MFMod models in Python with ModelFlow

Andrew Burns and Ib Hansen

Apr 02, 2023

CONTENTS

Andrew Burns and Ib hansen

Part I

Introduction

CHAPTER

ONE

INTRODUCTION

Warning: This Jupyter Book is work in progress.

This paper describes the implementation of the World Bank's MacroFiscalModel (MFMod) [BCJ+19] in the open source solution program ModelFlow (Hansen, 2023).

1.1 Background

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

MFMod is the World Bank's work-horse macro-structural economic modelling framework. It exists as a linked system of 184 country specific models that can be solved either independently or as a larger system. The MFMod system replaced the Bank's RMSIM-X model {cite}p:`addison_world_2019` **not in the bib file** and evolved from earlier macro-structural models developed by the Bank during the 2000s to strengthen the basis for the forecasts produced by the World Bank.

Some examples of these models were released on the World Bank's isimulate platform early in 2010 along with several CGE models dating from this period.

Beginning in 2015, the core model that was developed for the isimulate platform was developed and extended substantially into the main MFMod (MacroFiscalModel) model that is used today for the World Bank's twice annual forecasting exercise The Macro Poverty Outlook. This model continues to evolve and be used as the workhorse forecasting and policy simulation model of the World Bank.

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` in 2007 made several versions of this earlier model available for simulation on the isimulate platform, and these models continue to be available there. The isimulate platform housed (and continues to house) public access to earlier versions of the MFMod system, and allows simulation of these and other models – but does not give researchers access to the code or the ability to construct complex simulations.

[1] Economic modelling has a long tradition at the World Bank. The Bank has had a long-standing involvement in CGE modeling is the World Bank [DJ13], indeed the popular mathematics package GAMS, which is widely used to solve CGE and Linear Programming models, started out as a project begun at the World Bank in the 1970s.

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

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

1.2.1 Climate change and the MFMod system

Most recently, the Bank has extended the standard MFMod framework to incorporate the main features of climate change ([BJS21a])– both in terms of the impact of the economy on climate (principally through green-house gas emissions, like $CO_2, N_2O, CH_4, ...$) and the impact of the changing climate on the economy (higher temperatures, changes in rainfall quantity and variability, increased incidence of extreme weather) and their impacts on the economy (agricultural output, labor productivity, physical damages due to extreme weather events, sea-level rises etc.).

These climate enhanced versions of MFMod serve as one of the two main modelling systems (along with the Bank's MANAGE CGE system) in the World Bank's Country Climate Development Reports

1.3 Moving the framework to an open-source footing

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

This paper reports on the results of these efforts. In particular, it provides step by step instructions on how to install the `modelflow` framework, import a World Bank macrostructural model, perform simulations with that model and report results using the many analytical and reporting tools that have been built into `modelflow`. It is not a manual for `modelflow`, such a manual can be found here nor is it documentation for the MFMod system, such documentation can be found here Burns *et al.* and here [BJS21b], [BJS21a]). Nor is it documentation for the specific models described and worked with below.

Part II

Macrostructural Models

CHAPTER

TWO

MACROSTRUCTURAL MODELS

The economics profession uses a wide range of models for different purposes. Macro-structural models (also known as semi-structural or Macro-econometric models) are a class of models that seek to summarize the most important interconnections and determinants of an economy. Computable General Equilibrium (CGE), and Dynamic Stochastic General Equilibrium (DSGE) models are other classes of models that also seek, using somewhat different methodologies, to capture the main economic channels by which the actions of agents (firms, households, governments) interact and help determine the structure, level and rate of growth of economic activity in an economy. Olivier Blanchard, former Chief Economist at the International Monetary Fund, in a series of articles published between 2016 and 2018 that were summarized in [Bla18]. In these articles he lays out his views on the relative strengths and weaknesses of each of these systems, concluding that each has a role to play in helping economists analyze the macro-economy.

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

2.1 A system of equations

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

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

In a fully general form it can be written as:

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

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

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

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

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

2.2 Behavioural equations

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

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

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

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

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

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

and dividing both sides by Y_t^{disp} gives:

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

or in logarithms

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

we can then write our ECM equation as

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

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

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

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

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

CHAPTER

THREE

MODELFLOW AND THE MFMOD MODELS OF THE WORLD BANK

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

3.1 A brief history of ModelFlow

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

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

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

Part III

Installation of modelflow

CHAPTER

FOUR

INSTALLATION OF MODELFLOW

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

4.1 Installation of Python

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

Python comes in many flavors and `modelflow` will work with any of them. However, it is strongly suggested that you use the Anaconda version of Python. The remainder of this section points to instructions on how to install the Anaconda version of python (under Windows, MacOS and under Linux). Modelflow works equally well under all three.

This is followed by section that describes the steps necessary to create an anaconda environment with all the necessary packages to run `modelflow`.

4.1.1 Installation of Anaconda under Windows

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

It is strongly advised that Anaconda be installed for a single user (Just Me) This is much easier to maintain over time. Installing "For all users on this computer" will substabitally increase the complexity of maintaining python on your computer.

[1] Wikipedia article on python

4.1.2 Installation of Python under macOS

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

4.1.3 Installation of Python under Linux

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

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

CHAPTER

FIVE

INSTALLATION OF MODELFLOW

Warning: The following instructions concern the installation of `modelflow` within an Anaconda installation of python. Different flavors of Python may require slight changes to this recipe, but are not covered here.

`Modelflow` is built and tested using the anaconda python environment. It is strongly recommended to use Anaconda with ```modelflow```.

If you have not already installed Anaconda following the instructions in the preceding chapter, please do so **Now**.

`Modelflow` is a python package that defines the modelflow class `model` among others. `Modelflow1` has many dependencies. Installing the class the first time can take some time depending on your internet connection and computer speed. It is essential that you follow all of the steps outlined below to ensure that your version of `modelflow` operates as expected.

5.1 Installation of `modelflow` under Anaconda

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

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

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

At the end of the process you will have a new conda environment ModelFlow, and this will have been activated.

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

Note: NB: The next time you want to work with modelflow, you will need to activate the `modelflow` environment by

1) Opening the Anaconda command prompt window

2) Activate the ModelFlow environment we just created by executing the folling command

```
conda activate modelflow
```

CHAPTER

SIX

TESTING YOUR INSTALLATION OF MODELFLOW

To test that the installation of modelflow has worked properly, we will build a model using the modelflow framework and then simulate it. A simple model that illustrates many of the functions of modelflow is the Solow growth model.

The code below first sets up the python environment by importing the modelflow and pandas classes. The initial two lines of code and the final two lines just set up the environment for optimal display and are not required.

To test the installation on your system you can copy this code into a Jupyter notebook and execute it.

6.1 Specifying the model

Having loaded the model class from the modelflow library, we can start constructing the model.

The first step is to define the equations of the model, using `modelflow`'s Business Logic Language.

Business Logic Language

More on how to specify models here

The below code segment defines a string fsolow that contains the equations for the solow model, where:

- GDP is defined as a simple Cobb-Douglas production function as the product of TFP, Capital (raised to the share of capital in total income) and Labour (raised to the share of labor in total income)
- Investment is equal to GDP less consumption
- The change in capital is equal to investment this period less the depreciation of the capital stock from the previous period
- Labor grows at the rate of growth of the variable `Labor_growth`
- a pure reporting identity Capital_intensity the ratio of the Capital Stock to the Labor input

We thus have a system of 6 equations with 6 unknowns (GDP, Consumption, Investment, Change in the Capital stock, and change in Labor supply, and the capital_intensity) and exogenous variables (TFP, alfa,savings_rate,Depreciation_rate and Labor_growth).

The equations for Labor and Capital have been entered as difference equations. The `modelflow` object will automatically normalize them, generating an internal representation of `Labour=Labour(t-1)*(1+Labor_growth)` and `Capital=Capital(t-1)*(1-Depreciation_rate)+Investment`

```
fsolow = '''\
GDP          = TFP  * Capital**alfa * Labor **(1-alfa)
Consumption     = (1-saving_rate)  * GDP
Investment      = GDP - Consumption
diff(Capital)   = Investment-Depreciation_rate * Capital(-1)
diff(Labor)     = Labor_growth * Labor(-1)
Capital_intensity = Capital/Labor
'''
```

To create the model we instantiate (create) a variable `msolow` (which will ultimately contain both the equations and data for the model) using the `.from_eq()` method of the `modelflow` class – submitting to it the equations in string form, and giving it the name "Solow model".

```
msolow = model.from_eq(fsolow,modelname='Solow model')
```

The internal representation of the normalized equations can be displayed in normalized business language with the `modelflow` method `.print_model`:

```
msolow.print_model
```

```
FRML <> GDP          = TFP  * CAPITAL**ALFA * LABOR **(1-ALFA)  $
FRML <> CONSUMPTION     = (1-SAVING_RATE)  * GDP  $
FRML <> INVESTMENT      = GDP - CONSUMPTION    $
FRML <> CAPITAL=CAPITAL(-1)+(INVESTMENT-DEPRECIATION_RATE * CAPITAL(-1))$
FRML <> LABOR=LABOR(-1)+(LABOR_GROWTH * LABOR(-1))$
FRML <> CAPITAL_INTENSITY = CAPITAL/LABOR  $
```

6.2 Create some data

For the moment `msolow` has a mathematical representation of a system of equations but no data.

To add data we create a pandas dataframe with initial values for our exogenous variables. Technically capital and labor are endogenous in the Solow model, but because they are specified as change equations their initial values are exogenous and need to be initialized.

The code below instantiates (creates) a panda dataframe `df` and fills it with the variables for our model, initializing these with a series of values over 300 datapoints. The final command displays the first ten rows of the dataframe.

Note:

```
Pandas data frames is a foundational class of python.  There are thousands of web␣
↪sites dedicated to understanding pandas.  Some notable ones include:
```

```
N = 300
df = pd.DataFrame({'LABOR':[100]*N,
                   'CAPITAL':[100]*N,
                   'ALFA':[0.5]*N,
                   'TFP': [1]*N,
                   'DEPRECIATION_RATE': [0.05]*N,
                   'LABOR_GROWTH': [0.01]*N,
```

(continues on next page)

(continued from previous page)

```
                    'SAVING_RATE':[0.05]*N},index=[v for v in range(2000,2300)])
df.head() #this prints out the first 5 rows of the dataframe
```

	LABOR	CAPITAL	ALFA	TFP	DEPRECIATION_RATE	LABOR_GROWTH	SAVING_RATE
2000	100	100	0.5	1	0.05	0.01	0.05
2001	100	100	0.5	1	0.05	0.01	0.05
2002	100	100	0.5	1	0.05	0.01	0.05
2003	100	100	0.5	1	0.05	0.01	0.05
2004	100	100	0.5	1	0.05	0.01	0.05

6.3 Putting it together

Having defined an initial data set for all the exogenous variables, we can combine these with the equations and solve the model.

The command below solves the model `msolow` on the data contained in the dataframe `df` and stores the output in a new dataframe called `result`.

The last line displays the values of the simulated model, which now includes results for the endogenous variables, and different values for the Labor and Capital variables reflecting their endogeneity for periods 2 through 300.

```
result = msolow(df,keep='Baseline')
# The model is simulated for all years possible

result.head(10)
```

	LABOR	CAPITAL	ALFA	TFP	DEPRECIATION_RATE	LABOR_GROWTH	\
2000	100.000000	100.000000	0.5	1.0	0.05	0.01	
2001	101.000000	100.025580	0.5	1.0	0.05	0.01	
2002	102.010000	100.076226	0.5	1.0	0.05	0.01	
2003	103.030100	100.151443	0.5	1.0	0.05	0.01	
2004	104.060401	100.250762	0.5	1.0	0.05	0.01	
2005	105.101005	100.373733	0.5	1.0	0.05	0.01	
2006	106.152015	100.519926	0.5	1.0	0.05	0.01	
2007	107.213535	100.688931	0.5	1.0	0.05	0.01	
2008	108.285671	100.880357	0.5	1.0	0.05	0.01	
2009	109.368527	101.093830	0.5	1.0	0.05	0.01	

	SAVING_RATE	CONSUMPTION	GDP	INVESTMENT	CAPITAL_INTENSITY
2000	0.05	0.000000	0.000000	0.000000	0.000000
2001	0.05	95.486029	100.511609	5.025580	0.990352
2002	0.05	95.986562	101.038487	5.051924	0.981043
2003	0.05	96.501546	101.580575	5.079029	0.972060
2004	0.05	97.030930	102.137821	5.106891	0.963390
2005	0.05	97.574667	102.710176	5.135509	0.955022
2006	0.05	98.132713	103.297593	5.164880	0.946943
2007	0.05	98.705029	103.900030	5.195002	0.939144
2008	0.05	99.291576	104.517449	5.225872	0.931613
2009	0.05	99.892323	105.149813	5.257491	0.924341

6.4 Create a scenario and run again

dataframe.upd

When importing modelclass all pandas dataframes are enriched with a a handy way to create a new pandas dataframe as a copy of an existing one but with one or more series updated.

In this case df.upd will create a a new dataframe `dfscenaario` with updated LABOR_GROWTH

For more detail on the `.upd` method look here here

```
dfscenario = df.mfcalc('<2023 2200> LABOR_GROWTH = LABOR_GROWTH + 0.002')  # create a␣
↪new dataframe, increase LABOR_GROWTH by 0.002
scenario   = msolow(dfscenario,keep='Higher labor growth ') # simulate the model
```

6.5 Inspect results

`Modelflow` includes a range of methods to view data and results, either as graphs or as tables. Some of these are part of standard python, others are additional features that `modelflow` makes available.

Scenario results can be inspected either by referring to the scenario name given in the (optional) `keep` statement when the model was solved, by referring to the `basedf` and the `lastdf`.

- `basedf` is a dataframe that is automatically generated when the model is solved and contains a copy of the initial conditions of the model prior to the shock.
- `lastdf`is a dataframe that is automatically generated when the model is solved and contains a copy of the results from the simulation. Several built in display functions use these functions to display results.

Finally one could also look at the dataframe to which the results of the simulation were assigned `scenario` in the example above.

Below is a small sub-set of the visualization options available.

6.5.1 Graphical representations of results

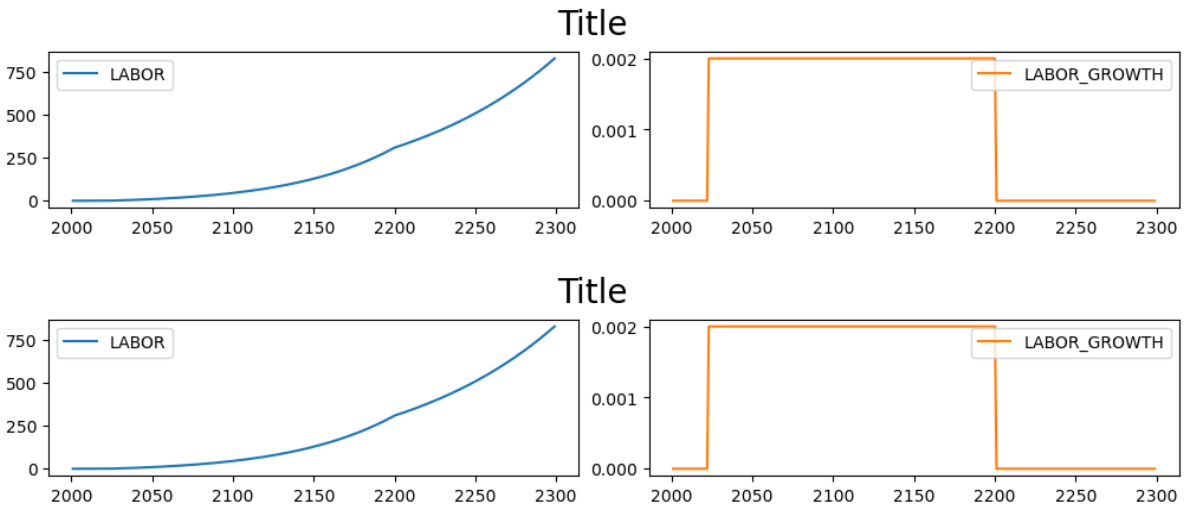
The .dif.plot() method

The `.dif.plot` method will plot the change in the level of requested variables. Requested variables can be selected either directly by name or using wildcards.

In this example, a wild card specification is used, requesting the display of all variables that begin with the text 'labor'. Note that the selector is not case sensitive.

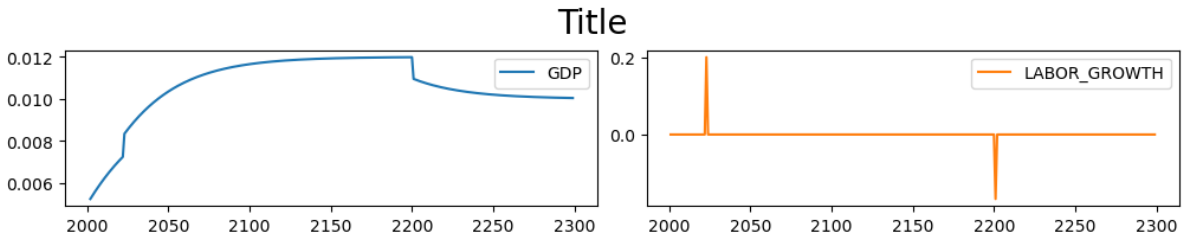
In this case we are displaying changes into the labor and labor growth variables due to the shock when we increased the growth rate of labor by .0002

```
msolow['labor*'].dif.plot()
```



In this example, instead of using a wild card selector we requested a variable explicitly by name.

```
msolow['GDP LABOR_GROWTH'].pct.plot()
```



Using the kept solutions

Because the keyword `keep` was used when running the simulations, we can refer to the scenarios by their names – or produce graphs from multiple scenarios – not just the first and last.

```
msolow.keep_plot('GDP')
```

```
{'GDP': <Figure size 1000x600 with 1 Axes>}
```

6.5.2 Textual and tabular display of results

Standard pandas syntax can be used to display data in the results dataframes.

Here we use the standard pandas `.loc` method to display every 10th data point for consumption from the results dataframe, beginning from observation 50 through 100.

```
msolow.lastdf.loc[50:100:10,'CONSUMPTION']
```

```
Series([], Name: CONSUMPTION, dtype: float64)
```

The `.dif.df` method

The `.dif.df` method prints out the changes in variables, i.e. eh difference between the level of specified variables in the `lastdf` dataframe vs the `basedf` dataframe.

```
msolow['GDP CONSUMPTION'].dif.df
```

```
              GDP  CONSUMPTION
2001     0.000000     0.000000
2002     0.000000     0.000000
2003     0.000000     0.000000
2004     0.000000     0.000000
2005     0.000000     0.000000
...           ...          ...
2295   665.334581   632.067852
2296   672.097592   638.492713
2297   678.925939   644.979642
2298   685.820324   651.529308
2299   692.781453   658.142380

[299 rows x 2 columns]
```

The `.difpct.df` method

The `.dif.pct.df` method express the changes between the last simulation and base simulation results as a percent differences in the level ($\frac{\Delta X_t}{X_{t-1}^{basedf}}$). In the example below the mul100 method multiplies the result by 100.

```
msolow['GDP CONSUMPTION'].difpct.mul100.df
```

```
           GDP  CONSUMPTION
2001       NaN          NaN
2002  0.000000     0.000000
2003  0.000000     0.000000
2004  0.000000     0.000000
2005  0.000000     0.000000
...        ...          ...
2295  0.005047     0.005047
2296  0.004892     0.004892
2297  0.004742     0.004742
2298  0.004596     0.004596
2299  0.004456     0.004456

[299 rows x 2 columns]
```

6.5.3 Interactive display of impacts

When working within Jupyter notebook the [ ] command will produce (without the .df termination) will generate a widget with the results expressed as level differences, percent differences, differences in the growth rate – both graphically and in table form.

Please consult here for a fuller presentation of the display routines built into `modelflfow`.

```
msolow['GDP CONSUMPTION']
```

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

Part IV

Some Jupyter, Python and pandas essentials

CHAPTER

SEVEN

INTRODUCTION TO JUPYTER NOTEBOOK

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

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

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

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

7.1 The idea of the notebook

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

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

The motivation for these notebooks and Jupyter notebook is to encourage practices that will ensure that if followed exactly by others, that they will be able to generate the same results.

7.2 Jupter Notebook cells

A Jupyter Notebook does all of that (and perhaps a bit more). It is divided into 'cells'.

Jupyter Notebook cells can contain:

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

- Results (in the form of tables or graphs) from the execution of computer code specified in a code cell

Every cell has two modes:

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

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

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

7.3 Execution of cells

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

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

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

7.3.1 Execution of code cells

Below is a code with some standard python that declares a variable "x", assigns it the value 10, declares a second variable "y" and assigns it the value 45. The final line of y alone, instructs python to display the value of the variable y. The results of the operation appear in Jupyter Notebook as an output cell Out[#].

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

```
45
```

the semi-colon ";" supresses output in Jupyter Notebook

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

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

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

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

```
10
```

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

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

```
90
```

7.3.2 Auto-complete and context-sensitive help

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

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

7.4 The markdown scripting language in Jupyter Notebook

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

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

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

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

7.4.1 Common markdown commands

Some of the most common of these include:

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

7.4.2 Tables in markdown

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

Below is the markdown code that generated the above table:

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

7.4.3 Displaying code

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

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

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

7.4.4 Rendering mathematics in markdown

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

Inline enclose the LaTeX code in $:

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

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

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

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

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

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

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

7.4.5 links to more info on markdown

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

7.5 How to add, delete and move cells

Jupyter Notebook cells can be added, deleted and moved.

Using the Toolbar

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

Using keyboard short cuts

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

7.6 Change the type of a cell

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

Using the Toolbar

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

Using keyboard short cuts

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

CHAPTER

EIGHT

SOME PYTHON BASICS

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

8.1 Python packages, libraries and classes

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

A **python class** is a code template that defines a python object. Classes can have member variables (data) associated with them and methods (behaviours or functions)associated with them. In python a class is created by the keyword class. An object of type class is created (instantiated) using the classes "constructor".

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

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

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

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

8.2 Importing packages, libraries, modules and classes

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

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

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

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

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

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

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

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

```
3.141592653589793
```

8.2.1 Import specific elements or classes from a module or library

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

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

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

```
from math import pi,cos,sin

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

```
3.141592653589793
-0.9899924966004454
```

8.2.2 import a class but give it an alias

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

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

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

```
3.141592653589793
-0.9899924966004454
```

8.2.3 Standard aliases

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

For example:

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

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

8.3 Introduction to Pandas dataframes

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

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

- Pandas homepage
- Pandas community tutorials

8.4 Import the pandas library

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

```
import pandas as pd
```

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

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

8.5 Pandas series

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

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

8.5.1 Series declared from a list

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

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

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

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

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

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

# Here the constructor is passed a numeric list
Simplest=pd.Series([2,3,4,5,-15])
Simplest
```

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

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

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

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

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

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

8.5.2 Series declared using a specific index

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

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

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

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

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

Now the Series look more like time series data!

8.5.3 Create Series from a dictionary

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

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

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

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

---

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

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

```
mydict2={1966:2,1967:3,1968:4,1969:5,2000:-15}
simple2=pd.Series(mydict2)
simple2
```

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

8.6 Properties and methods of dataframes in modelflow

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

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

8.6.1 dataframes

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

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

8.6.2 Creating or instantiating a dataframe

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

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

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

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

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

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

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

8.6.3 Adding a column to a dataframe

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

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

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

8.6.4 Revising values

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

Warning: The dimensions of the list assigned via the = method must be the same as the dataframe (i.e. you must provide exactly as many values as there are rows). Alternatively if you provide just one, then that value will replace all of the values in the specified column.

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

df
```

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

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

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

8.7 Column names in Modelflow

Modelflow variable names

Modelflow places more restrictions on columnnames than do pandas per se.

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

Specifically, in modelflow a variable name must:

- start with a letter
- be upper case

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

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

8.8 .index and time dimensions in Modelflow

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

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

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

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

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

For example:

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

8.8.1 Leads and lags

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

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

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

8.8.2 .columns lists the column names of a dataframe

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

```
df.columns
```

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

8.8.3 .size indicates the dimension of a list

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

```
df.columns.size
```

```
4
```

The dataframe df has 4 columns.

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

.eval is a native dataframe method, which allows us to do calculations on a dataframe. With this method expressions can be evaluated and new columns created.

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

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

```
df
```

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

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

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

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

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

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

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

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

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

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

.loc[row,column] A single element

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

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

```
2
```

.loc[:,column] A single column

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

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

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

.loc[row,:] A single row

Here all of the columns, for the selected row.

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

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

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

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

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

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

.loc using the colon to select a range

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

Here from 2018 to 2019.

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

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

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

This can be very handy when updating scenarios.

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

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

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

For more on broadcasting see here

For more info on the .loc[] method

- Description
- Search

For more info on pandas:

- Pandas homepage
- Pandas community tutorials

CHAPTER

NINE

MODELFLOWS EXTENSIONS TO PANDAS

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

As we have seen above we can modify data in a dataframe directly with built-in pandas functionalities like `.loc[]`.

9.1 .upd() method of modelflow

The `.upd()` method extends these capabilities in important ways. It gives the user a concise and expressive way to modify data in a dataframe in the way that a user or database-manager of the World Bank MFMod models might.

Notably it allows us to employ formula's to do updates, including lags and leads on variables.

`.upd()` 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 input for each time
- Preserve pre-shock growth rates for out of sample time-periods
- Display results

9.1.1 `.upd()` method operators

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

Types of update:

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

Danger: Note: the syntax of an update command requires that there be a space between variable names and the operators.

Thus `df.upd("A = 7")` is fine, but `df.upd("A =7")` will generate an error.

Similarly `df.upd("A * 1.1")` is fine, but `df.upd("A=* 1.1")` will generate an error.

9.2 `.upd()` some examples

9.2.1 Setting up the python environment

In order to use `.upd()` we need to first import all of the necessary libraries to out python session.

```
%load_ext autoreload
%autoreload 2

# First we must import pandas as modelflow into ouir workspace
# There is no problem importing multiple times, though it is not very efficient.
import pandas as pd

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

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

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

```
# Create a dataframe using standard pandas

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

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

If we want to be a bit more creative we could use a loop to create the dataframe for dates that we pass as an argument. For example, below we create a dataframe df with two Series (A and B), which we initialize with the values 100 for all data points.

In this case we we define the index dynamically as the result of a loop `index=[2020+v for v in range(number_of_rows)]`.

This bit of code runs a loop that runs for number_of_rows times setting v equal to 2020+0, 2020+1,…,202+5. The resulting list whose values are assigned to index is [2020,2021,2022,2023,2024,2025].

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

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

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

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

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

We know from above that with pandas we can add a column (series) to a dataframe simply by assigning a adding to a dataframe. For example:

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

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

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

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

Note: note that the new variable name is in lower case here. We know that lowercase letters are not legal `modelflow` variable names. But because the .upd() method knows this also, it automatically translates these into upper case so that the statement works.

9.2.3 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.

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

Danger: Note that the third line inherits the time period of the previous line.

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

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

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



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

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

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

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

 - To indicate the start of the dataframe use -0
 - To indicate the end of the dataframe use -1

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

Setting specific datapoints to specific values

In this example, upd uses the equals operator. This indicates that the variable a should be set equal to the indicated values following the = operator (42 44 45 46 in this example). The dates enclosed in <> indicate the period over which the change should be applied.

Either:

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

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

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

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

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

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

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

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

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

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

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

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

In this example:

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

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

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

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

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

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

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

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

```
res =df.upd('''
# Same number of values as years
<2021 2025> A =GROWTH  1
# now we add values  to the growth rate,
a +growth   2 3 4 5 6
''')
print(f'Dataframe:\n{res}\n\nGrowth:\n{res.pct_change()*100}\n')
```

```
Dataframe:
                A    B
2020  100.000000  100
2021  103.000000  100
2022  107.120000  100
2023  112.476000  100
2024  119.224560  100
2025  127.570279  100

Growth:
        A    B
2020  NaN  NaN
2021  3.0  0.0
2022  4.0  0.0
2023  5.0  0.0
2024  6.0  0.0
2025  7.0  0.0
```

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

```
df.upd('''
# Same number of values as years
< 2021 2022> A =diff  2 4
# cv number to all years in between start and end

<2020 > same = 100
<2021 2025> same =diff  2
''')
```

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

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

```
df
```

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

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

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

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

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

The default keep_growth behaviour

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

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

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

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

Note: At the line level:

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

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

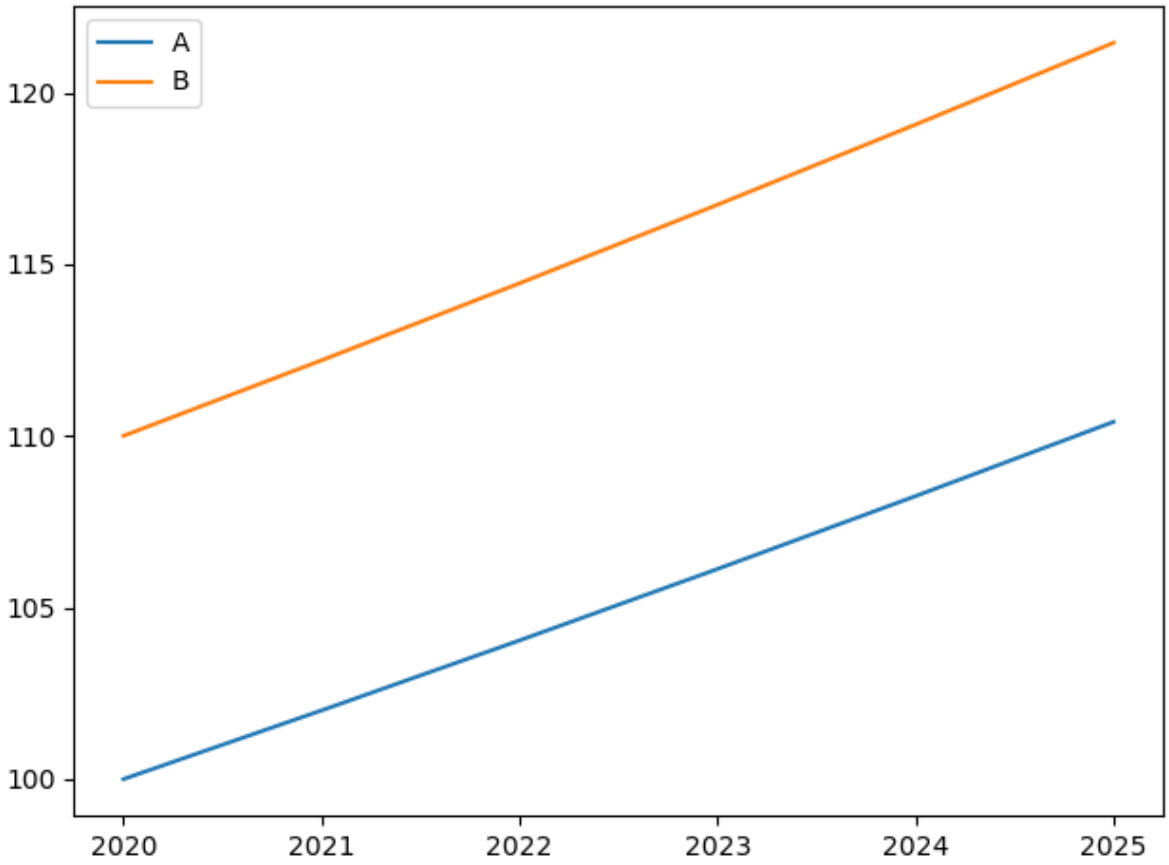
```
df = pd.DataFrame(100,
       index=[2020+v for v in range(number_of_rows)], # create row index
       # equivalent to index=[2020,2021,2022,2023,2024,2025]
       columns=['A','B'])

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

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

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

```
<Axes: >
```

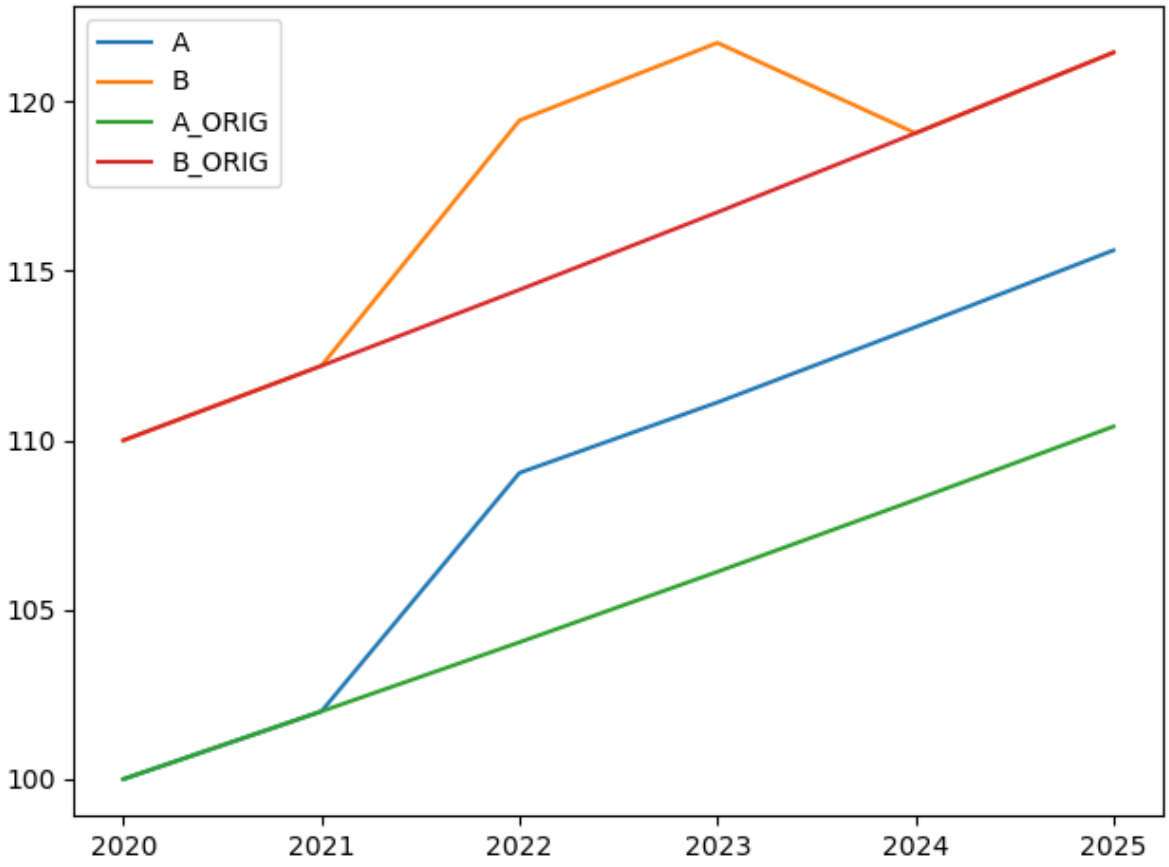


Now lets modify each by adding 5 to the level in 2022 and 2023. For B we will do setting the keep_growth option as False and for 'B' keep_growth positive. While the keep_growth is a global variable it can be set at the line level also using the –kg option (`keep_growth=True`) and –nkg option ((`keep_growth=False`).

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

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

```
<Axes: >
```



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

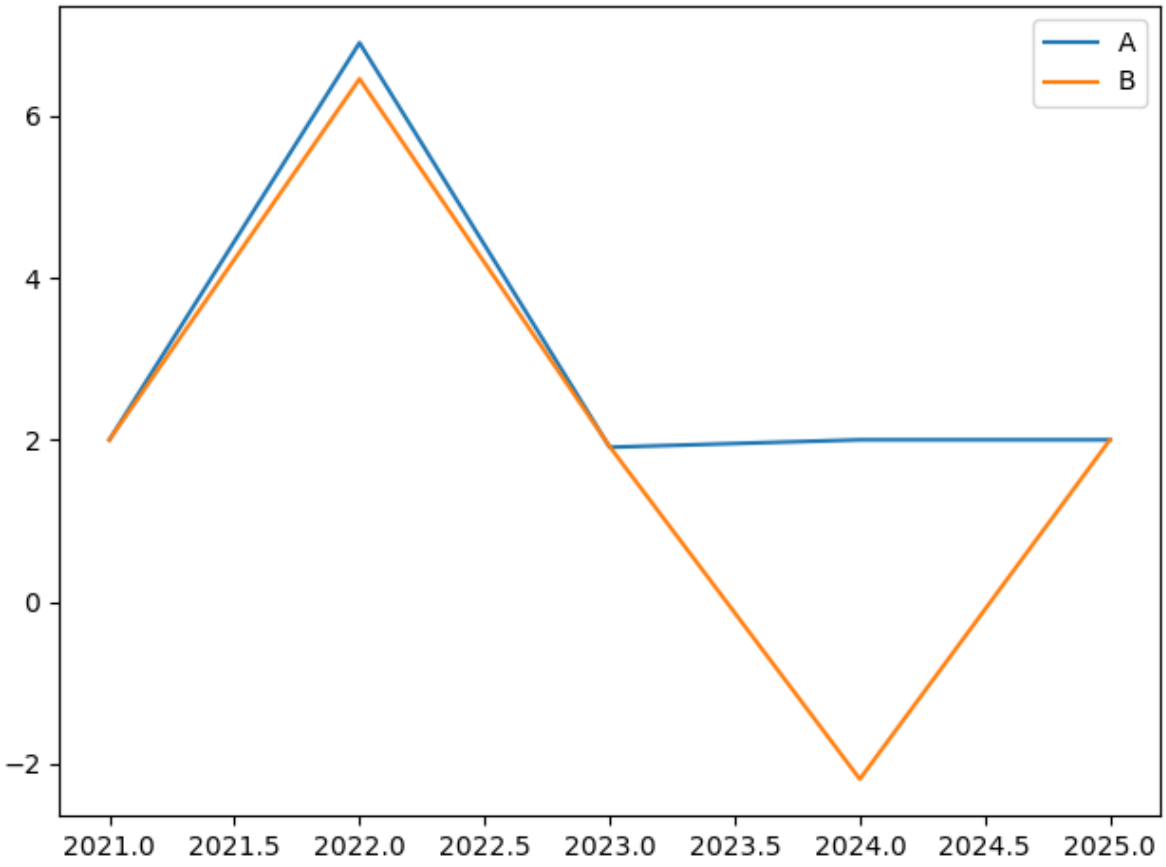
For the 'B' variable the same level change was input but because of the `--nkg` (equivalent to `keep_growth=False`) the periods after the change were affected retained their old values.

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

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

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

```
<Axes: >
```



Note: Python constructs

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

Uses

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

9.3 .upd(„,keep_growth) some more examples

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

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

(continues on next page)

(continued from previous page)

```
original = dftest.upd('<2021 2025> a =growth 1 2 3 4 5')
print(f'Levels:\n{original}\n\nGrowth:\n{original.pct_change()*100}\n')
```

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

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

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

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

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

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

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


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

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

(continues on next page)

(continued from previous page)

```
growth kg:
         A
2020   NaN
2021  20.0
2022   0.0
2023   0.0
2024   4.0
2025   5.0


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

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

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

.upd() with the option keep_growth set globally

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

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

We can also change the keep_growth variable for all the lines by setting the option in the command line.

In the below example we set the global option `keep_growth=True`.

Now as default, all lines will keep the growth rate

- 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,
```

(continues on next page)

(continued from previous page)

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

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

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

```
Dataframe:
        A    B      C      D    E
2020  100  100  100.0  100.0  100
2021  100  100  100.0  100.0  100
2022  100  100  200.0  300.0  400
2023  100  100  200.0  300.0  400
2024  100  100  200.0  300.0  100
2025  100  100  200.0  300.0  100

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

9.3.3 Some more advanced example

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

First create a string with update lines

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

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

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

Note: ***Python constructs***

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

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

Use the update lines to update a dataframe

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

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

```
                A           B      A_ORIG      B_ORIG           C           D  \
2020  100.000000  110.000000  100.000000  110.000000  100.000000  100.000000
2021  102.000000  112.200000  102.000000  112.200000  101.000000  101.000000
2022  109.040000  119.444000  104.040000  114.444000  103.020000  103.020000
2023  111.120800  121.732880  106.120800  116.732880  106.110600  106.110600
2024  113.343216  119.067538  108.243216  119.067538  110.355024  110.355024
2025  115.610080  121.448888  110.408080  121.448888  115.872775  115.872775
```

(continues on next page)

(continued from previous page)

```
                E           F
2020  100.000000  100.000000
2021  101.000000  101.000000
2022  103.020000  103.020000
2023  106.110600  106.110600
2024  110.355024  110.355024
2025  115.872775  115.872775
```

9.3.4 –kg can replace –keep_growth and –nkg can replace –non_keep_growth

Just to make typing more easy

9.4 Update several variable in one line

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

```
df.upd('''
<2022 2024> h i j k =      40
<2020>      p q r s =       1000
<2021 -1>   p q r s =growth 2     # -1 indicates the last year
''')
```

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

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

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

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

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

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

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

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

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

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


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

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


severity=0.5
Dataframe:
                A           B      A_ORIG      B_ORIG
2020  100.000000  110.000000  100.000000  110.000000
2021  100.500000  117.200000  102.000000  112.200000
```

(continues on next page)

(continued from previous page)

```
2022  101.505000  124.444000  104.040000  114.444000
2023  103.027575  126.732880  106.120800  116.732880
2024  105.088126  124.067538  108.243216  119.067538
2025  107.715330  126.448888  110.408080  121.448888

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


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

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

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

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

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

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

```
               A           B      A_ORIG      B_ORIG
2020  100.000000  110.000000  100.000000  110.000000
```

(continues on next page)

(continued from previous page)

```
2021  4284.000000  112.200000  102.000000  112.200000
2022  4797.760000  119.444000  104.040000  114.444000
2023   111.120800  121.732880  106.120800  116.732880
2024   113.343216  119.067538  108.243216  119.067538
2025   115.610080  121.448888  110.408080  121.448888
```

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

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

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

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

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

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

9.8 The call

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

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

    Returns:
        df (TYPE): the updated dataframe .

    A line in updates looks like this:

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

CHAPTER

TEN

.MFCALC() AN EXTENSION OF STANDARD PANDAS

10.1 .mfcalc usage

The `.mfcalc()` method extends dataframe and the method `.upd()`. It can be particularly useful when creating scenarios.

But it can also be used to perform quick and dirty calculations or even to see how modelflow would normalize an equation.

10.2 workspace initialization

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

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

10.3 Create a simple dataframe

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

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

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

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

```
          A
2020  100.0
2021  100.0
```

(continues on next page)

(continued from previous page)

```
2022  100.0
2023  100.0
2024  100.0
2025  100.0
```

10.4 `.mfcalc()` in action

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

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

The below call creates a new column x.

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

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

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

NOTE:

By default `.mfcalc` will initialize a new variable with zeroes. Moreover, if a formula passed to `.mfcalc` contains a lag a value will be calculated for the first row only if there is data in the series for the preceding row.

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

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

```
df
```

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

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

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

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

10.4.2 Recalculate A so it grows by 2 percent

mfcalcs knows that it can not start to calculate in 2020 as there is no lagged variable. So it will start calculating in 2021 and leave the pre-existing value unchanged.

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

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

```
                A
2020  100.000000
2021  102.000000
2022  104.040000
2023  106.120800
2024  108.243216
2025  110.408080
```

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

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

10.4.3 mfcalc(), the showeq option

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

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

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

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

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

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

$$dlog(a) = 0.02$$

$$log(a) - log(a_{t-1}) = .02$$

$$log(a) = log(a_{t-1}) + .02$$

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

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

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

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

10.4.4 Using .diff (△) with mfcalc

```
res = df.mfcalc('diff(a) =  2') # Set delta to 2
res.diff()                      # Display the delta
```

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

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

10.4.5 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 would be expected if you ran the two commands sequentially.

For example:

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

res

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

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

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

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

As a result `modelflow` generates a result only for the periodf 2021 through 2025 and it is this result that is passed to the second equation, which adds 42 to this number. Thus X in 2020 is not 142 as one might have expected but zero, the value to which the newly created variable defaults.

Compare the results above with the results (below) when the two steps are not undertaken in the same mfcalc command.

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

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

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

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

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

10.4.6 Setting a time frame with mfcalc.

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

Below, as in the example above we have zeroes for x prior to 2023 when the expressions are executed.

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

(continues on next page)

(continued from previous page)

```
res.diff()

res
```

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

Part V

Features

CHAPTER

ELEVEN

USEFUL MODEL INSTANCE PROPERTIES AND METHODS

The focus of this chapter is to introduce some properties and methods of the model instance.

First a model and data is loaded, then a scenario is run. Then we have some content to use.

A model instance gives the user access to a number of properties and methods which helps in managing the model and its results.

If `mmodel` is a model instance `mmodel.<property>` will return a property. Some properties can also be assigned by the user just by:

    mmodel.property = something

The model class itself also have a few properties. These are simple accessed by `model.<property>`.

Enjoy

11.1 Import the model class

This class incorporates most of the methods used to manage a model.

Assuming the ModelFlow library has been installed on your machine, the following imports set up your notebook so that you can run the cells in this notebook.

In order to manipulate plots later on matplotlib.pyplot is also imported.

```
#%matplotlib notebook
%matplotlib inline
```

```
from modelclass import model
```

```
import matplotlib.pyplot as plt # To manipulate plots
```

11.2 Class methods to help in Jupyter Notebook

11.2.1 .widescreen() use Jupyter Notebook in widescreen

Enables the whole viewing area of the browser.

```
model.widescreen()
```

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

11.2.2 .scroll_off() Turn off scroll cells in Jupyter Notebook

Can be useful

```
model.scroll_off()
```

11.3 .modelload Load a pre-cooked model, data and descriptions

In this notebook, we will be using a pre-existing model of Pakistan.

The file 'pak.pcim' has been created from a Eviews workspace. It contains all that is needed to run the model:

- Model equations
- Data
- Simulation options
- Variable descriptions

Using the 'modelload' method of the 'model' class, a model instance 'mpak' and a 'result' DataFrame is created.

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

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

mpak The *modelload* method processes the file and initiates the model, that we call 'mpak' (m for model and pak for Pakistan) with both equations and the data.

'mpak' is an **instance** of the model object with which we will work.

baseline 'result' is a Pandas dataframe containing the data that was loaded.

run=1 the model is simulated. The simulation timeframe and options from the time the file where dumped will be used. The two objects **mpak.basedf** and **mpak.lastdf** will contain the simulation result. If run=0 the model will not be simulated.

silent=1 if silent is set to 0 information regarding the simulation will be displayed.

keep='Baseline' This saves the result in a dictionary mpak.keep_solutions.

11.4 Create a scenario

Many objects relates to comparison of different scenarios. So first a scenario is created by updating some exogenous variables. In this case the carbon tax rates for gas, oil and coal are all set to 29 from 2023 to 2100. Then the scenario is simulated. Now the mpak object contains a number of useful properties and methods.

You can find more on this experiment here

```
scenario_exo  =  baseline.upd("<2020 2100> PAKGGREVCO2CER PAKGGREVCO2GER␣
↪PAKGGREVCO2OER = 29")
```

11.5 () Simulate on a dataframe

When calling the model instance like `mpak(dataframe,start, end)` the model will be simulated for the time frame `start to end` using the dataframe. Just above we created a dataframe `scenario_exo` where the tax variables are updated. Now the `mpak` can be simulated. We simulate from 2020 to 2100.

```
scenario = mpak(scenario_exo,2020,2100,keep=f'Coal, Oil and Gastax : 29') # runs the␣
↪simulation
```

11.6 Access results

Now we have 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.

```
print(f'mpak.basedf: Dataframe: with {mpak.basedf.shape[0]} years and {mpak.basedf.
↪shape[1]} variables')
print(f'mpak.lastdf: Dataframe: with {mpak.lastdf.shape[0]} years and {mpak.lastdf.
↪shape[1]} variables')
```

```
mpak.basedf: Dataframe: with 121 years and 1290 variables
mpak.lastdf: Dataframe: with 121 years and 1290 variables
```

11.6.1 .keep_solutions, A dictionary of dataframes with results

Create a dictionary of dataframes with .keep_solutions. 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.

In our example we have created two scenarios. A baseline and a scenario with the tax set to 29. So mpak.keep_solutions looks like this:

```
print('mpak.keep_solutions contains:')
for key,value in mpak.keep_solutions.items():
    print(f'key = {key:25}|Dataframe: {value.shape[0]} years and {value.shape[1]}␣
 ↪variables')
```

```
mpak.keep_solutions contains:
key = Baseline                 |Dataframe: 121 years and 1290 variables
key = Coal, Oil and Gastax : 29|Dataframe: 121 years and 1290 variables
```

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. Two methods can be used:

mpak.keep_solutions = {}

or in the simulation call:

mpak(,,keep=")

11.6.2 More on manipulating keep_solution:

Here

11.6.3 .oldkwargs, Options in the simulation call is persistent between calls

When simulating a model the parameters are persistent. So the user just have to provide the solution options once. These persistent parameters are located in the property .oldkwargs.

In this case the persistent parameters are:

```
mpak.oldkwargs
```

```
{'silent': 1, 'alfa': 0.7, 'ldumpvar': 0, 'keep': 'Coal, Oil and Gastax : 29'}
```

The user may have to reset the parameters, this is done like this:

To reset the options just do:

mpak.oldkwargs = {}

11.7 .current_per, The time frame operations are performed on

Most operations on a model class instance operates on the current time frame. It is a subset of the row index of the dataframe which is simulated.

In this case it is:

```
mpak.current_per
```

```
Int64Index([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.

```
scenario.index  # the index of the dataframe
```

```
Int64Index([1980, 1981, 1982, 1983, 1984, 1985, 1986, 1987, 1988, 1989,
            ...
            2091, 2092, 2093, 2094, 2095, 2096, 2097, 2098, 2099, 2100],
           dtype='int64', length=121)
```

11.7.1 .smpl, Set time frame

The time frame can be set like this:

```
mpak.smpl(2020,2025)
mpak.current_per
```

```
Int64Index([2020, 2021, 2022, 2023, 2024, 2025], dtype='int64')
```

11.7.2 .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   Int64Index([2020, 2021, 2022, 2023, 2024, 2025], dtype='int64
 ↪')
Local time frame      Int64Index([2022, 2023], dtype='int64')
Unchanged global time Int64Index([2020, 2021, 2022, 2023, 2024, 2025], dtype='int64
 ↪')
```

11.7.3 .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   Int64Index([2020, 2021, 2022, 2023, 2024, 2025], dtype='int64
 ↪')
Local time frame      Int64Index([2019, 2020, 2021, 2022, 2023, 2024, 2025], dtype=
 ↪'int64')
Unchanged global time Int64Index([2020, 2021, 2022, 2023, 2024, 2025], dtype='int64
 ↪')
```

11.8 Using the index operator [ ] to select and visualize variables.

The index operator [ ] can be used to select variables and then process the values for quick analysis.

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)

In the following example we are selecting the results of mpak['PAKNYGDPMKTPKN']

This call will return a special class (called `vis`). It implements a number of methods and properties which comes in handy for quick analyses.

Several properties and methods can be chained. An example:

```
with mpak.set_smpl(2020,2100):
    mpak['PAKNYGDPMKTPKN'].difpctlevel.mul100.rename().plot(colrow=1,
                title='Difference to baseline in percent',top=0.8);
```

But first some basic information

11.8.1 model['#ENDO']

Use '#ENDO' to access all endogenous variables in your model instance.

For the sake of space, the result is saved in the variable 'allendo' and not printed.

```
allendo = mpak['#ENDO']
# allendo.show
```

11.8.2 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.

```
mpak['PAKNYGDPMKTPKN PAKNECONPRVTKN']
```

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

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
```

11.8.3 .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    2.647002e+07    2.344055e+07
2021    2.676493e+07    2.366076e+07
2022    2.688965e+07    2.376966e+07
2023    2.708904e+07    2.395330e+07
```

11.8.4 .show as a html table with tooltips

If you want the variable descriptions use this

```
mpak['PAKNYGDPMKTPKN PAKNECONPRVTKN'].show
```

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

11.8.5 .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']
```

11.8.6 .frml The formulas

Use .frml to access all the equations for the endogenous variables.

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

```
PAKNYGDPMKTPKN : FRML <> PAKNYGDPMKTPKN =␣
↪PAKNECONPRVTKN+PAKNECONGOVTKN+PAKNEGDIFTOTKN+PAKNEGDISTKBKN+PAKNEEXPGNFSKN-
↪PAKNEIMPGNFSKN+PAKNYGDPDISCKN+PAKADAP*PAKDISPREPKN $
PAKNECONPRVTKN : FRML <Z,EXO> PAKNECONPRVTKN = (PAKNECONPRVTKN(-
↪1)*EXP(PAKNECONPRVTKN_A+ (-0.2*(LOG(PAKNECONPRVTKN(-1))-LOG(1.21203101101442)-
↪LOG((((PAKBXFSTREMTCD(-1)-PAKBMFSTREMTCD(-1))*PAKPANUSATLS(-1))+PAKGGEXPTRNSCN(-
↪1)+PAKNYYWBTOTLCN(-1)*(1-PAKGGREVDRCTXN(-1)/100))/PAKNECONPRVTXN(-1)))+0.
↪763938860758873*((LOG((((PAKBXFSTREMTCD-
↪PAKBMFSTREMTCD)*PAKPANUSATLS)+PAKGGEXPTRNSCN+PAKNYYWBTOTLCN*(1-PAKGGREVDRCTXN/
↪100))/PAKNECONPRVTXN))-(LOG((((PAKBXFSTREMTCD(-1)-PAKBMFSTREMTCD(-
↪1))*PAKPANUSATLS(-1))+PAKGGEXPTRNSCN(-1)+PAKNYYWBTOTLCN(-1)*(1-PAKGGREVDRCTXN(-
↪1)/100))/PAKNECONPRVTXN(-1))))-0.0634474791568939*DURING_2009-0.
↪3*(PAKFMLBLPOLYXN/100-((LOG(PAKNECONPRVTXN))-(LOG(PAKNECONPRVTXN(-1))))))  ))  *␣
↪(1-PAKNECONPRVTKN_D)+ PAKNECONPRVTKN_X*PAKNECONPRVTKN_D $
```

11.8.7 .rename() Rename variables to descriptions

Use .rename() to assign variable descriptions as variable names.

Handy when plotting!

```
mpak['PAKNYGDPMKTPKN PAKNECONPRVTKN'].rename().df
```

```
          Real GDP  HH. Cons Real
2020  2.647002e+07   2.344055e+07
2021  2.676493e+07   2.366076e+07
2022  2.688965e+07   2.376966e+07
2023  2.708904e+07   2.395330e+07
```

11.8.8 Transformations of solution results

When the variables has been selected through the index operator a number of standard data transformations can be performed.

Transfomation	Meaning	expression
pct	Growth rates	$\left(\frac{this_t}{this_{t-1}} - 1\right)$
dif	Difference in level	$l - b$
difpct	Differens in growth rate	$\left(\frac{l_t}{l_{t-1}} - 1\right) - \left(\frac{b_t}{b_{t-1}} - 1\right)$
difpctlevel	differens in level in pct of baseline	$\left(\frac{l_t - b_t}{b_t}\right)$
mul100	multiply by 100	$this_t \times 100$

- $this$ is the chained value. Default lastdf but if preseeded by .base the values from .basedf will be used
- b is the values from .basedf
- l is the values from .lastdf

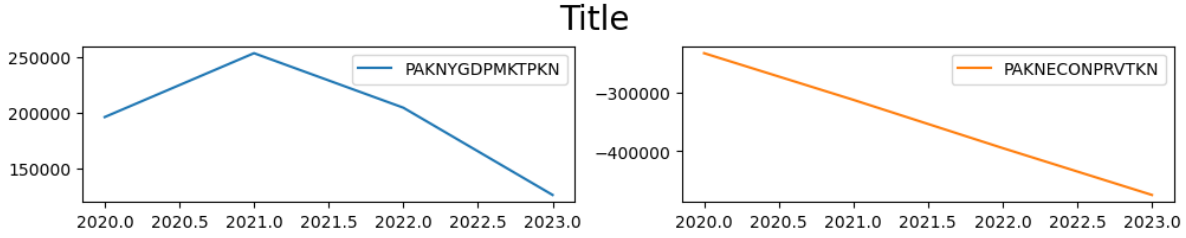
11.8.9 .dif Difference in level

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 .basedf.

```
mpak['PAKNYGDPMKTPKN PAKNECONPRVTKN'].dif.plot()
```



11.8.10 .pct Growthrates

Display growth rates

$$\left(\frac{l_t}{l_{t-1}} - 1\right)$$

```
mpak['PAKNYGDPMKTPKN PAKNECONPRVTKN'].pct.plot();
```

11.8.11 .difpct property difference in growthrate

The difference in the growth rates between the last and base dataframe.

$$\left(\frac{l_t}{l_{t-1}} - 1\right) - \left(\frac{b_t}{b_{t-1}} - 1\right)$$

```
mpak['PAKNYGDPMKTPKN PAKNECONPRVTKN'].difpct.plot() ;
```

11.8.12 .difpctlevel percent difference of levels

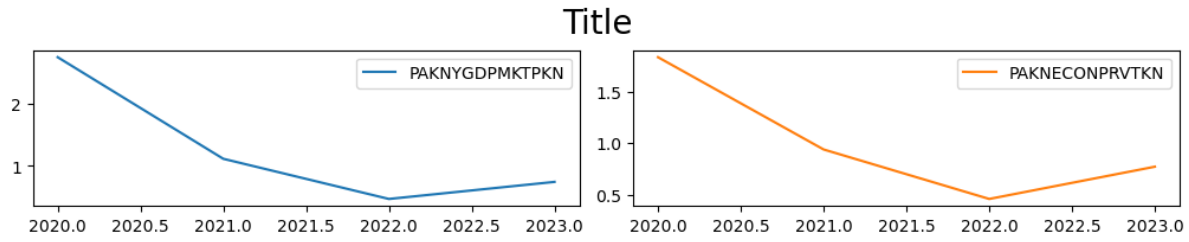
$$\left(\frac{l_t - b_t}{b_t}\right)$$

```
mpak['PAKNYGDPMKTPKN PAKNECONPRVTKN'].difpctlevel.plot();
```

11.8.13 mul100 multiply by 100

multiply growth rate by 100.

```
mpak['PAKNYGDPMKTPKN PAKNECONPRVTKN'].pct.mul100.plot()
```



11.9 .plot chart the selected and transformed variables

After the varaibles 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();
```

```
C:\Users\ibhan\miniconda3\envs\mfbooknew\lib\site-packages\pandas\plotting\_
 ↪matplotlib\tools.py:227: RuntimeWarning: More than 20 figures have been opened.␣
 ↪Figures created through the pyplot interface (`matplotlib.pyplot.figure`) are␣
 ↪retained until explicitly closed and may consume too much memory. (To control␣
 ↪this warning, see the rcParam `figure.max_open_warning`). Consider using␣
 ↪`matplotlib.pyplot.close()`.
  fig = plt.figure(**fig_kw)
```

11.9.1 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.
- top (TYPE, optional): Relative position of the title. Defaults to 0.90.

More excotic:

- 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.mul100.rename().plot(title='GDP growth ',top = 0.92);
```

11.10 Plotting inspiration

The following graph shows the components of GDP using the values of the baseline dataframe.

```
mpak['PAKNYGDPMKTPKN PAKNECONPRVTKN PAKNEGDIFTOTKN'].\
difpctlevel.mul100.rename().\
plot(title='Components of GDP in pct of baseline',colrow=1,top=0.90,kind='bar') ;
```

11.10.1 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):
    heatmap = mpak['PAKNYGDPMKTPKN PAKNECONPRVTKN'].pct.rename().mul100.heat(title=
 ↪'Growth rates',annot=True,dec=1,size=(10,3))
```

11.10.2 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()
```

11.10.3 Plot baseline vs alternative

A raw routine, only showing levels. To make it really useful it should be expanded.

```
mpak['PAKNYGDPMKTPKN PAKNECONPRVTKN'].plot_alt() ;
```

11.11 .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 the upstream variable to the impact on the downstream variable.

11.11.1 .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)  # diagram of all direct␣
 ↪dependencies
```

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

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

11.11.2 .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)
```

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

11.12 dekomp() Attrribution of right hand side variables to change in result.

For more information on attribution look here

The dekomp command decomposes the contributions of the right hand side variables to the observed change in the left hand side variables.

```
with mpak.set_smpl(2021,2025):
    mpak['PAKNYGDPMKTPKN PAKNECONPRVTKN'].dekomp()  # frml attribution
```

```
Formula        : FRML  <> PAKNYGDPMKTPKN =␣
 ↪PAKNECONPRVTKN+PAKNECONGOVTKN+PAKNEGDIFTOTKN+PAKNEGDISTKBKN+PAKNEEXPGNFSKN-
 ↪PAKNEIMPGNFSKN+PAKNYGDPDISCKN+PAKADAP*PAKDISPREPKN $

                       2021        2022        2023        2024        2025
Variable    lag
Base        0   26511370.41 26685141.87 26963077.57 27393200.36 27963231.53
Alternative 0   26764926.87 26889649.52 27089036.50 27454422.35 27979057.19
Difference  0     253556.46   204507.65   125958.93    61221.99    15825.66
Percent     0          0.96        0.77        0.47        0.22        0.06

 Contributions to differende for  PAKNYGDPMKTPKN
                         2021       2022       2023       2024       2025
Variable       lag
PAKNECONPRVTKN 0   -312052.97 -394466.14 -474558.93 -531755.17 -563616.01
PAKNECONGOVTKN 0    303335.99  268694.45  232506.87  209988.19  197439.80
PAKNEGDIFTOTKN 0    188565.48  188222.74  177226.36  163571.78  148739.93
PAKNEGDISTKBKN 0        -0.02      -0.01      -0.02      -0.02      -0.05
PAKNEEXPGNFSKN 0     -2911.23   -5414.50   -7960.34  -10272.64  -12204.84
PAKNEIMPGNFSKN 0     76619.12  147471.06  198744.89  229689.74  245466.56
PAKNYGDPDISCKN 0        -0.02      -0.01      -0.02      -0.02      -0.05
PAKADAP        0        -0.02      -0.01      -0.02      -0.02      -0.05
PAKDISPREPKN   0        -0.02      -0.01      -0.02      -0.02      -0.05

 Share of contributions to differende for  PAKNYGDPMKTPKN
                          2021        2022        2023        2024        2025
Variable       lag
PAKNEIMPGNFSKN 0           30%         72%        158%        375%       1551%
PAKNECONGOVTKN 0          120%        131%        185%        343%       1248%
PAKNEGDIFTOTKN 0           74%         92%        141%        267%        940%
PAKNEGDISTKBKN 0           -0%         -0%         -0%         -0%         -0%
PAKNYGDPDISCKN 0           -0%         -0%         -0%         -0%         -0%
PAKADAP        0           -0%         -0%         -0%         -0%         -0%
```

(continues on next page)

(continued from previous page)

```
PAKDISPREPKN    0           -0%         -0%         -0%         -0%         -0%
PAKNEEXPGNFSKN  0           -1%         -3%         -6%        -17%        -77%
PAKNECONPRVTKN  0         -123%       -193%       -377%       -869%      -3561%
Total           0          100%        100%        100%        100%        100%
Residual        0           -0%         -0%         -0%         -0%         -0%

 Contribution to growth rate PAKNYGDPMKTPKN
                           2021        2022        2023        2024        2025
Variable        lag
PAKNECONPRVTKN  0         -0.0%       -0.0%       -0.0%       -0.0%       -0.0%
PAKNECONGOVTKN  0          0.0%        0.0%        0.0%        0.0%        0.0%
PAKNEGDIFTOTKN  0          0.0%        0.0%        0.0%        0.0%        0.0%
PAKNEGDISTKBKN  0         -0.0%       -0.0%       -0.0%       -0.0%       -0.0%
PAKNEEXPGNFSKN  0         -0.0%       -0.0%       -0.0%       -0.0%       -0.0%
PAKNEIMPGNFSKN  0          0.0%        0.0%        0.0%        0.0%        0.0%
PAKNYGDPDISCKN  0         -0.0%       -0.0%       -0.0%       -0.0%       -0.0%
PAKADAP         0         -0.0%       -0.0%       -0.0%       -0.0%       -0.0%
PAKDISPREPKN    0         -0.0%       -0.0%       -0.0%       -0.0%       -0.0%

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

                        2021        2022        2023        2024        2025
Variable    lag
Base        0   23972815.36 24164128.02 24427863.05 24818524.47 25323255.17
Alternative 0   23660762.40 23769661.89 23953304.14 24286769.32 24759639.22
Difference  0    -312052.95  -394466.13  -474558.91  -531755.15  -563615.95
Percent     0         -1.30       -1.63       -1.94       -2.14       -2.23

 Contributions to differende for  PAKNECONPRVTKN
                           2021       2022       2023       2024       2025
Variable         lag
PAKNECONPRVTKN   -1  -187434.07 -250462.33 -317486.72 -384175.74 -432745.55
PAKNECONPRVTKN_A  0       -0.01      -0.01      -0.01      -0.01      -0.04
PAKBXFSTREMTCD   -1   -38694.42  -49412.27  -52084.76  -50817.15  -48170.52
PAKBMFSTREMTCD   -1      120.58     140.33     135.37     121.42     106.31
PAKPANUSATLS     -1     3137.57    3566.27    3817.26    3916.63    3901.04
PAKGGEXPTRNSCN   -1    -2382.89   -4372.94   -5966.91   -7223.04   -8206.86
PAKNYYWBTOTLCN   -1   -78794.18 -120093.04 -145773.43 -156461.75 -167189.37
PAKGGREVDRCTXN   -1       -0.01      -0.01      -0.01      -0.01      -0.04
PAKNECONPRVTXN   -1   204247.65  231199.67  249025.22  258789.48  262005.59
PAKBXFSTREMTCD    0    66466.87   69836.78   67727.29   63861.51   59511.44
PAKBMFSTREMTCD    0     -189.25    -182.00    -162.26    -141.32    -122.29
PAKPANUSATLS      0    -4809.78   -5132.40   -5233.84   -5184.63   -5043.02
PAKGGEXPTRNSCN    0     5895.35    8018.71    9646.92   10900.77   11850.22
PAKNYYWBTOTLCN    0   160980.65  194563.25  207466.32  220404.39  237003.52
PAKGGREVDRCTXN    0       -0.01      -0.01      -0.01      -0.01      -0.04
```

(continues on next page)

(continued from previous page)

```
PAKNECONPRVTXN    0  -410022.04 -440677.37 -455322.70 -458425.12 -453478.88
DURING_2009       0       -0.01      -0.01      -0.01      -0.01      -0.04
PAKFMLBLPOLYXN    0   -34994.35  -36409.85  -35203.57  -32189.52  -28049.75
PAKNECONPRVTKN_D  0       -0.01      -0.01      -0.01      -0.01      -0.04
PAKNECONPRVTKN_X  0       -0.01      -0.01      -0.01      -0.01      -0.04

 Share of contributions to differende for  PAKNECONPRVTKN
                          2021        2022        2023        2024        2025
Variable         lag
PAKNECONPRVTXN    0       131%        112%         96%         86%         80%
PAKNECONPRVTKN   -1        60%         63%         67%         72%         77%
PAKNYYWBTOTLCN   -1        25%         30%         31%         29%         30%
PAKBXFSTREMTCD   -1        12%         13%         11%         10%          9%
PAKFMLBLPOLYXN    0        11%          9%          7%          6%          5%
PAKGGEXPTRNSCN   -1         1%          1%          1%          1%          1%
PAKPANUSATLS      0         2%          1%          1%          1%          1%
PAKBMFSTREMTCD    0         0%          0%          0%          0%          0%
PAKNECONPRVTKN_A  0         0%          0%          0%          0%          0%
PAKGGREVDRCTXN   -1         0%          0%          0%          0%          0%
                  0         0%          0%          0%          0%          0%
DURING_2009       0         0%          0%          0%          0%          0%
PAKNECONPRVTKN_D  0         0%          0%          0%          0%          0%
PAKNECONPRVTKN_X  0         0%          0%          0%          0%          0%
PAKBMFSTREMTCD   -1        -0%         -0%         -0%         -0%         -0%
PAKPANUSATLS     -1        -1%         -1%         -1%         -1%         -1%
PAKGGEXPTRNSCN    0        -2%         -2%         -2%         -2%         -2%
PAKBXFSTREMTCD    0       -21%        -18%        -14%        -12%        -11%
PAKNYYWBTOTLCN    0       -52%        -49%        -44%        -41%        -42%
PAKNECONPRVTXN   -1       -65%        -59%        -52%        -49%        -46%
Total             0       101%        101%        101%        101%        101%
Residual          0         1%          1%          1%          1%          1%

 Contribution to growth rate PAKNECONPRVTKN
                          2021        2022        2023        2024        2025
Variable         lag
PAKNECONPRVTKN   -1        0.0%        0.0%        0.0%        0.0%        0.0%
PAKNECONPRVTKN_A  0       -0.0%       -0.0%       -0.0%       -0.0%       -0.0%
PAKBXFSTREMTCD   -1       -0.0%       -0.0%       -0.0%       -0.0%       -0.0%
PAKBMFSTREMTCD   -1        0.0%        0.0%        0.0%        0.0%        0.0%
PAKPANUSATLS     -1        0.0%        0.0%        0.0%        0.0%        0.0%
PAKGGEXPTRNSCN   -1       -0.0%       -0.0%       -0.0%       -0.0%       -0.0%
PAKNYYWBTOTLCN   -1       -0.0%       -0.0%       -0.0%       -0.0%       -0.0%
PAKGGREVDRCTXN   -1       -0.0%       -0.0%       -0.0%       -0.0%       -0.0%
PAKNECONPRVTXN   -1        0.0%        0.0%        0.0%        0.0%        0.0%
PAKBXFSTREMTCD    0        0.0%        0.0%        0.0%        0.0%        0.0%
PAKBMFSTREMTCD    0       -0.0%       -0.0%       -0.0%       -0.0%       -0.0%
PAKPANUSATLS      0       -0.0%       -0.0%       -0.0%       -0.0%       -0.0%
PAKGGEXPTRNSCN    0        0.0%        0.0%        0.0%        0.0%        0.0%
PAKNYYWBTOTLCN    0        0.0%        0.0%        0.0%        0.0%        0.0%
PAKGGREVDRCTXN    0       -0.0%       -0.0%       -0.0%       -0.0%       -0.0%
PAKNECONPRVTXN    0       -0.0%       -0.0%       -0.0%       -0.0%       -0.0%
DURING_2009       0       -0.0%       -0.0%       -0.0%       -0.0%       -0.0%
PAKFMLBLPOLYXN    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%
```

11.13 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 bespoken plots the they can be constructed directly in python scripts. The two main libraries are matplotlib, plotly but any ther python plotting library can be used. Here is an example using matplotlib.

11.14 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()

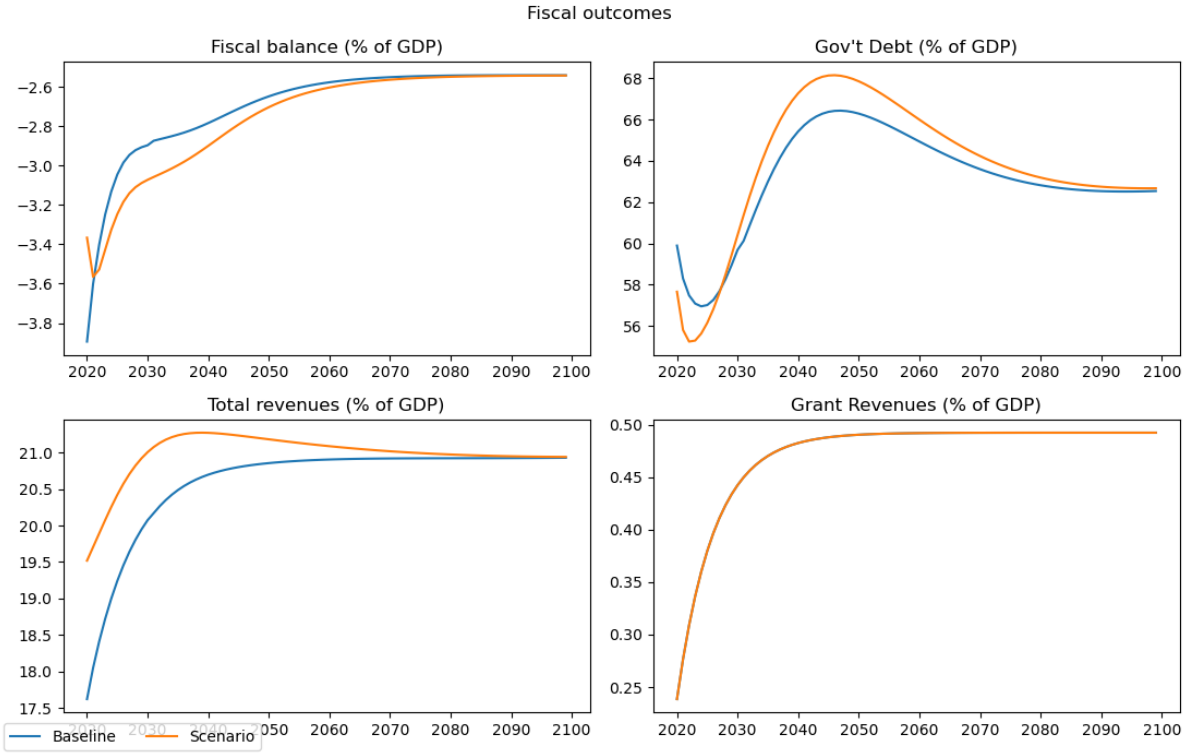
axs[0,1].plot(mpak.basedf.loc[2020:2099,'PAKGGDBTTOTLCN_'],label='Baseline')
axs[0,1].plot(mpak.lastdf.loc[2020:2099,'PAKGGDBTTOTLCN_'],label='Scenario')

axs[1,0].plot(mpak.basedf.loc[2020:2099,'PAKGGREVTOTLCN']/mpak.basedf.loc[2020:2099,
 ↪'PAKNYGDPMKTPCN']*100,label='Baseline')
axs[1,0].plot(mpak.lastdf.loc[2020:2099,'PAKGGREVTOTLCN']/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
```


Fiscal outcomes
Fiscal balance (% of GDP)
Gov't Debt (% of GDP)
Total revenues (% of GDP)
Grant Revenues (% of GDP)
Baseline
Scenario

Part VI

More

BIBLIOGRAPHY

[Bla18] Olivier Blanchard. On the future of Macroeconomic models. *Oxford Review of Economic Policy*, 34(1-2):43–54, 2018. URL: https://academic.oup.com/oxrep/article/34/1-2/43/4781808, doi:https://doi.org/10.1093/oxrep/grx045.

[BCJ+19] Andrew Burns, Benoit Campagne, Charl Jooste, David Stephan, and Thi Thanh Bui. *The World Bank Macro-Fiscal Model Technical Description*. Number 8965 in Policy Research Working Papers. World Bank, Washington DC., 2019. URL: https://openknowledge.worldbank.org/handle/10986/32217.

[BJS21a] Andrew Burns, Charl Jooste, and Gregor Schwerhoff. *Climate Modeling for Macroeconomic Policy : A Case Study for Pakistan*. Number 9780 in Policy Research Working Papers. World Bank, Washington, DC, 2021. URL: https://openknowledge.worldbank.org/bitstream/handle/10986/36307/Climate-Modeling-for-Macroeconomic-Policy-A-Case-Study-for-Pakistan.pdf?sequence=1&isAllowed=y.

[BJS21b] Andrew Burns, Charl Jooste, and Gregor Schwerhoff. *Macroeconomic Modeling of Managing Hurricane Damage in the Caribbean: The Case of Jamaica*. Volume 9505 of Policy Research Working Paper. World Bank, Washington DC., 2021. URL: https://documents1.worldbank.org/curated/en/593351609776234361/pdf/Macroeconomic-Modeling-of-Managing-Hurricane-Damage-in-the-Caribbean-The-Case-of-Jamaica.pdf.

[DJ13] Peter B Dixon and DFale W. Jorgenson. *Handbook of Computable General Equilibrium Modelling*. Volume 1A. Elsevier B.V., 2013. ISBN ISSN: 2211-6885. URL: https://www.sciencedirect.com/handbook/handbook-of-computable-general-equilibrium-modeling/.