
PAKNECONGOVTKN	0	348838.846058	308073.148036
PAKNEGDIFTOTKN	0	218058.909007	217582.809019
PAKNEGDISTKBKN	0	-0.017471	-0.00759
PAKNEEXPGNFSKN	0	-3355.878372	-6206.242849
PAKNEIMPGNFSKN	0	101595.208089	187424.776355
PAKNYGDPDISCKN	0	-0.017471	-0.00759
PAKADAP	0	-0.017471	-0.00759
PAKDISPREPKN	0	-0.017471	-0.00759
att_pct:			
		2021	2022
Variable	lag		
PAKNECONGOVTKN	0	124.572467	135.992802
PAKNEGDIFTOTKN	0	77.870158	96.047630
PAKNEIMPGNFSKN	0	36.280265	82.734963
PAKNEGDISTKBKN	0	-0.000006	-0.000003
PAKNYGDPDISCKN	0	-0.000006	-0.000003
PAKADAP	0	-0.000006	-0.000003
PAKDISPREPKN	0	-0.000006	-0.000003
PAKNEEXPGNFSKN	0	-1.198405	-2.739623
PAKNECONPRVTKN	0	-137.524511	-212.035786
Total	0	99.999950	99.999973
Residual	0	-0.000050	-0.000027
diff_growth:			
		2021	2022
Variable	lag		
Base	0	0.903664	0.655460
Alternative	0	1.152860	0.448946

(continues on next page)

(continued from previous page)

Difference	0	0.249196	-0.206514
att_growth:			
		2021	2022
Variable	lag		
PAKNECONPRVTKN	0	-3.418828e-01	-3.440505e-01
PAKNECONGOVTKN	0	9.539784e-02	-1.600896e-01
PAKNEGDIFTOTKN	0	1.061517e-01	-5.475665e-03
PAKNEGDISTKBKN	0	5.848919e-08	3.717548e-08
PAKNEEXPGNFSKN	0	-8.896135e-03	-1.058154e-02
PAKNEIMPGNFSKN	0	4.020092e-01	3.198729e-01
PAKNYGDPDISCKN	0	5.848919e-08	3.717548e-08
PAKADAP	0	5.848919e-08	3.717548e-08
PAKDISPREPKN	0	5.848919e-08	3.717548e-08
Total	0	2.527800e-01	-2.003243e-01
Residual	0	3.584398e-03	6.189603e-03

Useful wrappers of single equation attribution

Command	Explanation
Variable attribution	
<code><modelinstance>.variable_name.dekomp()</code>	Perform a decomposition analysis for the variable.
<code><modelinstance>[pattern].dekomp()</code>	Perform decomposition analysis on variables matching the given pattern.
Attribution Extraction	Use <code>threshold=<number></code> to limit small attributions
<code><modelinstance>.get_att_pct('<variable name>')</code>	Get the percentage attribution for the variable.
<code><modelinstance>.get_att_level('<variable name>')</code>	Get the level attribution for the variable.
Plotting attribution	Use <code>threshold=<number></code> to limit small attributions
<code><modelinstance>.dekomp_plot('<variable name>')</code>	Plot the decomposition analysis results for the variable.
<code><modelinstance>.dekomp_plot_per('<variable name>')</code>	Plot decomposition results for a specific period for the variable.
<code><modelinstance>.get_dekom_gui('<variable name>')</code>	Display an interactive GUI for performing decomposition analysis.
Enhance causal graph with attribution	Use <code>filter=<percent number></code> to prune causality graph
<code><modelinstance>.variable_name.draw()</code>	Generate a visual representation of the variable's relationships.
<code><modelinstance>.variable_name.tracepre()</code>	Trace the predecessors of the variable to understand upstream dependencies.
<code><modelinstance>.variable_name.tracedep()</code>	Trace the dependents of the variable to understand downstream effects.
<code><modelinstance>.modeldash('<variable name>')</code>	Display an interactive dashboard for analyzing the variable.

18.21.2 Model Attribution

The method `.totdif()` returns an instance of the **totdif class**.

It works by solving the model multiple times, each time modifying one of the right-hand side variables and calculating the impact on all dependent variables. By default, it uses the values from the `.lastdf` DataFrame as the shock values and the values in `.basedf` as the baseline (initial) values. Separate simulations are run for every exogenous (or exogenized) variable that has changed between the two DataFrames.

For advanced users, the right-hand side (RHS) variables can be grouped into user-defined blocks. This feature is particularly useful when there are many changes, as it helps identify the primary causal pathways.

The result of the attribution analysis is stored in the `.res` property of the `totdif` class instance. This property contains a dictionary with two keys: 'level' and 'growth'. These represent the impacts of each exogenous variable or group on the level and growth metrics, respectively.

The `totdif` class also provides several methods to display the results of the attribution analysis. This eliminates the need to directly access the `.res` property unless custom output is required beyond the standard visualizations.

The `.totdif()` Method

In most cases, there is no need to provide parameters to this method. However, if a model is very large, it can be useful to limit the number of variables included in the analysis. This can be controlled using the `summaryvar` parameter.

In certain circumstances, it can also be helpful to group exogenous variables into categories that are analyzed as a whole. For example, in a model with many countries, it might make sense to group shocks by country. Alternatively, it could be useful to group shocks by type across all countries. This grouping can be achieved by setting the `experiments` parameter.

Parameter	Type	Description	De-fault
<code>summaryvar</code>	<code>str</code> or <code>list</code>	Variables to summarize in the analysis. Use '*' to include all variables.	'*'
<code>experiments</code>	<code>dict</code>	A dictionary where keys are experiment names and values are lists of variables to reset to baseline values. If <code>None</code> , uses all variables with differences.	<code>None</code>

```
totdecomp = mpak.totdif() # Calculate the total derivatives of all equations in the model.
```

```
Total dekomp took : 3.728 Seconds
```

```
totdecomp.res.keys()
```

```
dict_keys(['level', 'growth'])
```

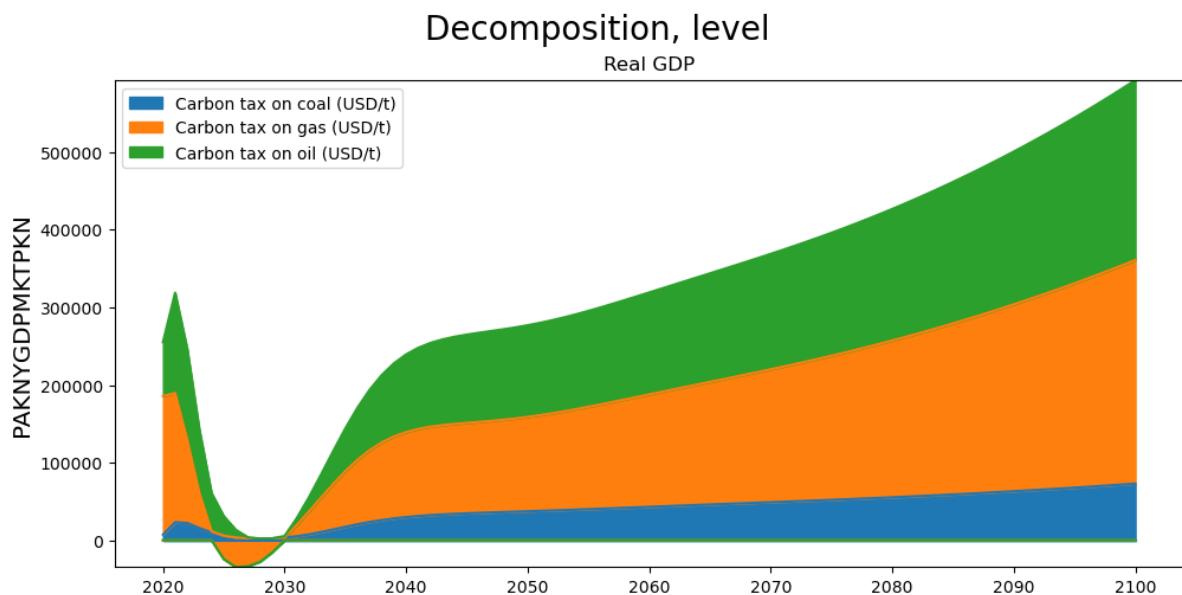
`.totexplain()` to display the results

Parameter	Type	Description	Default
pat	str	Pattern to match variables for attribution analysis.	'*'
vtype	str	Type of data to attribute. Options: 'all', 'per', 'last', 'sum'.	'all'
stacked	bool	Whether to stack the data in the visualization (relevant for bar or area plots).	True
kind	str	Type of plot to generate. Options: 'bar', 'line', 'area', etc.	'bar'
per	str or int	Specific period to analyze (used when vtype is 'per').	'' (empty)
title	str	Title for the plot. If empty, a default title will be generated based on the analysis type.	'' (empty)
use	str	Type of decomposition to use. Options: 'level', 'growth'.	'level'
threshold	float	Minimum impact value to include in the visualization (filters out smaller impacts).	0.0
ysize	int	Height of the plot in inches.	10
**kwargs	dict	Additional keyword arguments passed to underlying visualization methods.	N/A

The meaning of `vtype` is:

vtype Value	Description
'all'	Performs attribution analysis across all periods and provides a complete decomposition overview.
'per'	Analyzes and visualizes the decomposition for a specific period.
'last'	Focuses on the decomposition of the last period in the data.
'sum'	Summarizes and visualizes the cumulative impact across all periods.

```
totdekomp.totexplain('PAKNYGDPMKTPKN', kind='area', stacked=True);
```

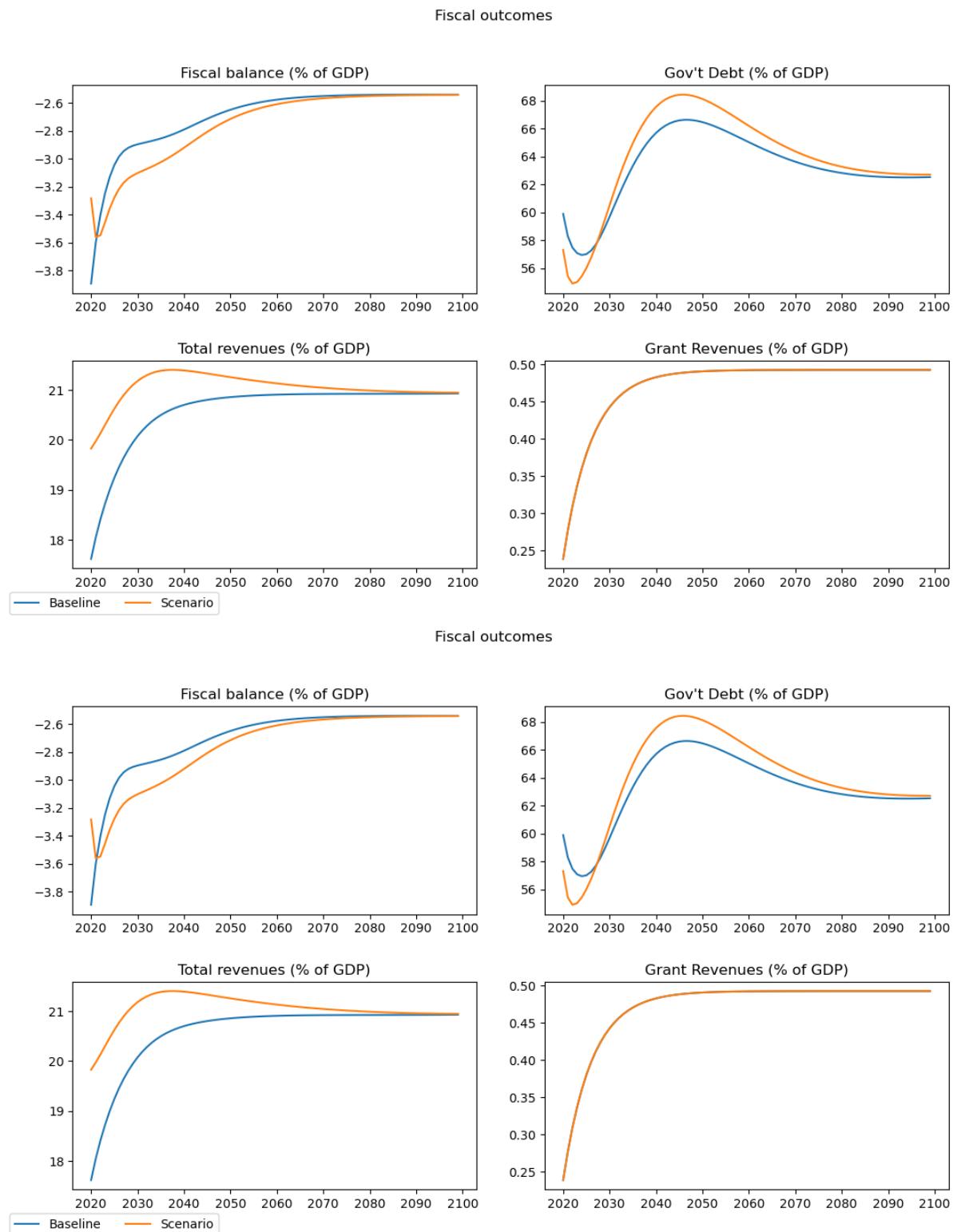


18.22 Bespoken plots using matplotlib (or plotly -later) (should go to a separate plot book)

The predefined plots are not necessary created for presentation purpose. To create bespoke plots they can be constructed directly in python scripts. The two main libraries are matplotlib, plotly but any other python plotting library can be used. Here is an example using matplotlib.

18.23 Plot four separate plots of multiple series in grid

```
figure, axs= plt.subplots(2,2, figsize=(11, 7))
axs[0,0].plot(mpak.basedf.loc[2020:2099, 'PAKGGBALOVRLCN_'], label='Baseline')
axs[0,0].plot(mpak.lastdf.loc[2020:2099, 'PAKGGBALOVRLCN_'], label='Scenario')
#axs[0,0].legend()
axs[0,1].plot(mpak.basedf.loc[2020:2099, 'PAKGDBTTOTLCN_'], label='Baseline')
axs[0,1].plot(mpak.lastdf.loc[2020:2099, 'PAKGDBTTOTLCN_'], label='Scenario')
axs[1,0].plot(mpak.basedf.loc[2020:2099, 'PAKGGREVTO TLCN']/mpak.basedf.loc[2020:2099,
    ↪'PAKNYGDPMKTPCN']*100, label='Baseline')
axs[1,0].plot(mpak.lastdf.loc[2020:2099, 'PAKGGREVTO TLCN']/mpak.lastdf.loc[2020:2099,
    ↪'PAKNYGDPMKTPCN']*100, label='Scenario')
axs[1,1].plot(mpak.basedf.loc[2020:2099, 'PAKGGREVGRNTCN']/mpak.basedf.loc[2020:2099,
    ↪'PAKNYGDPMKTPCN']*100, label='Baseline')
axs[1,1].plot(mpak.lastdf.loc[2020:2099, 'PAKGGREVGRNTCN']/mpak.lastdf.loc[2020:2099,
    ↪'PAKNYGDPMKTPCN']*100, label='Scenario')
#axs2[4].plot(mpak.lastdf.loc[2000:2099, 'PAKGGREVGRNTCN']/mpak.basedf.loc[2000:2099,
    ↪'PAKNYGDPMKTPCN']*100, label='Scenario')
axs[0,0].title.set_text("Fiscal balance (% of GDP)")
axs[0,1].title.set_text("Gov't Debt (% of GDP)")
axs[1,0].title.set_text("Total revenues (% of GDP)")
axs[1,1].title.set_text("Grant Revenues (% of GDP)")
figure.suptitle("Fiscal outcomes")
plt.figlegend(['Baseline', 'Scenario'], loc='lower left', ncol=5)
figure.tight_layout(pad=2.3) #Ensures legend does not overlap dates
figure
```



Part VII

Backmatter

BIBLIOGRAPHY

- [1] Doug Addison. *The World Bank revised minimum standard model (RMSM) : concepts and issues*. Number WPS231 in Policy Research Working Papers. World Bank, Washington DC., 1989. URL: <https://documents.worldbank.org/en/publication/documents-reports/documentdetail/997721468765042532/the-world-bank-revised-minimum-standard-model-rmsm-concepts-and-issues>.
- [2] Ron Berndsen. Causal ordering in economic models. *Decision Support Systems*, 1995. doi:10.1016/0167-9236(94)00034-P ([https://doi.org/10.1016/0167-9236\(94\)00034-P](https://doi.org/10.1016/0167-9236(94)00034-P)).
- [3] 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> (<https://doi.org/https://doi.org/10.1093/oxrep/grx045>).
- [4] 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>.
- [5] Andrew Burns and Charl Jooste. *Estimating and Calibrating MFMMod: A Panel Data Approach to Identifying the Parameters of Data Poor Countries in the World Bank's Structural Macro Model*. Number 8939 in Policy Research Working Papers. World Bank, Washington DC., 2019. URL: <https://openknowledge.worldbank.org/entities/publication/2122957a-fa68-5761-8141-5f54934668bd>.
- [6] 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>.
- [7] 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>.
- [8] Hollis Chenery. *Studies in Development Planning*. Harvard University Press,, Cambridge, MA., 1971.
- [9] Kamal Dervis and Sherman Robinson. *General Equilibrium Models for Development Policy*. Cambridge University Press, Cambridge, UK, 1982.
- [10] K.C. Kogiku. *An Introduction to Macroeconomic Models*. McGraw-Hill, 1968. URL: <https://books.google.de/books?id=jp4LzQEACAAJ>.
- [11] A Meeraus. *General Algebraic Modeling System (GAMS), User's Guide: Version 1.0*. World Bank, Development Research Center, Washington DC., 1982.
- [12] J. Tinbergen. *Economic Policy: Principles and Design*. Contributions to economic analysis. North-Holland Publishing Company, 1967. URL: <https://books.google.de/books?id=hA5gAAAAIAAJ>.

- [13] David Vines and Samuel Wills. The rebuilding macroeconomic theory project part II: multiple equilibria, toy models, and policy models in a new macroeconomic paradigm. *Oxford Review of Economic Policy*, 2020. URL: <https://doi.org/10.1093/oxrep/graa066>.
- [14] M. R. Wickens and T. S. Breusch. Dynamic Specification, the Long-Run and the Estimation of Transformed Regression Models. *The Economic Journal*, 98:189–205, April 1988.

INDEX

.current_per, Master time index
model instance, 281
.set_smpl
 limiting the display time frame - example, 111
.upd()
 Setting the time period to the entire sample, 54
operator
 Instructs the search algorithim to restrict itself to groups, 89

Accessing WB models, 77
Add a group, 89
Add-factors
 Use in simulations, 119
Anaconda, 257
 Installation, 15
 start session, 31
Anaconda/MiniConda
 Updating model flow, 18

Balance of Payments, 11
baseddf/lastdf
 Storage system, 160, 278
Behavioral Equations, 9, 102
Boxes
 Box 1. Opening the Anaconda/MiniConda prompt under windows, 16
 Box 2. Time scope of .upd() commands, 54
 Box 3. World Bank Mnemonics, 87
 Box 4. The steps performed by the `.fix()` method, 114
 Box 5. Endogenous (Add-factor) shocks versus temporarily exogenized shocks, 120
 Box 6. Compilation of a model, 140
 Box 7. Targeting background, 155
 Box 8. `ModelFlow` report writing routines, 159
 Box 9. Where ModelFlow Stores Results, 160
 Box 10. PDFs and latex outputs, 163

Carbon tax example(s)
 Complex simulations, 132
 simple simulation, 126
Colors, 106
Customized Table visualizations, 105, 106

Data display - format output, 91
Data formatting, 91
DataFrame, 39
 .columns.size - # of cols), 42
 .eval() method, 42
 .loc[] method, 44
 .loc[] method specific dates, specific series, 45
 .loc[] set specific cells, 46
 .loc[] specific range, 45
 Add a column, 41
 Color code table, 105
 Customized Table visualizations, 105, 106
 Dated Indexes, 39
 Format output, 105
 leads and lags in ModelFlow, 50
 Limit decimal places, 105
 List columns, 42
 mfcalc() ModelFlow extension for variable transformations, 68
 ModelFlow extensions, 49
 ModelFlow naming conventions, 50
 ModelFlow specific features, 49
 ModelFlow time index, 50
 ModelFlow use, 49
 Revise existing series or column, 41
 Set one or more cells to a specific value, 46
 Slicing, 44
 Style property to generate fancy tables, 105
 upd() ModelFlow extension for easy updating of variables, 51
 Variable descriptions - set or change, 127
Dated Indexes, 39
Decimal places
 Limit using DataFrame styler, 105
decimal points, 91
Dependencies
 .show method, display RHS variables and values from .based and .lastdf, 102
Downloading WB models from github, 77
ECMs, 102
 lambda - the speed of adjustment parameter in ECMs, 104

The Error Correction specification, 104
Equation dependencies, 204
Equations
 .eqdelete() Deleting model equations, 270
 .equpdate() Revising model equations, 132, 270
 .show method, 102
The Eviews representation of an equation(prior to normalization), 94
The normalized representation of an equation, 94
Error Correction models - ECMs, 102
Exogenous variables, 9

Fiscal Accounts, 11
Flow of Funds, 11
Formatting data display, 115
from package import function, 33

gdppct in reports
 Only World Bank conventions, 165
Goal Seeking
 Defining Instruments, 140
 Defining Targets, 140
 solve for instruments, 141
Government Accounts, 11
Groups
 Add a group of variables, 89
 List variables in group, 89
 The # operator instructs search to restrict itself to groups, 89

Help
 Initialize a ModelFlow session, 258
 Modeflow, 258

Impact Decomposition, 216
 examples, 230
 Find all exogenous shocks .exodif() method, 242
 Single equation, 219
 Single equation - impact over time, 252
 Single equation, .get_attr() more output transformations, 227
 Single equation, charts, 233
 Single equation, charts , .dekomp_plot(), 232
 Single equation, cutout threshold, 229
 Single equation, impact accumulated across lags, 228
 Single equation, output as "growth/pct/Level", 229
 Single equation, time dimension, 234
 Single equation, Trace preceding variables, 235
 Single equation, Waterfall graphs, 239
 Whole model .totdif() method, 242
 Whole model - accumulated effects, 249
 Whole model - grouping variables, 250
 Whole model - interactive widget, 247
 Whole model - last year, 248
 Whole model - waterfall, 245

Whole model as charts, 243
importing packages, libraries and modules, 32
Index
 indices for quarterly models, 50
 indices for quarterly models
 Index, 50
Information about model variables
 model instance, 85
Information on equations, 94
Install ModelFlow package, 16
Installation
 Anaconda, 15
 MiniConda, 15
 Python, 14

Jupyter Notebook, 257
 Cell Execution, 25
 cell modes, 24
 Change cell type, 25
 code cells, 24
 common markdown commands, 27
 Create a New Notebook, 23
 Delete, Add, Move cells, 24
 Executing python code, 26
 Introduction, 21
 JN cells, 23
 markdown - Display code, 27
 Markdown cells, 26
 markdown cells, 24
 Markdown rendering mathematics, 28
 multi-line equations, 28
 Startup, 22
 Supress output with semicolon ';', 26
 Tables in markdown, 27

Keep
 The Keep option to retain scenario results, 158
keep_solutions
 keep_variables, 160, 278
 Storage system, 160, 278
keep_variables
 keep_solutions, 160, 278
Kinds of simulation, 109

Lags in ModelFlow, 50
Lambda the speed of adjustment in ECMs, 104
Leads in ModelFlow, 50
limiting the display time frame - example
 .set_smpl, 111
list, 36
List variables in group, 89
ljit option, 139

Markdown
 Display code, 27

- multi-line equations, 28
- rendering mathematics, 28
- Markdown commands, 27
- Markdown tables, 27
- Mathplotlib, 257
- mfcalc(), 68
 - Create series, 69
 - Give variable a specific growth rate, 71
 - Multiple equations, 72
 - showeq option, 71
 - Specify timeframe for transformations, 74
 - The diff operator, 72
 - to initialize shocked dataframe, 131
- MFMod
 - Model coverage, 10
- MiniConda
 - Choosing between Anaconda and MiniConda, 14
 - Installation, 15
- Model Adjacency matrix, 206
- Model equations
 - Revise equations with .equupdate(), 133
- Model Information, 205
 - Model Name, 205
 - Model Structure, 205
 - Number of endogenous variables, 205
 - Number of equations, 205
 - Number of exogenous variables, 205
 - Number of variables, 205
- model instance
 - .current_per, Master time index, 281
 - .endogene property, 93
 - .substitution, Defining deferred substitution, 290
 - .var_descriptions, a dictionary of variable descriptions, 292
 - .var_groups, a dictionary of variable groups, 291
 - [] To select and visualize variables, 288
 - Add a group, 89
 - compilation, 139
 - Groups, 89
 - Groups - list variables in group, 89
 - Index operator [], 288
 - information about equations, 93
 - Information about model variables, 85
 - test if mnemonic is endogenous, 93
 - Wildcard search on variable descriptions the ! operator, 288
 - wildcard selection of data - return plot, 92
- model instance()
 - Model simulation, 273
- model instance.<variable>
 - .frml - the normalized representation of an equation, 94
 - .tracedep() Causal tree at the variable level, 210
- .tracedep() down option displays the causal tree of variables that are impacted by the referenced variable, 214
- .tracepre() fokus2 option adds a table of impacts to the causal flow graph, 212
- .tracepre() method - trace influence of causal variables, 207
- .tracepre(); filter option restricts output to variables with a large impact, 211
- tracepre() up option extends the causal tree beyond the initial set of RHS variables, 212
- model instance[]
 - .base Access basedf, 293
 - .df Return a dataframe, 293
 - .dif - difference in level, 294
 - .dif operator, 111
 - .difgrowth/.difpct - difference in growth rate, 294
 - .difpctlevel - difference in level as a pct of baseline, 294
 - .difpctlevel operator, 109
 - .eviews - The Eviews representation of an equation(prior to normalization), 94
 - .frml, 93
 - .growth/.dif Growth rates, 294
 - .names Variable names, 293
 - .qoq_ar - Annualized quarterly growth rate, 294
 - .rename() Rename variables to description, 294
 - .yoy_ar - Growth over 4 periods, 294
 - rplot, 199
 - rtable, 199
 - Transformations, 294
- Model Name, 205
- Model Structure, 204--206
 - .tracedep() Causal tree at the variable level, 210
 - .tracedep() method, 214
 - .tracepre() method, 207
 - Model Adjacency matrix, 206
 - Modeldash method - an interactive display of linkages within a model., 215
 - Recursive Block, 204
 - Recursive equation block, 206
 - Simultaneous block, 204
 - Simultaneous equation block, 206
- Modeldash - an interactive display of linkages within a model., 215
- ModelFlow
 - .equupdate() - revising model equations, 133
 - .set_smpl(begin,end) method, 129
 - .upd() used to initialize shock dataframe, 128
 - Allowed column names, 49
 - Dataframe - variable descriptions, 127
 - DataFrame naming conventions, 50
 - Load model from file, 84
 - ModelFlow environment, 16

time index, 50
types of simulation, 109
ModelFlow - wildcard selection of data - return dataframe, 90
ModelFlow extensions
 DataFrame, 49
ModelFlow Help
 Retrieve descriptions of function options, 258
 The ? operator, 258
ModelFlow Prepare your workspace, 82, 98
ModelFlow use
 DataFrame, 49
ModelFlow versions
 ModelFlow_book vs ModelFlow_stable, 17

National Income Accounts, 11
Number of endogenous variables in model, 205
Number of equations in model, 205
Number of exogenous variables in model, 205
Number of variables in model, 205

Pandas, 34, 257
 DataFrame, 39
 Display Options, 108
 Series, 36
PDF routines under Google Colab, 163
Plot data from wildcard search of data, 92
Plotting scenario results, 111
Preparing your workspace, 82, 98
Python, 257
 classes, 32
 from package import function, 33
 importing packages, libraries and modules, 32
 libraries, 32
 Named colors, 106
 packages, 32
 start session, 29
 start session with anaconda, 31

Recursive block, 204
Recursive equation block, 206
Report plots from selection, 199
Report table from selection, 199
Report writing, 158
Reports
 gdppct - only World Bank conventions, 165
 joining tables, plots and text by the + operator, 189
 Plots, 174
 Plots from selection, 199
 Plots, by_var= , plots by scenario or by variable, 181
 Plots, datatype= transformations of results, 179
 Plots, Joing plots with †, 185
 Plots, options, 179
 Plots, Special scenario: 'base_last', 183
 Table from selection, 199

Tables, 161
Tables .display(table) method - as html :::, 162
Tables .show method - as text, 163
Tables Jupyter Notebook output :::, 162
Tables, A complex table using |, 172
Tables, joining tables with †, 171
Tables; datatype= transformations of results, 164
Tables; Display settings, 164
Text, 187
Vertical tables, 165
Restrict the time period of displayed output, 91
Restricting the amount of data displayed, 111
Return normalized formula of equations, 93
Revising model equations, 132, 270
 .equpdate(), 133

Saving results, 278
 .basedf,.lastdf first and last simulation, 278
 keep= saves to .keep_solutions, 279
 keep_variables= select variables to keep, 279
Scenario set up
 use .upd() to initialize shock dataframe, 128
Scenarios
 .fix() Exogenizing an endogenous variable, 113
 A shock to an exogenous variable, 113
 Changing equations, 132
 Exogenizing an endogenous variable, 113
 Exogenous shock over a limited time period, 116
 Exogenous shocks, 109, 113
 Impact Decomposition examples, 230
 print results, 112
 Print scenario results, 112
 Report writing, 158
 Results - display results as percent change from base-line, 113
 results - plots, 111
 Results - shock-control display, 113
 Simulating a shock to an exogenous variable, 109
 Simulation execution, 110
 Solve the model, 129
 temporarily exogenize endogenous variable, 118
 Temporarily exognization of a behavioral equation, 116
 the Keep option, 158
 use .mfcalc() to initialize shocked dataframe, 131
 using .upd() to create an input DataFrame for a simulation, 114
 Using the upd() KG (KEEP_GROWTH) in an exogenous simulation, 117
Series, 36
 Create Series from dictionary, 38
 Declare with specific index, 37
Setting the time period to the entire sample
 .upd(), 54

- Shock-control display of results, 113
- Simulation types
 - Add-factor based endogenous simulation, 119
 - Exogenizing an endogenous variable, 113
 - Exogenous shocks, 113
 - temporarily exogenize endogenous variable, 118
- Simulations
 - .fix() method, 113
 - .tracedep() Causal tree at the variable level, 210
 - A shock to an exogenous variable, 113
 - Changing equations, 132
 - Endogenous simulation, 119
 - Exogenous shock over a limited time period, 116
 - Exogenous shocks, 109
 - Report writing, 158
 - Results - display results as percent change from base-line, 113
 - Results - shock-control display, 113
 - Simulating a shock to an exogenous variable, 109
 - Solve the model, 129
 - Temporarily exognization of a behavioral equation, 116
 - the keep option, 158
 - use .mfcalc() to initialize shocked dataframe, 131
 - using .upd() to create an input DataFrame for a simulation, 114
 - Using the upd() KG (KEEP_GROWTH) in an exogenous simulation, 117
- Simultaneous equation block, 206
- Slicing
 - DataFrame, 44
- Solve
 - ljit option, 139
- Solve the model, 129
- Speed of adjustment in ECMs, 104
- Storage system
 - basedf/lastdf, 160, 278
 - keep_solutions, 160, 278
- strings
 - multi-line strings, 43
- Style property of Pandas to customize DataFrame outputs, 105
- Targeting
 - Convergence, 157
 - Impulse, 156
 - Max iterations, 157
 - Nonlinearity, 157
 - Tuning, 156
- Targeting a result
 - Defining Instruments, 140
 - Defining Targets, 140
 - solve for Instruments, 141
- Test if mnemonic is endogenous, 93
- tracepre()
 - method trace influnce of causal variables, 207
- Tutorials
 - Anaconda, 257
 - Jupyter Notebook, 257
 - Mathplotlib, 257
 - Pandas, 257
 - Python, 257
- upd(), 51
 - + operator, 55
 - +GROWTH operator, 57
 - = operator, 55
 - =GROWTH operator, 57
 - =diff operator, 58
 - % operator, 56
 - Create new variable, 52
 - Example the Keep_Growth option, 117
 - Examples, 52
 - keep_growth option, 59
 - lprint option, 67
 - Options, 59
 - scale option, 65
 - Time scope of command, 54
 - Update several variables simultaneously, 59
- Update ModelFlow, 18
- Updating ModelFlow
 - Anaconda/MiniConda, 18
- Variable dependencies, 204, 207
- variable names
 - Wildcard, 86
- Variable selection, 289
 - ! Variable descriptions, 88, 90, 292
 - # Variable groups, 90, 291
 - #ENDO all endogenous variables, 292
 - Deferred substitution, 289
 - Use of {cty}, 289
 - Variable names with wildcards, 289
 - Wildcards, 85
- Wildcard
 - variable names, 86
- Wildcard searches! The ! operator: search on variable descriptions, 288
- Wildcard selection of data, 90
- with .set_smpl()
 - Restrict the time period, 91
- with clause
 - .set_smpl - local time scope, 115
 - .set_smpl(begin,end) method to temporarily alter the active rows in the model object, 129
 - data display format, 115

