DRAFT: ModelFlow, A library to manage and solve Models

Ib Hansen [1]

May 20, 2019

[1]European Central Bank, email to ib.hansen@ecb.europa.eu

Contents

Chapter 1

ModelFlow, A library to manage and solve Models

```
In [1]: %matplotlib inline  # initialization
```

1.1 Why Model flow?

Specify very large (or small) models as concise and intuitive equations. The program to calculate/solve the model is automatic generated. The user don't have to do the household chores.

Large models. Models with many banks and granular data can get very large. 1 million equation and more can be handled.

Agile model development Model are specified at a high level of abstraction and are processed fast. Also solving/calculating models are fast. This makes experiments with model specification agile and fast.

Get rid of models models implemented in Excel. Large models in Excel are difficult to maintain, revise and quality check. Excel is also very slow in calculating models. Moreover Excel don't handle models with contemporaneous feedback well.

Onboarding models and combining from different sources. Creating a Macro prudential model often entails recycling several models specified in different ways: Excel, Latex, Dynare, Python or other languages. Python's ecosystem makes it possible to transform many different models into ModelFlow models or to wrap them into functions which can be called from ModelFlow models.

Onboarding data from different sources. Pythons Pandas Library and other tools are fast and efficient for data management.

A rich set of analytical tools for model and result analytic helps to understand the model and its results.

The user can extend and modify the tools to her or his needs. All code is in Python and the core is quite small.

1.2 Introduction

The main purpose of ModelFlow is to bring a model and a data set together**. Then the model can be solved, and the results analyzed and compared. The user can do this either with the Python methods that ModelFlow will wrap around the model, or by by using tools from the Python ecosystem.

Models can be specified in a high level Business logic language (a Domain Specific language). This allows the formulation of a model in a concise and expressive language which is close to the economic of the model. The user can concentrate on the economic or financial content - not the coding of the solution. The code for solving the model is generated by the tool. Then you can *solve the simultaneous* (or *non-simultaneous* model) in an efficient way.

Thus a model **flows** through a number of phases, from a formulation in business logic language to a Python program which can solve the model.

If the model is not initially specified in the Business logic language, Python offers a wide range of possibilities to grab a model and transform it to the Business logic language. Also, models in the form of Matlab functions can be wrapped and called from ModelFlow. This allows for the recycling and integration models sourced from a number of different origins.

ModelFlow is written in Python. Python comes "batteries included" and is the basis of a very rich ecosystem, which consists of a wide array of libraries. ModelFlow is just another library. It supplements the existing libraries regarding modeling language and solving and allows the use of Python as a model management framework.

Data handling and wrangling is done in the Pandas library. This library is the Swiss army knife of data science in Python. It can import and export data to most systems and it is very powerful in manipulating and transforming data. The core element of Pandas is the *Dataframe*. A Dataframe is a two-dimensional tabular data structure. Each *column* consists of cells of the same type -- it can be a number, a string, a matrix or another Python data object.This includes matrices and other dataframes. Each *row is indexed.* The index can basically be any type of variable including dates, which is especially relevant for economic and financial models.

ModelFlow gives the user tools for more than solving models. This includes:

- *Visualization* and comparison of results
- *Integration* of models from different sources
- *Analyze the logical structure of a model*. By applying graph theory, ModelFlow can find data lineage, find a suitable calculating sequence and trace causes of changes through the calculations.
- *Inverting* the model to calculating the necessary instruments to achieve a desired target.
- Calculating the *attributions* from input to the results of a model.
- Calculating the *attribution* from input to the result of each formula.
- Finding and calculating partial *derivatives* of formulas
- *Integrating user defined python functions* in the Business logic language (like optimization, calculating risk weights or to make a matrices consistent with the RAS algorithm)
- *Wrap matlab* models so they can be used in the Business logic language.
- *Speed up* solving using "Just in time compilation"
- Analyze the model structure through tools from graph theory
- Handle *large models.* 1,000,000 formulas is not a problem.
- Integrate model management in Jupyter notebooks for *agile and user friendly model use*

The core code of ModelFlow is small and documented. Thus it can easily be modified and expanded to the specific need of the user. *ModelFlow is a toolset*. It can handle models, which conform to the tools.

If you need a feature or have a model which can't be handled in ModelFlow, you are encouraged to improve ModelFlow. Please share the improvement, other users may have the same need, or can be inspired by your work.

Also bear in mind that ModelFlow is experimental. It is provided "as is", without any representation or warranty of any kind either express or implied.

The code is located here: `https://github.com/IbHansen/ModelFlow`

The rest of this document will start with a simple example, then it will go into more detail regarding how to use (some of) the features.

1.3 This document

This document is available in two forms: as a **Jupyter notebook** and as a **PDF document**.

The **Jupyter notebook** version can most easily be accessed through a virtual machine. You can run the examples and change the input cells and immediately watch the result.

If you are viewing the workbook through Github, you might find some of the in Latex don't render correct.

If you are reading the **PDF document** you can read, but not update and experiment with the examples in the document.

1.4 Run this notebook on a virtual machine.

To get access to the jupyter notebooks in the repo, run this:

https://mybinder.org/v2/gh/Ibhansen/ModelFlow/master

This will open a Home tab in your browser. The notebooks are the .ipynb files.

The virtual machine is created using Binder .

Select **modelflow.ipynb** to open this notebook. You can scroll through it, and you can run it. Each cell is run by pressing **shift enter**. You can also run the whole notebook by **Cell>Run All** on the toolbar.

Using the toolbar you can toggle the Table of content, and you can activate the RISE. RISE will allow you to view the notebook as a slideshow.

To benefit fully of the notebook look into how Jupyter works:

https://jupyter-notebook.readthedocs.io/en/stable/examples/Notebook/Notebook%20Basics.html

1.5 Use ModelFlow on your local machine

ModelFlow libraries are located in a Github repo which you can download. It is located here:

`https://github.com/IbHansen/ModelFlow`

You need Python 3.6+ with associated libraries. The easy way to get Python is to install Anaconda Python from here:

`https://www.anaconda.com/distribution`

In addition to the standard packages in the Anaconda distribution you need: Graphviz and cvxopt: they can be installed by running a command window from the Anaconda prompt and stating

`conda install graphviz conda install cvxopt`

You will find the anaconda prompt by searching "anaconda" in the start menu search field.

The PYTHONPATH enviroment variable has to point to the location og the ModelFlow .py files. This is most easily done through the Spyder editor:tools>PYTHONPATH manager. Add the pahth and press the synchronize button to synchronize the spyder PYTHONPATH with the environment variable.

1.6 Background.

Python for Financial Stability (ModelFlow) was initially created in order to facilitate[1] a major revision of the top-down stress testing model used in Danmarks Nationalbank.

In order to facilitate the integration of Danmarks Nationalbanks macro model (MONA) with the stress test model, the ability to handle simultaneous models were introduced[2].

ModelFlow has been going through a major refactoring and expansion during my stay at ECB.

ModelFlow can be used as a calculation engine for simple calculation tasks. It also allows efficient formulation, data handling and solving of conforming macroeconomic, stress testing and network models.

ModelFlow provides an agile platform to develop system wide models. Such models can incorporate a number of different feedback and behavioral mechanisms.

ModelFlow is a Python toolkit. So the user need to be comfortable both with the model at hand and with Python

Chapter 2

A small example model with ModelFlow

ModelFlow is used to bring a model in the form of equations/formulas together with data in the form of a Pandas Dataframe, solve the model and place the solution in a Pandas Dataframe.

We will start with a simple model which will be used to illustrate the flow of models and data.

2.1 Import relevant libraries

Here the nessecary libraries are imported. Imports starting with *model* are part of the ModelFlow library

```
In [2]: import matplotlib.pyplot as plt
        import pandas as pd                # Python data science library
        import numpy as np
        import sys
        sys.path.append('modelflow/')


        import modelclass as mc            # ModelFlow, model handeling library
        import modelmanipulation as mp     # ModelFlow librery to text process models
        import modelpattern as pt
        from modelmanipulation import explode
        import modeldekom as md
        import modelvis as mv
        import modelinvert as mi
        import modelnet as mn
        import modeldiff as mdif
```

2.2 Create model equations / formulas

The model equations are a Python string. In a simple model the calculations are specified just as formulas. Time is implicit and lags are marked by (-).

Each formula starts with a "FRML" and ends with a $. Also each formula has formula options enclosed in <>. In this simple model we don't use any advanced features.

You will notice, that the formulas do not have to be ordered in the logical calculation sequence. This will be done by ModelFlow.

```
In [3]: model      = '''
        frml <> a = c(-1) + b $
        frml <> d1 = x + 3 * a(-1)+ c **2 +a  $
        frml <> d3 = x + 3 * a(-1)+c **3 $
        Frml <> x = 0.5 * c $'''
```

2.3 Create a model instance

Now we want to take the formulas (equations) and create a devise which can run the model. This device is an instance of the **model class**.

After the creation the instance can solve the model, visualize results and provide information regarding the model

```
In [4]: mmodel = mc.model(model,modelname = 'My first model')
```

2.4 Create a Dataframe

The model needs some data to run. So we create a pandas dataframe.

A dataframe is a matrix, where the columns designate variables in the model, and rows are time dimensions.

```
In [5]: df = pd.DataFrame({'B': [1,1,1,1],'C':[1,2,3,6],'E':[4,4,4,4]},index=[2018,2019,2020,2021])
        df

Out[5]:       B  C  E
        2018  1  1  4
        2019  1  2  4
        2020  1  3  4
        2021  1  6  4
```

2.5 Run the model

We can now run the model using the data. The result is a new dataframe.

If the dataframe don't contain all the variables from the model, the dataframe will be padded with the missing variables. Their values are set to 0.0.

The model can be solved for the rows in the dataframe for which it is possible. This means that the maximal lags has to be taken into account. In this example the maximal lag is 1, so the model can be solved from 2019 to 2021.

```
In [6]: mmodel(df)

Will start calculating: My first model
2019  solved
```

```
2020  solved
2021  solved
My first model calculated
```

```
Out[6]:        B    C    E    X     D3    D1    A
        2018  1.0  1.0  4.0  0.0    0.0   0.0  0.0
        2019  1.0  2.0  4.0  1.0    9.0   7.0  2.0
        2020  1.0  3.0  4.0  1.5   34.5  19.5  3.0
        2021  1.0  6.0  4.0  3.0  228.0  52.0  4.0
```

2.6 Make a new experiment

One experiment is useful, but usually we want to compare the results of several experiments.

So the C variable is updated to new values, and the model is run again.

```
In [7]: df.C = [4,5,7,9]
        df.B = [1,2,3,4]
        mmodel(df)
```

```
Will start calculating: My first model
2019  solved
2020  solved
2021  solved
My first model calculated
```

```
Out[7]:        B    C    E    X     D3     D1     A
        2018  1.0  4.0  4.0  0.0    0.0    0.0   0.0
        2019  2.0  5.0  4.0  2.5  127.5   33.5   6.0
        2020  3.0  7.0  4.0  3.5  364.5   78.5   8.0
        2021  4.0  9.0  4.0  4.5  757.5  120.5  11.0
```

2.7 Visualizing

A model run returns a dataframe with the results. These can be analyzed and visualized by python libraries like pandas and matplotlib.

However, the results are also saved as propertied in the model instance. By using the information regarding both the model and the results a number of tools for visualizing, analyzing and debugging can applied.

Per default the first result dataframe and the last result dataframe are retained in the model instance. The first is called:**basedf** the last is called:**lastdf**.

By setting or accessing **model.basedf** and **model.lastdf**, these properties can also be set or retrieved by the user.

2.7.1 Single variable

All model variable can be accessed with model_instance.variable_name notation. The resulting object will return information regarding the variable.

The python feature "Tab-completion" can be used to select which variable to show. So when the user pres the tab-key, a list of matching variable names will be displayed, and the user can those from the list.

For a single variable there are a number of methods for gaining insight in both the dependencies and the results. If we just give the variable as attribute like in the cell below. We will get a some basic information regarding the variable.

```
In [8]:                                     mmodel.D1
```

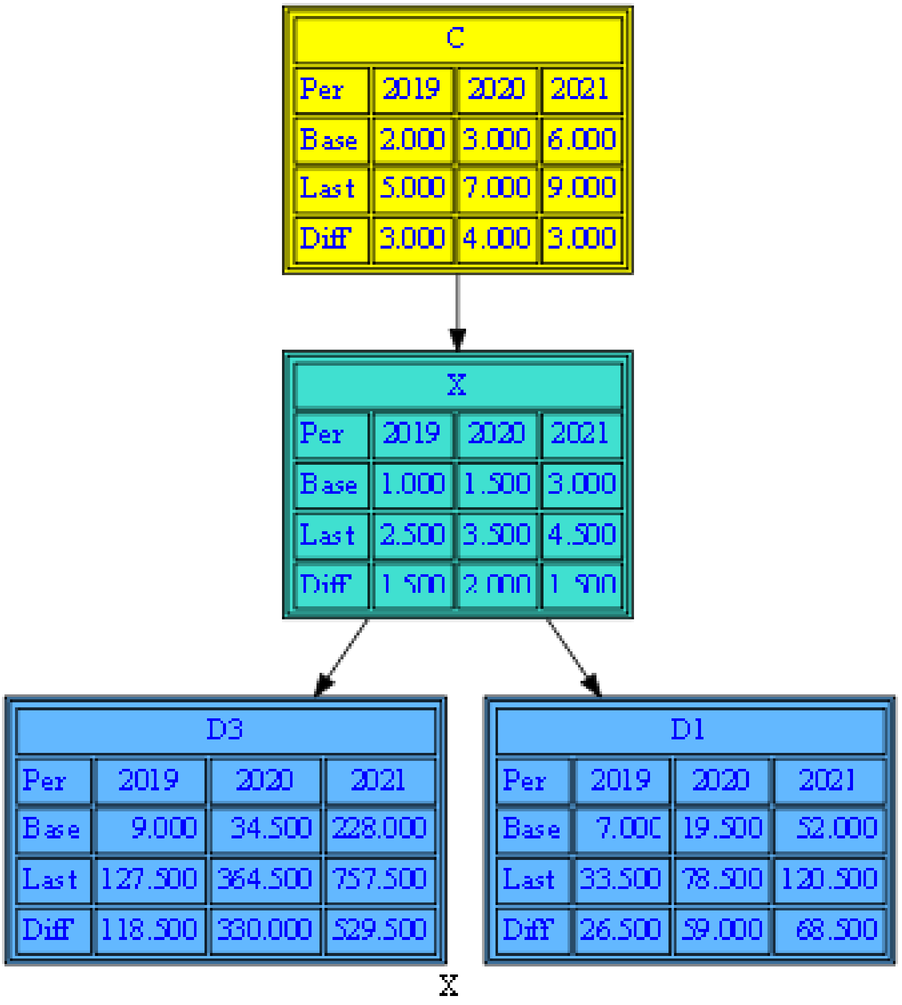
```
Out[8]: Endogeneous: D1
        Formular: FRML <> D1 = X + 3 * A(-1)+ C **2 +A  $
        Values :
              2019  2020   2021
        Base   7.0  19.5   52.0
        Last  33.5  78.5  120.5
        Diff  26.5  59.0   68.5

        Input last run:
                2019  2020  2021
        A        6.0   8.0  11.0
        A(-1)    0.0   6.0   8.0
        C        5.0   7.0   9.0
        X        2.5   3.5   4.5
```

Some methods are available to use after the variable name: One such is **draw**, the next cell shows the parent and child's of the X variable, and also the values of each of the variables. Both in the baseline run (default: the first run of the model) and in the last run of the model.

So in this model X depends on C, and X is contributing to D1 and D3.

Output from **draw** can become quite large for complex and deep models. Therefor the user can control both the amount of information for the variables and how many generations the graph has to cover.

```
In [9]: mmodel.draw('X',all=1,size=(4,4))
```



C			
Per	2019	2020	2021
Base	2.000	3.000	6.000
Last	5.000	7.000	9.000
Diff	3.000	4.000	3.000

X			
Per	2019	2020	2021
Base	1.000	1.500	3.000
Last	2.500	3.500	4.500
Diff	1.500	2.000	1.500

D3			
Per	2019	2020	2021
Base	9.000	34.500	228.000
Last	127.500	364.500	757.500
Diff	118.500	330.000	529.500

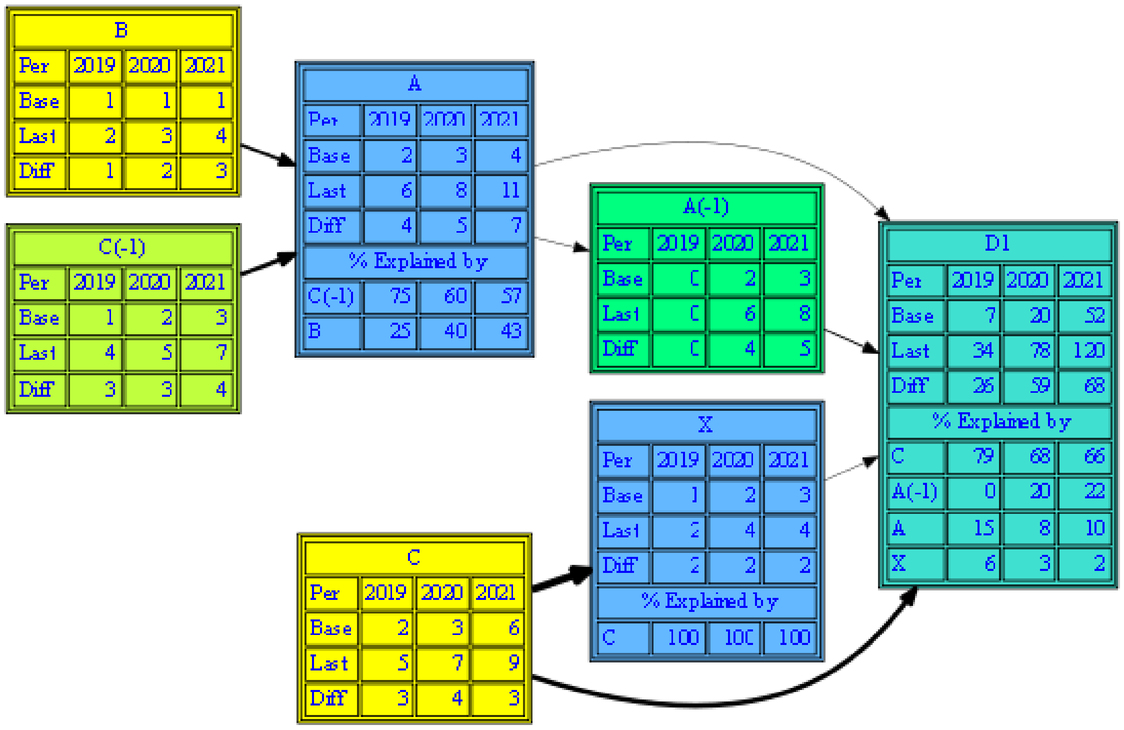
D1			
Per	2019	2020	2021
Base	7.000	19.500	52.000
Last	33.500	78.500	120.500
Diff	26.500	59.000	68.500

2.7.2 We can also explain the change in the variable

Having two runs of the model, it is useful to find the attribution to the change in values from the first to the second run. This is done by the explain method.

The width of arrows reflects how much of the change in a variable can be explained by the change in each incoming variable.

```
In [10]: _=mmodel.D1.explain(up=2)
```



B			
Per	2019	2020	2021
Base	1	1	1
Last	2	3	4
Diff	1	2	3

C(-1)			
Per	2019	2020	2021
Base	1	2	3
Last	4	5	7
Diff	3	3	4

A			
Per	2019	2020	2021
Base	2	3	4
Last	6	8	11
Diff	4	5	7
% Explained by			
C(-1)	75	60	57
B	25	40	43

A(-1)			
Per	2019	2020	2021
Base	C	2	3
Last	C	6	8
Diff	C	4	5

X			
Per	2019	2020	2021
Base	1	2	3
Last	2	4	4
Diff	2	2	2
% Explained by			
C	100	10C	100

C			
Per	2019	2020	2021
Base	2	3	6
Last	5	7	9
Diff	3	4	3

D1			
Per	2019	2020	2021
Base	7	20	52
Last	34	78	120
Diff	26	59	68
% Explained by			
C	79	68	66
A(-1)	0	20	22
A	15	8	10
X	6	3	2

D1 explained

This can become a very busy graph for large models where there are really many nodes. Like shocks for all countries in an EBA stress test. So take care what you ask for.

The output can also be produced in PDF or svg format. These formats are vector based, and it is possible to zoom into a large graph without loosing information.

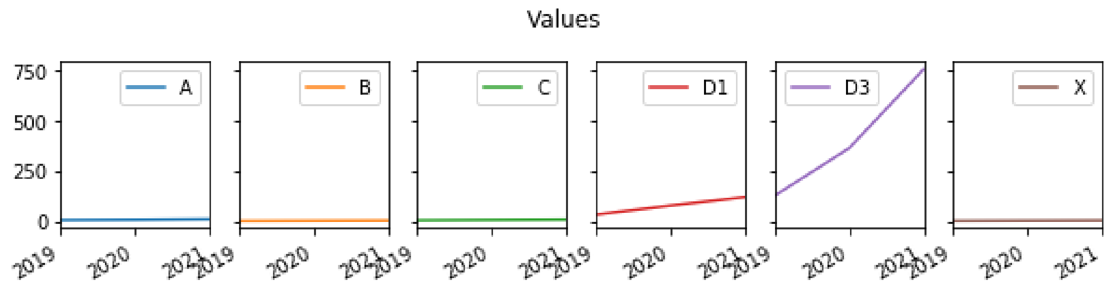
2.7.3 Slices of variables

In order to make life easy when using models with consistent and structured variable naming conventions wildcards can be used to select the variables to visualize through ModelFlow. Some are displayed below.

If this is not sufficient the whole suite of Python visualization libraries (as Matplotlib, Seaborn, Plotly) can be used on top of the resulting Dataframes.

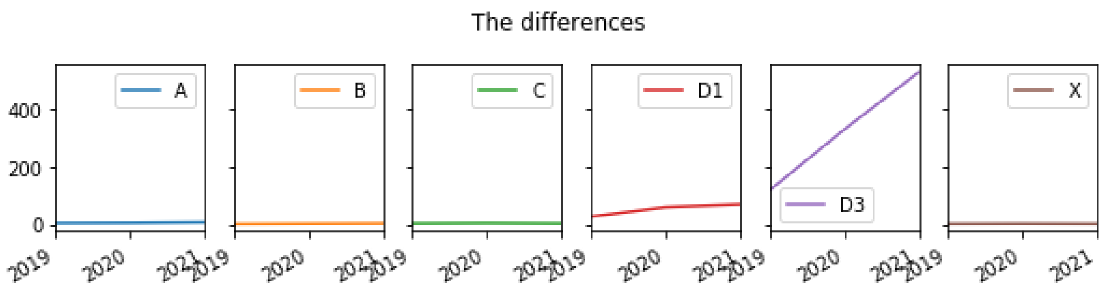
The values

```
In [11]: _=mmodel['*'].plot(title='Values')
```



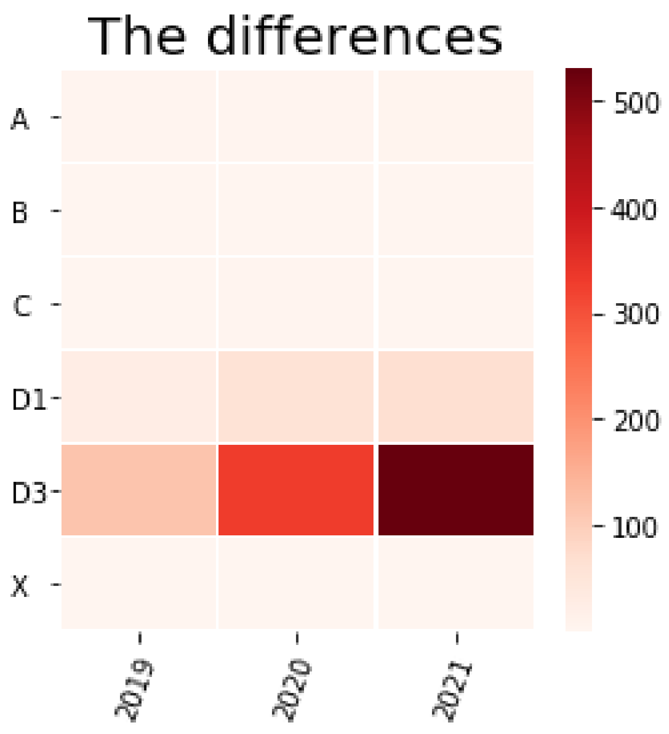
The values

```
In [12]: _=mmodel['*'].dif.plot(title='The differences')
```



The differences between the runs, as heatmap

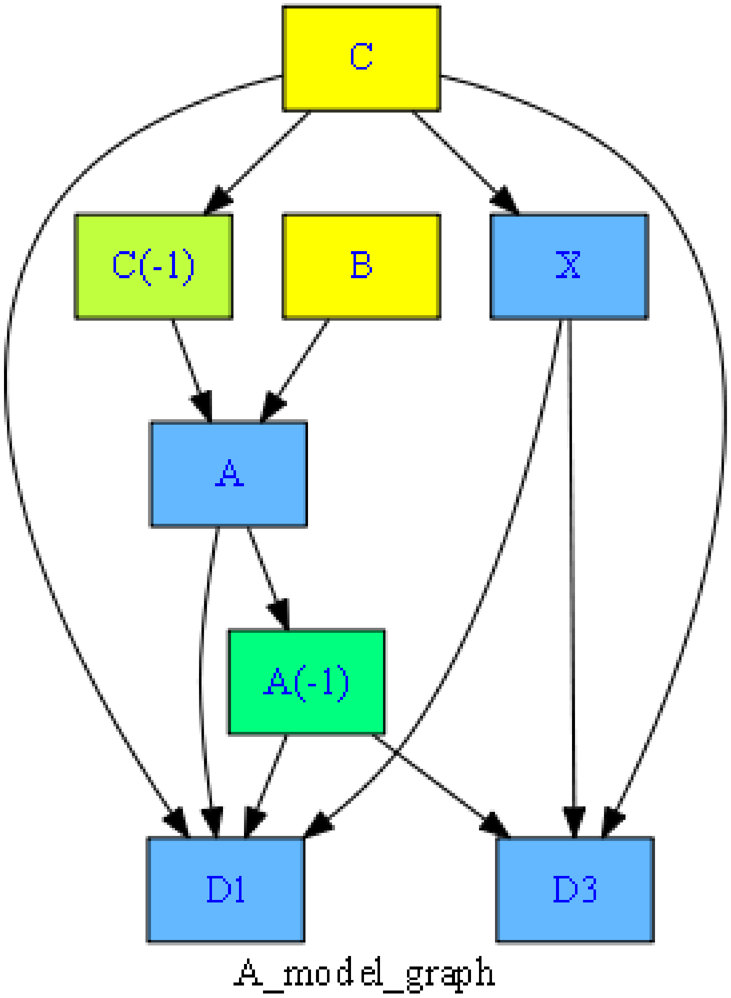
```
In [13]: _=mmodel['*'].dif.heat(size=(4,4),title='The differences')
```



2.7.4 Structure

As the structure of the model is in the model instance, we can visualize the structure of the whole model.

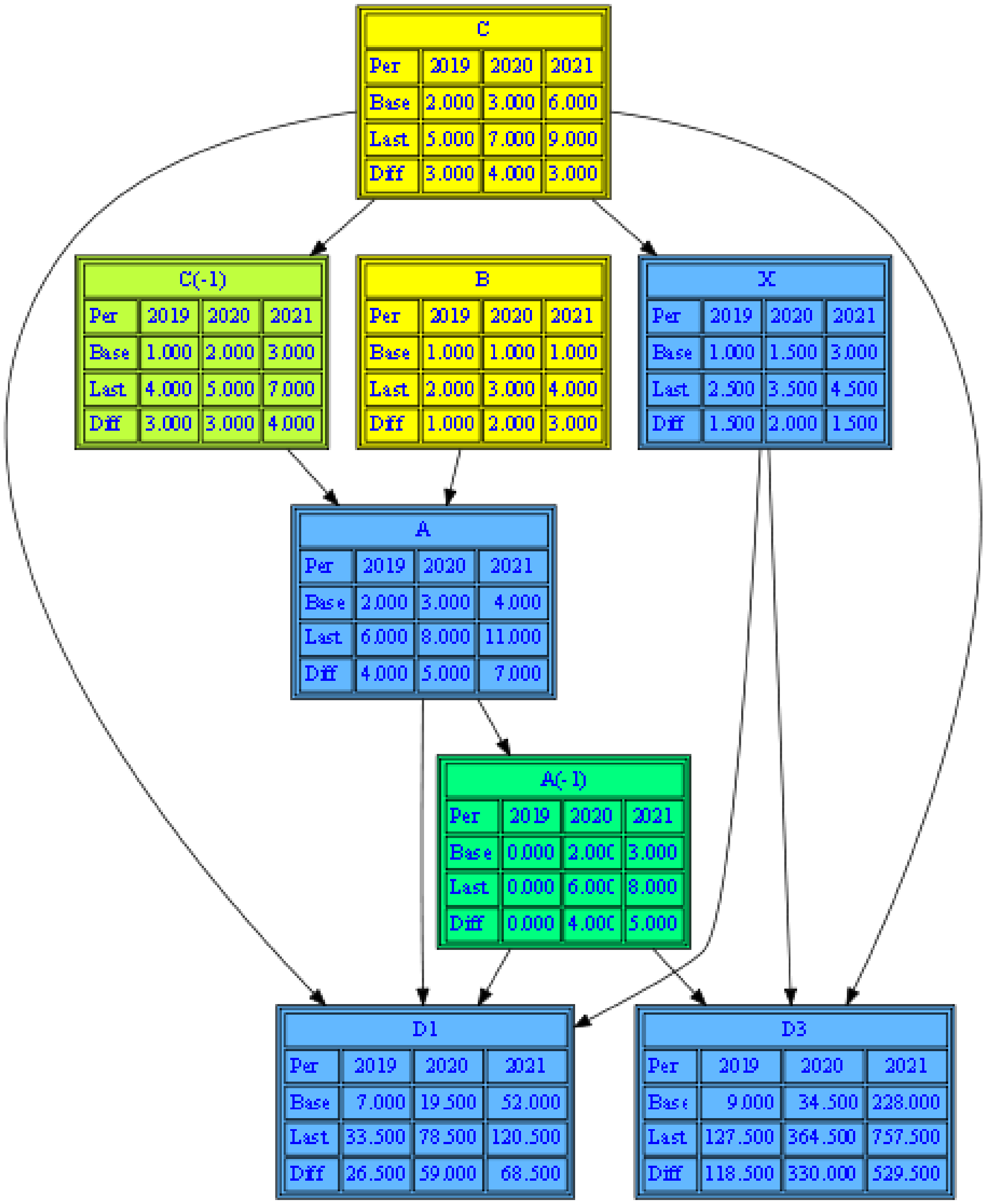
```
In [14]: mmodel.drawmodel(size=(4,4))
```



2.7.5 The model structure with values

Also we can combine the structure and the model in order to get the complete picture.
However take care for large models.

```
In [15]: mmodel.drawmodel(all=1)
```



C

Per	2019	2020	2021
Base	2.000	3.000	6.000
Last	5.000	7.000	9.000
Diff	3.000	4.000	3.000

C(-1)

Per	2019	2020	2021
Base	1.000	2.000	3.000
Last	4.000	5.000	7.000
Diff	3.000	3.000	4.000

B

Per	2019	2020	2021
Base	1.000	1.000	1.000
Last	2.000	3.000	4.000
Diff	1.000	2.000	3.000

X

Per	2019	2020	2021
Base	1.000	1.500	3.000
Last	2.500	3.500	4.500
Diff	1.500	2.000	1.500

A

Per	2019	2020	2021
Base	2.000	3.000	4.000
Last	6.000	8.000	11.000
Diff	4.000	5.000	7.000

A(-1)

Per	2019	2020	2021
Base	0.000	2.00	3.000
Last	0.000	6.00	8.000
Diff	0.000	4.00	5.000

D1

Per	2019	2020	2021
Base	7.000	19.500	52.000
Last	33.500	78.500	120.500
Diff	26.500	59.000	68.500

D3

Per	2019	2020	2021
Bas	9.000	34.500	228.000
Last	127.500	364.500	757.500
Diff	118.500	330.000	529.500

A_model_graph

This was a quick walk through how to define a simple model, make some mock data, run the model and visualize the structure and the results.

Now we can explore the features of ModelFlow.

Chapter 3

A model in ModelFlow

In this chapter we will first look at the type of models for which ModelFlow is designed. Then how to look at the structure of a model and finally on how to solve a model depending on feedback or not.

3.1 The model equations (formulas)

The scope of models handled by ModelFlow is models which *can* have lagged variables, but not leaded variables and follows this pattern:

$$\begin{aligned}
y_t^1 &= f^1(., y_t^2 ..., y_t^n ... y_{t-r}^1 ..., y_{t-r}^n, x_t^1 ... x_t^k, ... x_{t-s}^1 ..., x_{t-s}^k) && (3.1)\\
y_t^2 &= f^2(y_t^1 ..., y_t^n ... y_{t-r}^1 ..., y_{t-r}^n, x_t^1 ... x_t^k, ... x_{t-s}^1 ..., x_{t-s}^k) && (3.2)\\
&\vdots && (3.3)\\
y_t^n &= f^n(y_t^1 ..., y_t^{n-1} ... y_{t-r}^1 ..., y_{t-r}^n, x_t^1 ... x_t^r, x ..._{t-s}^1 ..., x_{t-s}^k) && (3.4)\\
& && (3.5)\\
& && (3.6)
\end{aligned}$$

Many stress test, liquidity, macro or other models conforms to this pattern. Or the can easily be transformed to thes pattern.

Written in matrix notation where $\mathbf{y}_t$ and $\mathbf{x}_t$ are vectors of endogenous/exogenous variables for time t

$$\begin{aligned}
\mathbf{y}_t &= \mathbf{F}(\mathbf{y}_t \cdots \mathbf{y}_{t-r}, \mathbf{x}_t \cdots \mathbf{x}_{t-s}) && (3.7)\\
& && (3.8)
\end{aligned}$$

The functions are normalized:

- Each endogenous variable is on the left hand side one time - and only one time.
- An endogenous variable without lags can **not** be on the right hand side in an equation, which has the variable on the left hand side.

Equations are the same for all projection time periods.
t is the same time frame (year, quarter for instance).

If aa function are stochastic, the error term will be an exogenous variable.

The purpose of ModelFlow is not to estimate the functions in the model. There are many excellent programs which can do estimation. ModelFlow is about model handling after specification.

ModelFlow allows variable to be scalars, tupels, matrices, arrays and pandas dataframes.

The functions in **F** can be specified in an **template business logic language** which allows concise and parsimonious definition of models. The **template business logic language** is preprocessed to **business logic language** which resemble python. This model is then transpiled to a python function which can solve the model.

3.2 The model structure

The structure of a model can be seen as a directed graph. All variables are node in the graph. If a variable b is on the right side of the formula defining variable a there is an edge from a to b.

3.2.1 First we define the nodes (vertices) of the dependency graph.

The set of nodes is the set of relevant variables. Actually we want to look at **two dependency graphs**: one containing *all variables*, and one only containing *endogenous contemporaneous variable* (the y_t^j's). So we define two sets S and E:

All endogenous, exogenous, contemporaneous and lagged variables

$S = \{y_{t-i}^j | j = 1..n, i = 1..r\} \cup \{x_{t-i}^j | j = 1..k, i = 1..s\}$

Contemporaneous endogenous variables

$E = \{y_t^j | j = 1..n\}$ contemporaneous endogenous variables

Naturally:

$E \subseteq S$

3.2.2 Then we define the edges of the dependency graph.

Again two sets are relevant:

From all variables to contemporaneous endogenous variables

$T = \{(a, b) | a \in E, b \in S\}$ a is on the right side of b

From contemporaneous endogenous variables to contemporaneous endogenous variables

$T_e = \{(a, b) | a \in E, b \in E\}$ a is on the right side of b

3.2.3 And we can construct useful dependency graphs

The we can define a graph TG which defines the data dependency of the model:

$TG = (S, T)$ The graph defined by nodes S and edges T.

TG can be used when exploring the dependencies in the model. This is useful for the user when drilling down the results.

However for preparing the solution a smaller graph has to be used. When solving the model for a specific period both exogenous and lagged endogenous variables are predetermined. Therefor we define the the dependency graph for contemporaneous endogenous variables:

$TE = (E, T_e)$ The graph defined by nodes S and edges T_e.

TE is used to determine if the model is simultaneous or not.

If the model is not simultaneous, then TE have no cycles, that is, it is a Directed Acyclical Graph (DAG). Then we can find an order in which the formulas can be calculated. This is called a topological order.

The topological order is a linear ordering of nodes (vertices) such that for every edge (v,u), node v comes before u in the ordering.

A topological order is created by doing a topological sort of TE.

If TE, the dependency graph associated with F is **not** a Directed Acyclical Graph (A DAG). Then F has contemporaneous feedback and is simultaneous. Or - in Excel speak - the model has circular references. And we need to use an iterative methods to solve the model. Sometime a model contains several simultaneous blocks. Then each block is a strong element of the graph. Each formula which is not part of a simultaneous bloc is in itself a strong element.

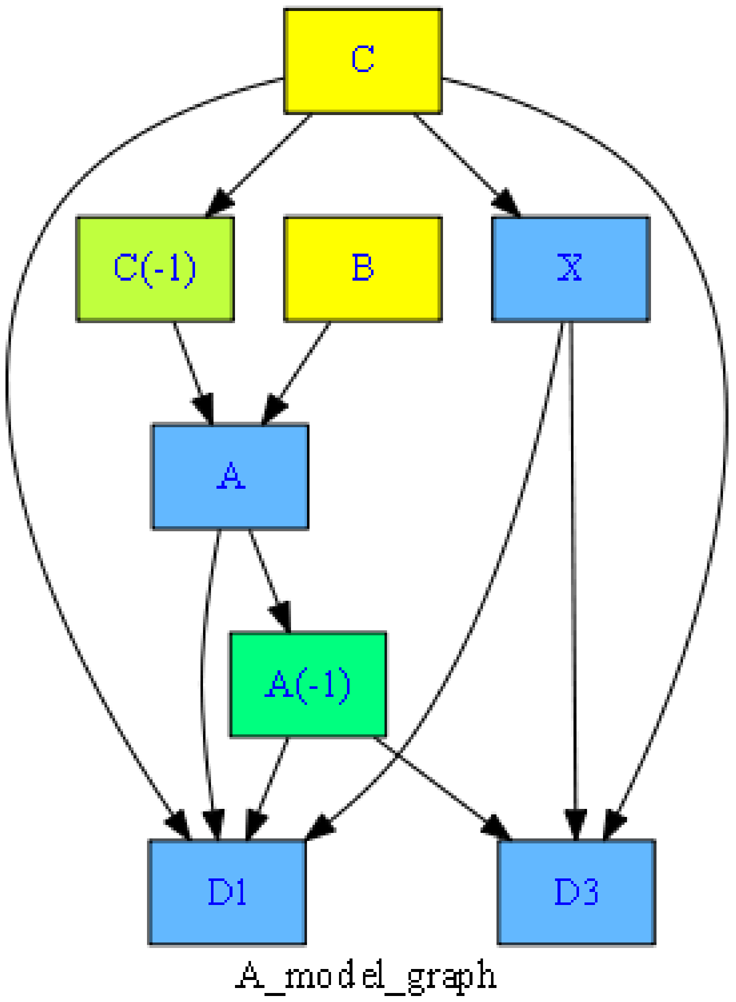
A condensed graph where each strong element is condensed to a node is a DAG. So the condensed graph have a topological order. This can be used when solving the model.

The dependency graphs are constructed, analyzed and manipulated through the **Networkx** Python library.

3.2.4 The complete dependency graph

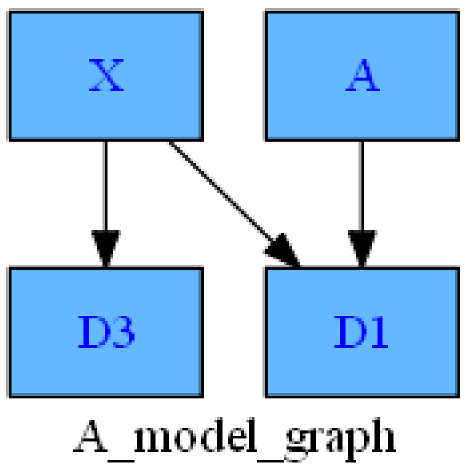
This shows TG mentioned above.

```
In [16]: mmodel.drawmodel(size=(4,4))
```



3.2.5 The dependency graph for contemporaneous endogenous variables (TE), no feedback

```
In [17]: mmodel.drawendo(sixe=(3,3))
```

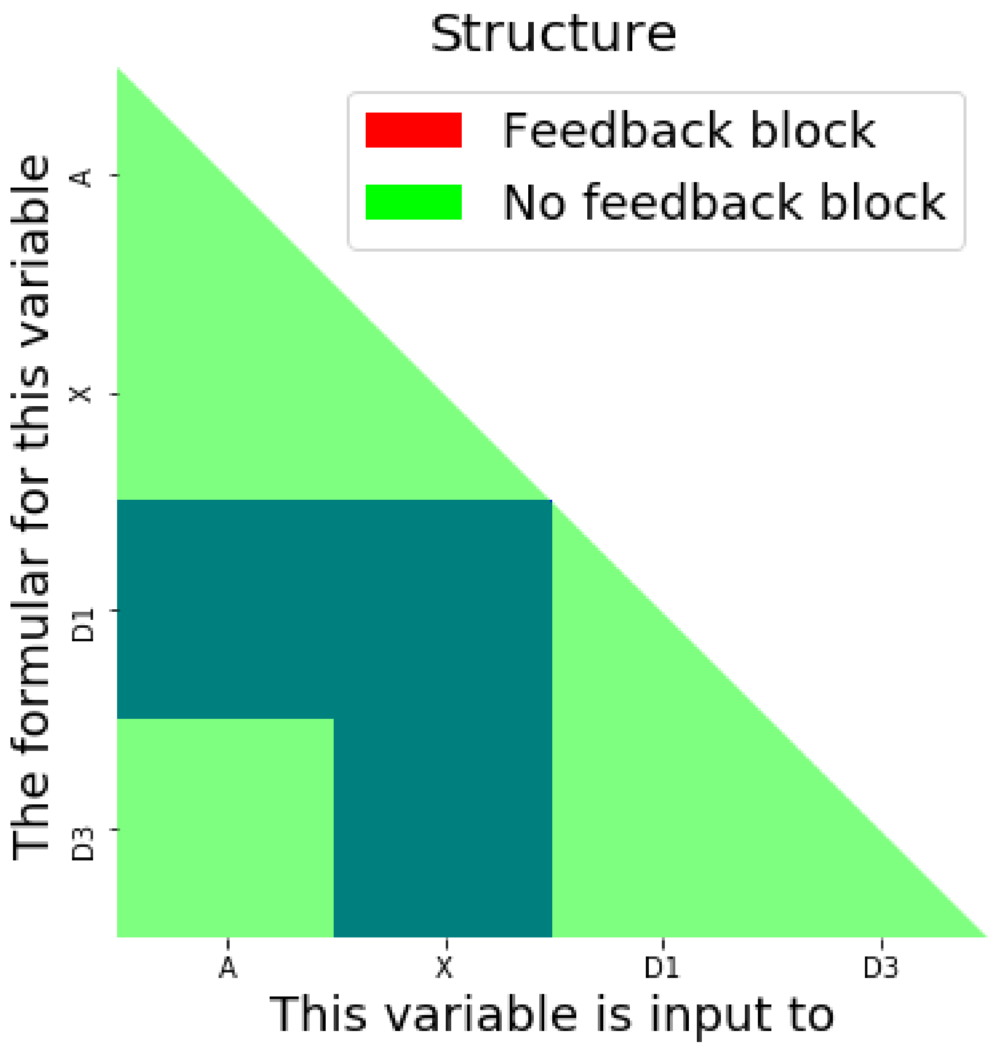


3.2.6 And the adjacency matrix of the graph

The graph can also be represented as a adjacency matrix. This is a is a square matrix A. $A_{i,j}$ is one when there is an edge from node i to node j, and zero when there is no edge.

If the graph is a DAG the adjacency matrix, and the elements are in a topological order, is a lover triangular matrix.

```
In [18]: a = mn.draw_adjacency_matrix(mmodel.endograph,mmodel.strongorder,mmodel.strongblock,
                                      mmodel.strongtype,size=(6,6))
```



3.2.7 A topological sorted list of contemporaneous endogenous variables (the solve order)

Also a property of the model

```
In [19]: mmodel.topo
```

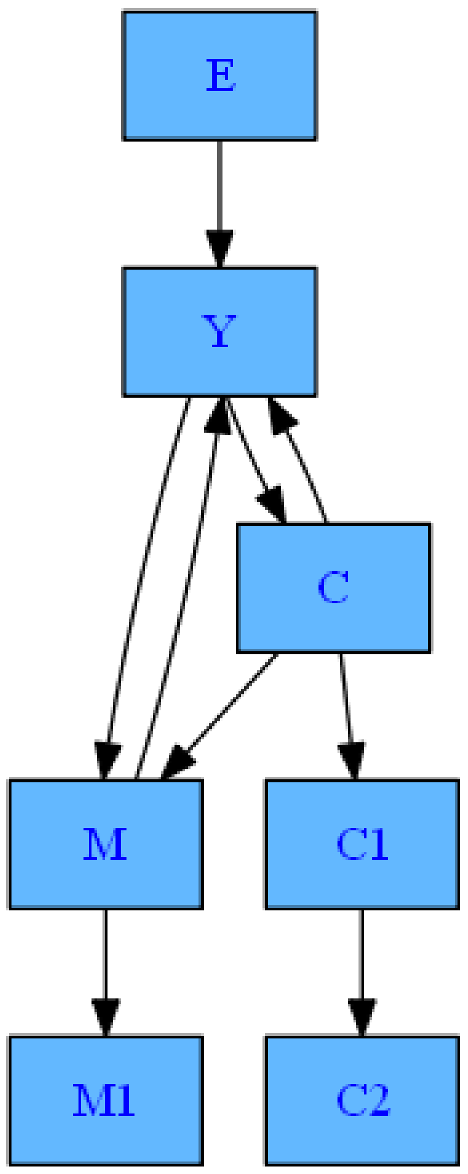
```
Out[19]: ['A', 'X', 'D1', 'D3']
```

3.2.8 The dependency graph for contemporaneous endogenous variables, with feedback

Now lets look at another small model. This time with feedback.

```
In [20]: fms = '''
         FRML xx Y  = C+I+E-M $
         FRML xx C  = 0.9*Y+Y0 $
         FRML xx C1 = 0.2*C $
         FRML xx C2 = 0.2*C1 $
         FRML xx M  = 0.1*Y + 0.2 * c $
         FRML xx M1 = 0.2*M $
         FRML xx E  = 0.1*E0 $
         '''
         ms = mc.model(fms)   # make a model instance
         ms.drawendo()
```

```
This model has simultaneous elements or cyclical elements. The formulars will be evaluated in input seq
```



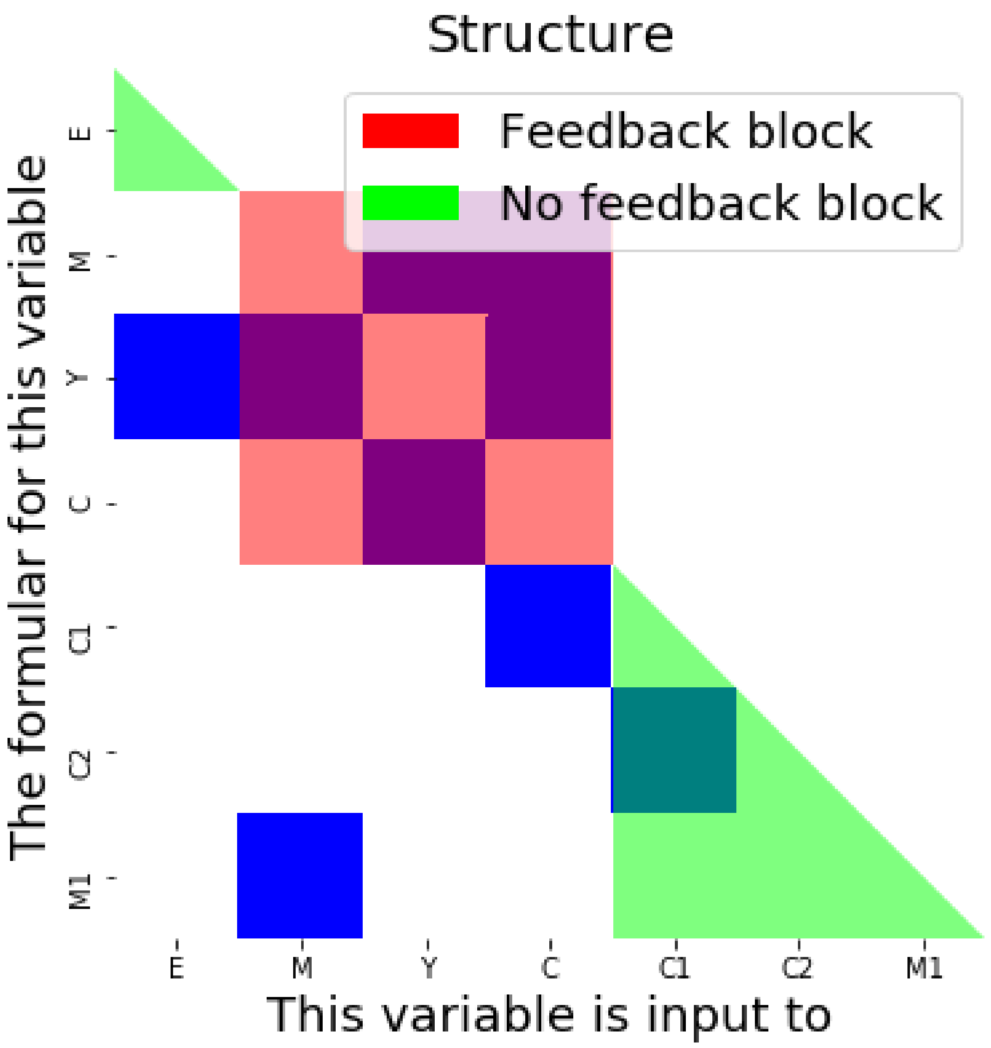
A_model_graph

3.2.9 And the adjacency matrix of the graph

When the model has contemporaneous feedback the TE will not be a DAG. There will be strong components with more than 1 node. This mean that the adjacency matix will have elements above the diagonal - like below.

A graph where each strong component is condensed to a node will be a DAG. This can be calculated by ModelFlow. And the user can choose this as solving order. Else the model will be solved in the order in which the model is specified.

```
In [21]: a = mn.draw_adjacency_matrix(ms.endograph,ms.strongorder,ms.strongblock,ms.strongtype,
                                     size=(6.,6))
```



```
In [22]: # the strong components of the model, DAG's (recursive elements) are condensed
         print(ms.strongblock)
         print(ms.strongtype)

[['E'], ['M', 'Y', 'C'], ['C1', 'C2', 'M1']]
['Recursiv', 'Simultaneous', 'Recursiv']
```

3.3 Model solution

So much about the formulas and the structure. We want a solution.

If we have a model:

$$\mathbf{y}_t \quad = \quad \mathbf{F}(\mathbf{y}_t \cdots \mathbf{y}_{t-r}, \mathbf{x}_t \cdots \mathbf{x}_{t-r}) \tag{3.9}$$

and the derived graph: $TE = (E, T_e)$

Then we have a solution, if we can find $\mathbf{y}_t^*$ so that:

$$\mathbf{y}_t^* \quad = \quad \mathbf{F}(\mathbf{y}_t^* \cdots \mathbf{y}_{t-r}, \mathbf{x}_t \cdots \mathbf{x}_{t-r}) \tag{3.10}$$

3.3.1 The simple model

If TE is a Directed Acyclical Graph (A DAG), F has no contemporaneous feedback, and is particular easy to solve.

The equations has to be evaluated in a logical (topological sorted) order. As mentioned, the NetworkX library is used to perform this task on TE.

3.3.2 The simultaneous model

If TE is not a Directed Acyclical Graph (A DAG) F has contemporaneous feedback, and has to be solved with an iterative method. The candidates are algorithms of the **Newton** or the **Gauss** family.

For large sparse nonlinear models Gauss-Seidel works fine. It solves a model quite fast and we don't need the additional handiwork of handling derivatives and invert large matrices that Newton methods require. Moreover many models in question do not have smooth derivatives.

Therefor ModelFlow uses a Gauss-Seidel solve the a model.

The Gauss-Seidel algorithm

The Gauss-Seidel algorithm is quite straight forward. It basically iterate over the formulas, until convergence.

Let z be all predetermined values: all exogenous variable and lagged endogenous variable. Let n be the number of endogenous variables.

For each time period we can find a solution by doing Gauss-Seidel iterations:

```
for k = 1 to convergence
..for $i$ = 1 to $n$
....$y_i^k = (1-\alpha) * y_i^{k-1} + \alpha f_i(y_1^k, \cdots, y_{i-1}^k, y_{i+1}^{k-1}, \cdots, y_n^{k-1}, z)$
```

Convergence has been reached when the solution don't change to much between each iteration. There is no need to check all variable for convergence. So the user can define a set {convergence variable} Then all relative changes from one iteration to has to be smaller than a convergence criteria $\epsilon_{relative}$

$\forall y_j$ where $y_j \in$ {convergence variable}$\wedge \mid y_j^{k-1} \mid \geq \epsilon_{absolute}$ we calculate $\mid \frac{y_j^k - y_j^{k-1}}{y_j^{k-1}} \mid \leq \epsilon_{relative}$

Under some circumstances the the iterations will diverge, even if there is a solution. This can (mostly) be handled by introducing the dampening factor α in the algorithm.

One challenge with Gauss-Seidel can be that the evaluation order of the formulas influence the number of iterations. As default ModelFlow will use the order in which the model is specified. If the feedback variables are easy to identify. They should be placed in the end of the model.

However the user can just set the property `model.solveorder` for alternative orders. One could be the solveorder of the strong elements. `model.strongorder`.

3.3.3 Solving in ModelFlow

As we have seen an instance of a model can be solved just by calling the model instance.

In the most plain vanilla call, the class will determine if the model is simultaneous or not and use the appropriate solver. Also the model will be solved for the max possible time span.

The user can use a number of arguments to control the solution in more detail - time, convergence, dump of iterations, saving of results in the instance and so on. For this look at the API documentation.

Speeding up solving through Just In Time compilation (Numba)

Python is an interpreted language. Therefor the calculation speed can be improved. There are several avenues to speed up calculations.

Firstly, use the matrix extensions if possible. This will force the use of the highly optimized routines in the Numpy library.

Secondly, python code can be compiled. The most straight forward way is to use the **Numba** library. For simultaneous models which only operates on numerical data the user can specify jit=1. This will engage the numba Just In Time Compiler and the model will solve significantly faster. However the compilation will take some time and for large models Numba will choke on the model. Experience with a Danish model (1700 equations) shows a speedup from 5 million floating point operations per second (MFlops) to 800 MFlops.

Also experiments with the **Cython** library has been performed. This library will translate the Python code to C++ code. Then a C++ compiler can compile the code and the run time will be improved a lot.

Chapter 4

From model equations to solution

The core of ModelFlow is a transpiler. The purpose of the transpiler is to convert a model in the form of a string of formulas to a python class instance of the ***model*** class. Combining this instance with data in the form of a Pandas DataFrame and we have a solution.

Python come batteries included, so a number of powerful tools are available. Therefore, it is not necessary to design a full-fledged compiler/interpreter in order to solve a model. Some of the tough jobs are outsourced to the Python ecosystem.

When given as input to the model class a model flows through several phases:

4.1 A syntax checker.

Formulas must be legal Python expressions. This allows us to outsource syntax checking of formulas to the Python ecosystem. The ***ast*** Python standard library can check the syntax, and it is easy to make a function which catch syntax errors in the formulas.

```
In [23]: mp.check_syntax_frml('frml <> a = b $')
```

```
Out[23]: True
```

```
In [24]: mp.check_syntax_frml('frml <> a = b + = c$')
```

```
Out[24]: False
```

4.2 A tokenizer.

The model is chopped into tokens. This is done by using the regular expressing (***re***) Python standard library. The tokenizer delivers an expression ad a list of tokens in the form of named tuple's (terms). Each token can be either of four types:

- A variable perhaps with lags
- A number
- An operator, function.
- A comment

```
In [25]: pt.udtryk_parse('a=b(-1)+3')
```

```
Out[25]: [nterm(comment='', number='', op='', var='A', lag=''),
          nterm(comment='', number='', op='=', var='', lag=''),
          nterm(comment='', number='', op='', var='B', lag='-1'),
          nterm(comment='', number='', op='+', var='', lag=''),
          nterm(comment='', number='3', op='', var='', lag='')]
```

And a model which consists of several formulas is parsed to a structure like this

```
In [26]: pt.model_parse('frml <> a = b+2 $ frml <> v = log(x) $')
```

```
Out[26]: [(fatoms(whole='FRML <> A = B+2 $', frml='FRML', frmlname='<>', expression='A = B+2 $'),
            [nterm(comment='', number='', op='', var='A', lag=''),
             nterm(comment='', number='', op='=', var='', lag=''),
             nterm(comment='', number='', op='', var='B', lag=''),
             nterm(comment='', number='', op='+', var='', lag=''),
             nterm(comment='', number='2', op='', var='', lag=''),
             nterm(comment='', number='', op='$', var='', lag='')]),
           (fatoms(whole='FRML <> V = LOG(X) $', frml='FRML', frmlname='<>', expression='V = LOG(X) $'),
            [nterm(comment='', number='', op='', var='V', lag=''),
             nterm(comment='', number='', op='=', var='', lag=''),
             nterm(comment='', number='', op='LOG', var='', lag=''),
             nterm(comment='', number='', op='(', var='', lag=''),
             nterm(comment='', number='', op='', var='X', lag=''),
             nterm(comment='', number='', op=')', var='', lag=''),
             nterm(comment='', number='', op='$', var='', lag='')])]
```

The model_parse function takes a model and returns a list of named tuples, Each tuple contains: - the complete formula - FRML - Formula name - The expression - A list of terms from the expression

Care has been taken to optimize the speed of the tokenizing function. By tokenizing the whole model in one python list comprehension, even quite large models (number of formulas > 600,000) can be tokenized in a few seconds.

4.3 An analyzer

This function extract information needed to handle the model such as.

- Endogenous and exogenous variables
- Starting positions in calculating vector.
- The dependency graphs TG and TE.

These objects are stored as property in the class instance. Some of the properties like TG and TG can be time consuming to calculate. In these cases lazy evaluation is used. This means that computing of the properties is delayed until the property is requested (perhaps implicit) by the user.

4.4 Code generation

This phase converts the tokenized model to python code, which can solve the model.

The purpose is to substitute variable names in the the model formulas to an access to a DataFrame.

It is quite straight forward to write a function, which generates solving routines in other languages than Python.

The code generation phase produces a factory function, which will return the actual calculating function. This methods is chosen in order to manage the namespace in which the calculation function is operating.

In order to catch calculation errors, the code performing the calculation is braced by a error catching clause. Thus if numerical error such as taking the log of a negative number raises an error, we can identify the offending formula.

For our little model the factory function looks like this:

```
In [27]: print(mmodel.make_los_text)
```

```
def make_los(funks=[]):
    from modeluserfunk import pd_to_w, pd_to_w_corp, pd_to_w_mrtg, pd_to_w_retail, phi, phiinv
    from modelBLfunk import array, classfunk, exp, inspect, log, logit, matrix, mv_opt, mv_opt_prop, no
    def los(values,row,solveorder, allvar):
        try :
            values[row,6]=values[row-1,1]+values[row,0]
            values[row,3]=0.5*values[row,1]
            values[row,5]=values[row,3]+3*values[row-1,6]+values[row,1]**2+values[row,6]
            values[row,4]=values[row,3]+3*values[row-1,6]+values[row,1]**3
        except :
            print("Error in",allvar[solveorder[sys.exc_info()[2].tb_lineno-6]]["frml"])
            raise
        return
    return los
```

4.4.1 Optimizing access to DataFrame values

Execution speed is crucial. Values in a DataFrame can be accessed in a number of ways. Some are fast, others are not. Therefore, some care has been exercised when accessing each value.

If we have a DataFrame df and a column with the name 'FY', the value in 2018 can be accessed like this: df.loc[2018,'FY'], This works fine for small models. However it is quite slow as the DataFrame has to look up the position (row,column) of 2018 and 'FY' before it can access the value in an underlying structure. When looking for the value for 2019 the Dataframe again has to to the lookup.

When calculating simple (non simultaneous) models, the row and column of each value is resolved before the code generation and the value is accessed directly in an underlying property of dataframes called values. So if FY is the 42 column of the DataFrame and the position of 2018 is assigned to the variable row. The value is accessed as df.values[row,42]. In this case no time is wasted for looking up the row and column of the position.

When using the iterative Gauss-Seidel solution method, all necessary values are transferred to a one-dimensional numpy calculation array - called a. So the value would just be accessed like: a[84]. This requires some overhead to stuff and unstuff the values from df. But data access is faster, as accessing elements in a one-dimensional array is faster than in a two-dimensional array. Also, the locality of data is increased, thus the chance of a cache miss is reduced. This is especially important if the model is compiled.

Chapter 5

Some Model manipulation capabilities

5.1 Model inversion aka Target/instruments or Goal Seek

In ordet to answer questions like:

- How much capital has to be injected in order to maintain a certain GDP level in a stressed scenario?
- How much loans has to be shredded by the banks in order to maintain a minimum level of capital (slim to fit)?
- How much capital has to be injected in order to keep all bank above a certain capital threshold ?

The model instance is capable to **"invert"** a model. To use the terminology of Tinbergen(1955) that is to calculate the value of some exogenous variables - **the instruments** which is required in order to achieve a certain target value for some endogenous variables - **the targets**.

To use the terminology of Excel it is a goal/seek functionality with multiple cells as goals and multiple cells as targets.

The problem can be thought as follows: From the generic description of a model above, we can think of the model like this: $\mathbf{y}_t = \mathbf{F}(\mathbf{x}_t)$. Here $\mathbf{x}_t$ are all predetermined variables - lagged endogenous and all exogenous variables.

Think of a condensed model ($\mathbf{G}$) with a few endogenous variables($\bar{\mathbf{y}}_t$): the targets and a few exogenous variables($\bar{\mathbf{x}}_t$): the instrument variables. All the rest of the predetermined variables are fixed:
$\bar{\mathbf{y}}_t = \mathbf{G}(\bar{\mathbf{x}}_t)$

If we invert G we have a model where instruments are functions of targets: $\bar{\mathbf{x}}_t = \mathbf{G}^{-1}(\bar{\mathbf{y}}_t)$. Then all we have to do is to find $\mathbf{G}^{-1}(\bar{\mathbf{y}}_t)$

5.1.1 And how to solve for the instruments

For most models $\bar{\mathbf{x}}_t = \mathbf{G}^{-1}(\bar{\mathbf{y}}_t)$ do not have a nice close form solution. However it can be solved numerically. This time we will leave the Gauss and turn to Newton.

$\bar{\mathbf{x}}_t = \mathbf{G}^{-1}(\bar{\mathbf{y}}_t^*)$ can be found using Newton–Raphson method:
for k = 1 to convergence
..$\bar{\mathbf{x}}_t{}^k = \bar{\mathbf{x}}_t{}^{k-1} + \mathbf{J}_t^{-1}(\bar{\mathbf{y}}_t^* - \bar{\mathbf{y}}_t{}^{k-1})$
..$\bar{\mathbf{y}}_t{}^k = \mathbf{G}(\bar{\mathbf{x}}_t{}^k)$
convergence: $\mid \bar{\mathbf{y}}_t^* - \bar{\mathbf{y}}_t \mid \leq \epsilon$
Now we just need to find:

$\mathbf{J}_t = \frac{\partial \mathbf{G}}{\partial \mathbf{\bar{x}}_t}$

A number of differentiation methods can be used (symbolic, automated or numerical). ModelFlow uses numerical differentiation, as it is quite simple and fast.

$\mathbf{J}_t \approx \frac{\Delta \mathbf{G}}{\Delta \mathbf{\bar{x}}_t}$

That means that we should run the model one time for each instrument, and record the effect on each of the targets, then we have $\mathbf{J}_t$

In order for $\mathbf{J}_t$ to be invertible there has to be the same number of targets and instruments.

However, each instrument can be a basket of exogenous variable. They will be adjusted in fixed proportions. This can be useful for instance when using bank leverage as instruments. Then the leverage instrument can consist of several loan types.

Now lets look at the toy model and a simple example of model inversion.

5.1.2 An example

The workflow is as follow:

1. Define the targets
2. Define the instruments
3. Create a target_instrument class istance
4. Solve the problem

Step one is to define the targets. This is done by creating a dataframe where the target values are set. In this case it can be done like this.

```
In [28]: target = mmodel.basedf.loc[2020:,['D3','D1']]+[1,10]
         target
```

```
Out[28]:          D3    D1
         2020   35.5  29.5
         2021  229.0  62.0
```

Then we have to provide the instruments. This is **a list of list of tuples**. - Each element in the outer list is an instrument. - Each element in the inner list is and instrument variable - Each element of the tuple contains a variable name and the associated impulse Δ.

The $\Delta variable$ is used in the numerical differentiation. Also if one instrument contains several variables, the proportion of each variable will be determined by the relative $\Delta variable$.

For this experiment the inner list only contains one variable.

```
In [29]: instruments = [ [('C',1)] , [('B',1)]]
```

Now an instance of the ModelFlow class:target_instrument is created.

```
In [30]: ti = mi.targets_instruments(target,instruments,mmodel,silent=True)
```

Then it is called, and we can display the result

For models which are relative linear we don't need to update $\mathbf{J}_t$ for each iteration and time frame. As our small toy model is nonlinear, the jacobi matrix has to be updated frequently. This is controlled by the **nonlin=True** option below.

```
In [31]: result       = ti(nonlin=True, maxiter=100,silent=True)
         result

Out[31]:               B         C    E         X          D3         D1          A
         2018  1.000000  4.000000  4.0  0.000000    0.000000   0.000000   0.000000
         2019  2.000000  5.000000  4.0  2.500000  127.500000  33.500000   6.000000
         2020 -1.179527  2.532568  4.0  1.266284   35.509926  29.500659   3.820473
         2021  9.173873  5.986590  4.0  2.993295  229.009678  62.000416  11.706441
```

And do the result match the target?

```
In [32]: print(result-target)

        A    B    C        D1        D3    E    X
2018 NaN  NaN  NaN       NaN       NaN  NaN  NaN
2019 NaN  NaN  NaN       NaN       NaN  NaN  NaN
2020 NaN  NaN  NaN  0.000659  0.009926  NaN  NaN
2021 NaN  NaN  NaN  0.000416  0.009678  NaN  NaN
```

So we got results for the target variable very close to the target values.

5.1.3 Shortfall targets

Above the target for each target variable is a certain values. Sometime we we need targets being above a certain shortfall value. In this case an instrument should be used to make the achieve the target threshold only if the target is belove the target. This is activated by an option:**shortfall=True**.

This feature can be useful calculating the amount of deleverage needed for banks to achieve a certain threshold of capital.

5.2 Attribution / Explanation

Experience shows that it is useful to be able to explain the difference between the result from two runs. The first level of understanding the difference is to look at selected formulas and find out, how much each input variables accounts for. The second level of understanding the difference is to look at the attribution of the exogenous variables to the results of the model.

If we have:

$y = f(a, b)$

and we have two solutions where the variables differs by $\Delta y, \Delta a, \Delta b$

How much of Δy can be explained by Δa and Δb ?

Analytical the attributions Ωa and Ωb can be calculated like this:

$$\Delta y = \underbrace{\Delta a \frac{\partial f}{\partial a}(a, b)}_{\Omega a} + \underbrace{\Delta b \frac{\partial f}{\partial b}(a, b)}_{\Omega b} + Residual$$

ModelFlow will do a numerical approximation of Ωa and Ωb. This is done by looking at the two runs of the model:

$$y_0 = f(a_0, b_0) \tag{5.1}$$
$$y_1 = f(a_0 + \Delta a, b_0 + \Delta b) \tag{5.2}$$

So Ωa and Ωb can be determined:

$$\Omega f_a = f(a_1, b_1) - f(a_1 - \Delta a, b_1) \tag{5.3}$$
$$\Omega f_b = f(a_1, b_1) - f(a_1, b_1 - \Delta b) \tag{5.4}$$

And:

$$residual = \Omega f_a + \Omega f_b - (y_1 - y_0) \tag{5.5}$$

If the model is fairly linear, the residual will be small.

5.2.1 Formula attribution

Attribution analysis on the formula level is performed by the method **.dekomp**.

This method utilizes that two attributes .basedf and .lastdf containing the first and the last run are contained in the model instance. Also all the formulas are contained in the instance. Therefore a model, just with one formula - is created. Then experiments mentioned above is run for each period and each right hand side variable.

```
In [33]: _ = mmodel.dekomp('D1')

Formula        : FRML <> D1 = X + 3 * A(-1)+ C **2 +A  $

                      2019       2020       2021
Variable    lag
Base        0     7.000000  19.500000  52.000000
Alternative 0    33.500000  29.500659  62.000416
Difference  0    26.500000  10.000659  10.000416
Percent     0   378.571422  51.285427  19.231569

 Contributions to differende for  D1
                   2019       2020       2021
Variable lag
X          0    1.500000  -0.233716  -0.006705
A         -1    0.000000  12.000000   2.461419
C          0   21.000000  -2.586099  -0.160739
A          0    4.000000   0.820473   7.706441

 Share of contributions to differende for  D1
                    2019        2020        2021
Variable lag
```

```
A          0          15%          8%          77%
          -1           0%        120%          25%
X          0           6%         -2%          -0%
C          0          79%        -26%          -2%
Total      0         100%        100%         100%
Residual   0          -0%         -0%          -0%
```

If the baseline dataframe is set to the lagged variable of the last dataframe. We can calculate the attributions to the change of results from time period to time period.

5.2.2 Walk the explanations.

Sometime we want to trace the attributions to variables in higher in the dependency graph.

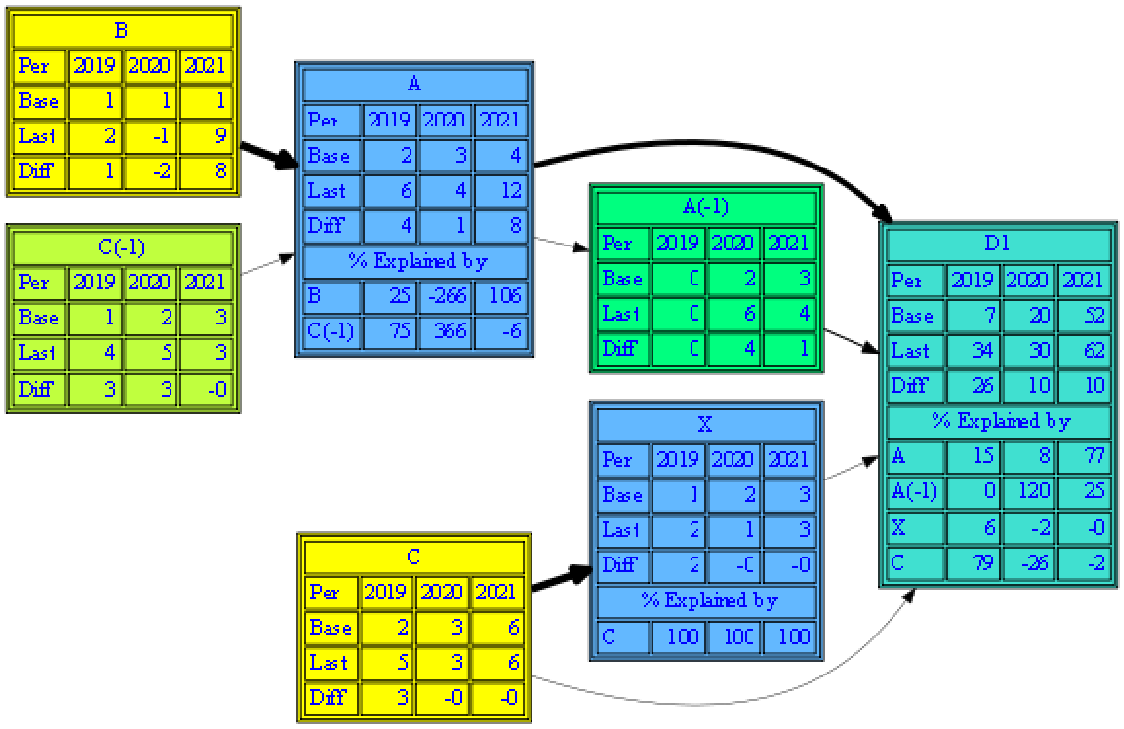
Using the dependency graph TG, we can walk up the graph, and do an attribution calculation for each of the parents, grand parents and so on.

It should be mentioned that for *lagged* endogenous variable an attribution calculation will not be performed.

So we use the **.explain** method.

The result will also be made into a vector graphic both in .pdf and .svg format. This allows for zooming into the tree, even if there are many leaves.

```
In [34]: _ = mmodel.D1.explain(pdf=1,up=2)
```



B

Per	2019	2020	2021
Base	1	1	1
Last	2	-1	9
Diff	1	-2	8

C(-1)

Per	2019	2020	2021
Base	1	2	3
Last	4	5	3
Diff	3	3	-0

A

Per	2019	2020	2021
Base	2	3	4
Last	6	4	12
Diff	4	1	8
% Explained by			
B	25	-266	106
C(-1)	75	366	-6

A(-1)

Per	2019	2020	2021
Base	0	2	3
Last	0	6	4
Diff	0	4	1

X

Per	2019	2020	2021
Base	1	2	3
Last	2	1	3
Diff	2	-0	-0
% Explained by			
C	100	100	100

C

Per	2019	2020	2021
Base	2	3	6
Last	5	3	6
Diff	3	-0	-0

D1

Per	2019	2020	2021
Base	7	20	52
Last	34	30	62
Diff	26	10	10
% Explained by			
A	15	8	77
A(-1)	0	120	25
X	6	-2	-0
C	79	-26	-2

D1 explained

5.2.3 Model Attribution

At the model level we start by finding which exogenous variables have changed between two runs.

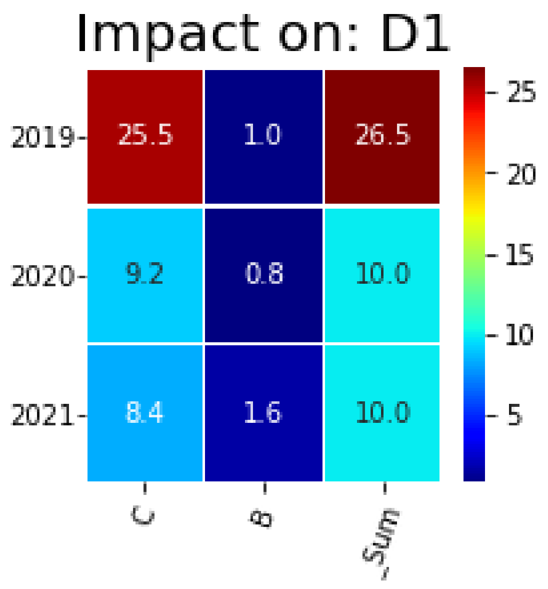
```
In [35]: difdf = mmodel.exodif()
         difdf

Out[35]:               B         C
         2018  0.000000  3.000000
         2019  1.000000  3.000000
         2020 -2.179527 -0.467432
         2021  8.173873 -0.013410
```

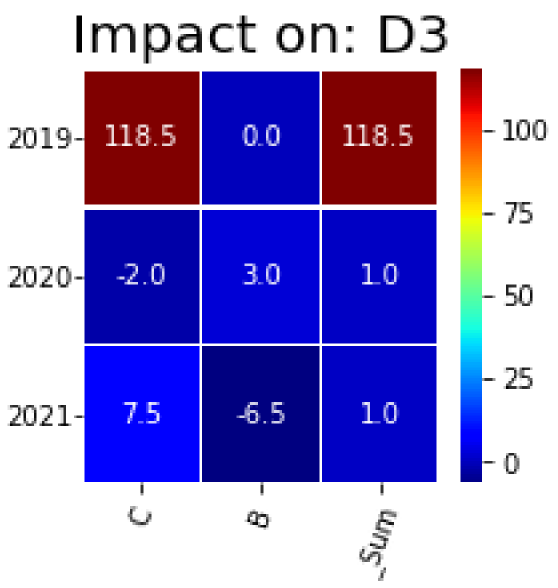
Now we make a dictionary of experiments where the key is the name of the experiment and the value of each entry is the variable to investigate in the experiment

```
In [36]: experiments = {e : [e]  for e in difdf.columns}
         experiments

Out[36]: {'B': ['B'], 'C': ['C']}
```

Then we can let the model run through each experiment and record a number of summary variable.

```
In [37]: impact = md.attribution(mmodel,experiments,start=2019,end=2021,summaryvar=['D*'])
         impact

0 Experiment : B
 Touching:
 ['B']
1 Experiment : C
 Touching:
 ['C']


Out[37]:       B                            C
            2019      2020      2021    2019      2020      2021
         D1  1.0  0.820473  1.635292    25.5  9.180185  8.365124
         D3  0.0  3.000000 -6.538581   118.5 -1.990074  7.548258

In [38]: impact['C',2021]

Out[38]: D1    8.365124
         D3    7.548258
         Name: (C, 2021), dtype: float64

In [39]: for sumaryvar in impact.index:
             mv.attshow(md.GetAllImpact(impact, sumaryvar),tshow=0,t=False,annot=True,dec=1,
                        title='Impact on: '+sumaryvar,size=(3,3),showsum=True,head=1,tail=1)
```





When the results are displayed, they can be filtered, sliced and diced in a number of ways.

For a model of the EBA stress test the number of changed exogenous variable can be large. Using a dictionary to contain the experiments allows us to create experiments where all variable for each country are analyzed, or each macro variable for all countries are analyzed.

Also it is possible to use aggregated sums - useful for looking at impact on PD's. Or just the last time period - useful for looking at CET1 ratios.

If there are many experiments, data can be filtered in order to look only at the variables with an impact above a certain threshold.

The is also the possibility to anonymize the row and column names and to randomize the order of rows and/or columns - useful for bank names.

5.3 Optimization

Using the convex optimization package CVXOPT, which is part of the Anaconda Python distribution a range of optimization problems can be solved. One example is the standard mean variance problem.

5.3.1 A mean variance problem

If we look at a fairly general mean variance optimization problem which has been adopted to banks it looks like this:

$$\begin{aligned}
\mathbf{x} &\quad \text{Position in each asset(+)/liability(-) type} && (5.6)\\
\mathbf{x} &\quad \text{Position in each asset(+)/liability(-) type} && (5.7)\\
\boldsymbol{\Sigma} &\quad \text{Covariance matrix} && (5.8)\\
\mathbf{r} &\quad \text{Return vector} && (5.9)\\
\lambda &\quad \text{Risk aversion} && (5.10)\\
\mathbf{riskweights} &\quad \text{Vector of risk weights, liabilities has riskweight = 0} && (5.11)\\
\mathit{Capital} &\quad \text{Max of sum of risk weighted assets} && (5.12)\\
\mathbf{lcrweights} &\quad \text{Vector of LCR weights, liabilities has lcrweight = 0} && (5.13)\\
\mathit{LCR} &\quad \text{Min of sum of lcr weighted assets} && (5.14)\\
\mathbf{leverageweight} &\quad \text{Vector of leverage weights, liabilities has leverageweight = 0} && (5.15)\\
\mathit{Equity} &\quad \text{Max sum of leverage weighted positions} && (5.16)\\
\mathit{Budget} &\quad \text{initial sum of the positions} && (5.17)\\
& && (5.18)
\end{aligned}$$

$$\begin{aligned}
\text{minimize:} &\quad \lambda \mathbf{x}^T \boldsymbol{\Sigma} \mathbf{x} - (1-\lambda)\mathbf{r}^T\mathbf{x} &&\text{If } \lambda = 1 \text{ minimize risk, if } \lambda = 0 \text{ maximize return} && (5.19)\\
\text{subject to:} &\quad \mathbf{x} \succeq \mathbf{x^{min}} && \text{Minimum positions} && (5.20)\\
&\quad \mathbf{x} \preceq \mathbf{x^{max}} && \text{Maximum positions} && (5.21)\\
&\quad \mathbf{riskweights}^T\mathbf{x} \le \mathit{Capital} && \text{Risk weighted assets <= capital} && (5.22)\\
&\quad \mathbf{lcrweights}^T\mathbf{x} \ge \mathit{LCR} && \text{lcr weighted assets >= LCR target} && (5.23)\\
&\quad \mathbf{leverageweight}^T\mathbf{x} \le \mathit{equity} && \text{leverage weighted assets <= equity} && (5.24)\\
&\quad \mathbf{1}^T\mathbf{x} = \mathit{Budget} && \text{Sum of positions = B} && (5.25)
\end{aligned}$$

5.3.2 The mean variance problem in the business language language

In the business logic language this problem can be specified like this:

```
positions =  mv_opt(msigma,return,riskaversion, budget,
            [[risk_weights] , [-lcr_weights] , [leverage_weights]],
              [capital, -lcr , equity] ,min_position,max_position)
```

Where the arguments are appropriately dimensioned CVX matrices and vectors.

For a more elaborate example there is an special notebook on the subject of optimization.

Also it should be mentioned that there is an expansion of the basic problem taking transaction cost into account.

5.4 Symbolic model differentiation

ModelFlow have the capability to calculate expressions for differentiation of the formulas of a model. This is done by the Python symbolic math library **Sympy**. After finding the expressions for the differential coefficients they can be evaluated - just like the model. Based on these a linearized version of the model can be constructed.

We want to be able to find he Jacobi matrices:

$$\mathbf{A} = \frac{\partial \mathbf{F}}{\partial \mathbf{y}_t^T} \tag{5.26}$$

$$\tag{5.27}$$

$$\mathbf{E}_i = \frac{\partial \mathbf{F}}{\partial \mathbf{y}_{t-i}^T} \quad i = 1, \cdots, s \tag{5.28}$$

$$\tag{5.29}$$

$$\mathbf{F}_j = \frac{\partial \mathbf{F}}{\partial \mathbf{x}_{t-j}^T} \quad j = 0, \cdots, r \tag{5.30}$$

These matrices can be useful when finding stability properties.

Now lets differentiate our toy model

```
In [40]: mdif.modeldiff(mmodel)
```

```
Model                          : My first model
Number of variables            : 6
Number of endogeneus variables : 4
Number of derivatives          : 10
```

Now we can display the expressions for the differential coefficient - and the values in a selected time (row), in this case 2018.

```
In [43]: mdif.display_all(mmodel,mmodel.basedf,2018)
```

$$Formula : A = C_{-1} + B$$

$$\begin{aligned} \frac{\partial A}{\partial B} &= 1 \\ &= 1 \\ \frac{\partial A}{\partial C_{-1}} &= 1 \\ &= 1 \end{aligned}$$

$$
\begin{aligned}
Formula: D1 &= X + 3 * A_{-1} + C**2 + A \\
\\
\frac{\partial D1}{\partial A} &= 1 \\
&= 1 \\
\frac{\partial D1}{\partial A_{-1}} &= 3 \\
&= 3 \\
\frac{\partial D1}{\partial C} &= 2 * C \\
&= 2.0 \\
\frac{\partial D1}{\partial X} &= 1 \\
&= 1
\end{aligned}
$$

$$
\begin{aligned}
Formula: D3 &= X + 3 * A_{-1} + C**3 \\
\\
\frac{\partial D3}{\partial A_{-1}} &= 3 \\
&= 3 \\
\frac{\partial D3}{\partial C} &= 3 * C**2 \\
&= 3.0 \\
\frac{\partial D3}{\partial X} &= 1 \\
&= 1
\end{aligned}
$$

$$
\begin{aligned}
Formula: X &= 0.5 * C \\
\\
\frac{\partial X}{\partial C} &= 0.500000000000000 \\
&= 0.5
\end{aligned}
$$

Also the matrices of differential coefficient (Jacobi matrices) for each lag, can be calculated

```
In [44]: # calculate
         matdir = mdif.calculate_allmat(mmodel,mmodel.basedf,2018)
         # Display
         for l,m in matdir.items():
             print(f'Lag:{l} \n {m} \n')
```

```
Finding derivatives started at :         10:40:27
Finding derivatives took       :    0.0099999905 Seconds
Calculating derivatives started at :         10:40:27
Calculating derivatives took       :    0.0075004101 Seconds
creating dataframe started at :         10:40:27
creating dataframe took       :    0.0000000000 Seconds
Lag:0
     A  D1  D3  X  B    C
A    0   0   0  0  1  0.0
D1   1   0   0  1  0  2.0
D3   0   0   0  1  0  3.0
X    0   0   0  0  0  0.5

Lag:-1
     A  D1  D3  X  B  C
A    0   0   0  0  0  1
D1   3   0   0  0  0  0
D3   3   0   0  0  0  0
X    0   0   0  0  0  0
```

We can carve out the matrices only for the endogenous variables. The A and E matrices mentioned above can be extracted like this:

```
In [45]: for l,m in matdir.items():
             endomat = m.loc[:,[c for c in m.columns if c in mmodel.endogene]]
             print(f'Lag:{l} \n {endomat} \n')
```

```
Lag:0
     A  D1  D3  X
A    0   0   0  0
D1   1   0   0  1
D3   0   0   0  1
X    0   0   0  0

Lag:-1
     A  D1  D3  X
A    0   0   0  0
D1   3   0   0  0
D3   3   0   0  0
X    0   0   0  0
```

A word of caution.

Symbolic math can take time if the model is large. So it is advisable to test this feature on a small model which captures the essence before moving to a large model.

Also the Sympy library will only handle functions and expressions that it knows.

5.5 Stability

Jacobi matrices can be used to evaluate the stability properties of the model. To do this we first look at a linearized version of the model. We are interested in the effect of shocks to the system. Will shocks be damped or will they be amplified.

To calculate the effect of small perturbations the model can be linearized around a solution

$$\begin{aligned}
\Delta y_t &= A\Delta y_t + E_1\Delta y_{t-1} + \cdots + E_r\Delta y_{t-r} + F_0\Delta x_t + \cdots + F_s\Delta x_{t-s} \\
I\Delta y_t - A\Delta y_t &= E_1\Delta y_{t-1} + \cdots + E_r\Delta y_{t-r} + F_0\Delta x_t + \cdots + F_s\Delta x_{t-s} \\
(I-A)\Delta y_t &= E_1\Delta y_{t-1} + \cdots + E_r\Delta y_{t-r} + F_0\Delta x_t + \cdots + F_s\Delta x_{t-s} \\
\Delta y_t &= (I-A)^{-1}(E_1\Delta y_{t-1} + \cdots + E_r\Delta y_{t-r} + F_0\Delta x_t + \cdots + F_s\Delta x_{t-s}) \\
\Delta y_t &= (I-A)^{-1}E_1\Delta y_{t-1} + \cdots + (I-A)^{-1}E_r\Delta y_{t-r} + (I-A)^{-1}F_0\Delta x_t + \cdots + (I-A)^{-1}F_s\Delta x_{t-s}
\end{aligned}$$

$$\begin{aligned}
y_t &= F(y_t \cdots y_{t-r}, x_t \cdots x_{t-s}) \\
\Delta y_t &= (I-A)^{-1}E_1\Delta y_{t-1} + \cdots + (I-A)^{-1}E_r\Delta y_{t-r} + (I-A)^{-1}F_0\Delta x_t + \cdots + (I-A)^{-1}F_s\Delta x_{t-s}
\end{aligned}$$

where:

$$\begin{aligned}
A &= \frac{\partial F}{\partial y_t^T} \\
E_i &= \frac{\partial F}{\partial y_{t-i}^T} \qquad i = 1, \cdots, r \\
F_j &= \frac{\partial F}{\partial x_{t-j}^T} \qquad j = 0, \cdots, s
\end{aligned}$$

We have

$$\begin{aligned}
y_t &= F(y_t \cdots y_{t-r}, x_t \cdots x_{t-s}) \\
\Delta y_t &= (I-A)^{-1}E_1\Delta y_{t-1} + \cdots + (I-A)^{-1}E_r\Delta y_{t-r} + (I-A)^{-1}F_0\Delta x_t + \cdots + (I-A)^{-1}F_s\Delta x_{t-s}
\end{aligned}$$

A rather messy problem. It can be made more simple. This is done by making a new linear model only incorporating variables with one lag.

This is done the standard way by introducing new variables: $y_t^1 = y_{t-1}$, $y_t^2 = y_{t-1}^1$, and $y_t^3 = y_{t-1}^2$ Using theese transformed variables, a system with max lag of 3 can e rewritten like this:

$$\underbrace{\begin{bmatrix}\Delta y_t^3 \\ \Delta y_t^2 \\ \Delta y_t^1 \\ \Delta y_t\end{bmatrix}}_{\Delta\bar{y}_t} = \underbrace{\begin{bmatrix}0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & A\end{bmatrix}}_{\bar{A}} \underbrace{\begin{bmatrix}\Delta y_t^3 \\ \Delta y_t^2 \\ \Delta y_t^1 \\ \Delta y_t\end{bmatrix}}_{\Delta\bar{y}_t} + \underbrace{\begin{bmatrix}0 & \mathbb{I} & 0 & 0 \\ 0 & 0 & \mathbb{I} & 0 \\ 0 & 0 & 0 & \mathbb{I} \\ E_3 & E_2 & E_1 & E\end{bmatrix}}_{\bar{E}} \underbrace{\begin{bmatrix}\Delta y_{t-1}^3 \\ \Delta y_{t-1}^2 \\ \Delta y_{t-1}^1 \\ \Delta y_{t-1}\end{bmatrix}}_{\Delta\bar{y}_{t-1}} + \underbrace{\begin{bmatrix}0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ F_3 & F_2 & F_1 & F_0\end{bmatrix}}_{\bar{F}} \underbrace{\begin{bmatrix}\Delta x_{t-3} \\ \Delta x_{t-2} \\ \Delta x_{t-1} \\ \Delta x_t\end{bmatrix}}_{\bar{x}_t}$$

So this expression:

$\Delta y_t = A\Delta y_t + E_1\Delta y_{t-1} + \cdots + E_r\Delta y_{t-r} + F_0\Delta x_t + \cdots + F_r\Delta x_{t-r}$

Has been simplified to this expression :

$\Delta\bar{y}_t = \bar{A}\Delta\bar{y}_t + \bar{E}\Delta\bar{y}_{t-1} + \bar{F}\Delta\bar{x}_t$

It can be solved, so we only get the: $\Delta\bar{y}_t$ on the left hand side:

$\Delta\bar{y}_t = (I-\bar{A})^{-1}\bar{E}\Delta\bar{y}_{t-1} + (I-\bar{A})^{-1}\bar{F}\Delta\bar{x}_t$

This expression :

$\Delta\bar{y}_t = (I-\bar{A})^{-1}\bar{E}\Delta\bar{y}_{t-1} + (I-\bar{A})^{-1}\bar{F}\Delta\bar{x}_t$

Which can be written this way:

$$\underbrace{\begin{bmatrix}\Delta y_t^3\\ \Delta y_t^2\\ \Delta y_t^1\\ \Delta y_t\end{bmatrix}}_{\Delta\bar{y}_t} = \underbrace{\begin{bmatrix}0 & \mathbb{I} & 0 & 0\\ 0 & 0 & \mathbb{I} & 0\\ 0 & 0 & 0 & \mathbb{I}\\ (\mathbb{I}-A)^{-1}E_3 & (\mathbb{I}-A)^{-1}E_2 & (\mathbb{I}-A)^{-1}E_1 & (\mathbb{I}-A)^{-1}E\end{bmatrix}}_{\text{companion matrix}=(I-\bar{A})^{-1}\bar{E}} \underbrace{\begin{bmatrix}\Delta y_{t-1}^3\\ \Delta y_{t-1}^2\\ \Delta y_{t-1}^1\\ \Delta y_{t-1}\end{bmatrix}}_{\Delta\bar{y}_{t-1}} +$$

$$\underbrace{\begin{bmatrix}0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\\ (\mathbb{I}-A)^{-1}F_3 & (\mathbb{I}-A)^{-1}F_2 & (\mathbb{I}-A)^{-1}F_1 & (\mathbb{I}-A)^{-1}F_0\end{bmatrix}}_{(I-\bar{A})^{-1}\bar{F}} \underbrace{\begin{bmatrix}\Delta x_{t-3}\\ \Delta x_{t-2}\\ \Delta x_{t-1}\\ \Delta x_t\end{bmatrix}}_{\Delta\bar{x}_t}$$

So now we have a linear model with only one lag. For this kind of models the dynamic behavior can be evaluated by looking at the eigenvalues e_t of $((I-\bar{A})^{-1}\bar{E})$

If all $|e_t| < 1$ the system will converge. If at least one of the eigenvalues is larger than one, the system will explode. e_t can are complex numbers. If at least one e_t has an imaginary part the system will oscillate - either dampened if all $|e_t| < 1$ or exploding if one $|e_t| > 1$.

The eigenvalues and associated eigenvectors can be found by the function mdif.get_eigen which will be used below.

5.5.1 Example: Samuelson multiplier accelerator model

To illustrate the feature this small classic model is created in a function which will display the results and a polar plot of the complex eigenvalues.

Then experiment with different parameter values are performed in order to establish if shocks are amplified or dampened and if shocks will induce oscillations or not.

```
In [46]: def geteigen(mul,acc,years=100,show=False):
             '''Function which creates a Samuelson Multiplier accelerator model, runs it
             and alculates the eigenvalues for the compaignion matrix in order to
             evaluate stability.

             A polar plot of the compex eigenvalues and graphs of the endogenous variables can
             be displayes '''

             fam    = f'''
             frml <>  y = c+i $
             frml <I> c = {mul} * y(-1) $
```

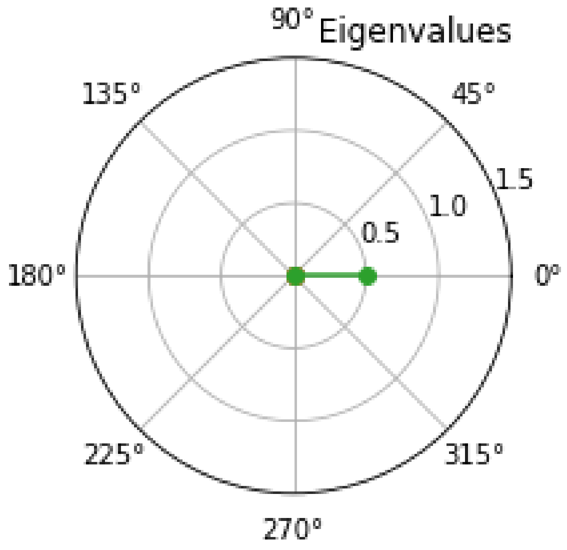
```
        frml <>  i = {acc} * (c-c(-1)) + im $'''

        mma = mc.model(fam,modelname = 'Accelerator multiplicator model')
        df = pd.DataFrame([[1000,200]]*years,index=range(2018,2018+years),columns=['Y','IM'])
        start = mc.model(f'c = {mul} * y(-1)')(df,silent=True) # Generate lagged variables for c
        base  = mma(start,silent=True)     # Solve the model

        compeig=mdif.get_eigen(mma,base,2021)       # find the eigenvalues and eigenvectors
        if show:
            mdif.eigplot(compeig[0])         # show the eigenvalues
            _ = mma[['Y','C','I']].plot()  # Show the solution

        return
```

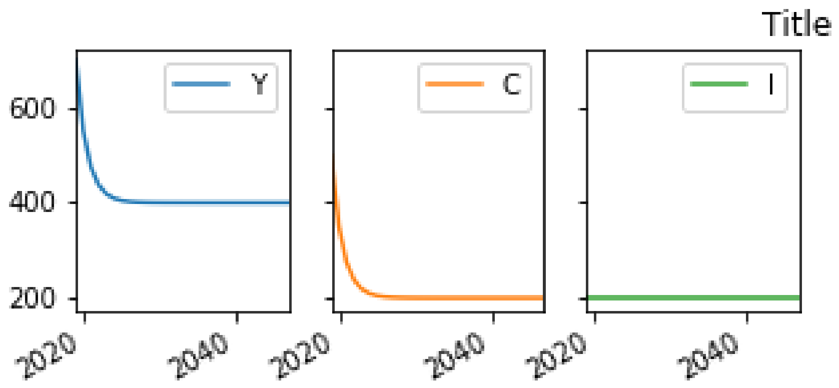
Stability

All eigenvalues shorter than 1 and no imaginary parts

```
In [47]: geteigen(mul=0.5,acc=0,years=30,show=1)
```

```
Finding derivatives started at :        10:40:27
Finding derivatives took       :    0.0125000477 Seconds
Calculating derivatives started at :        10:40:27
Calculating derivatives took       :    0.0025000572 Seconds
creating dataframe started at :        10:40:27
creating dataframe took       :    0.0000000000 Seconds
```



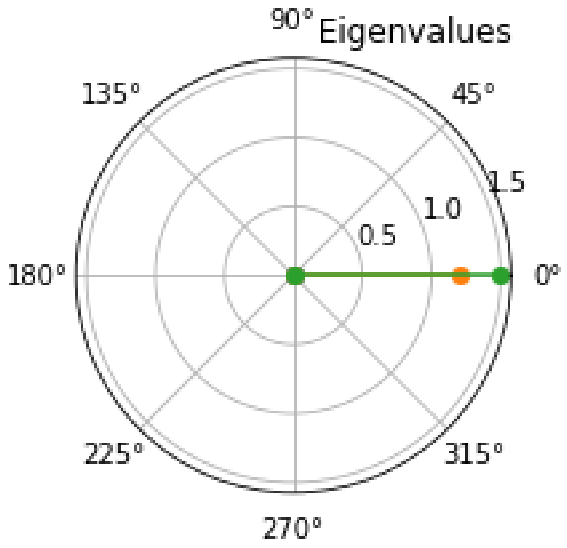


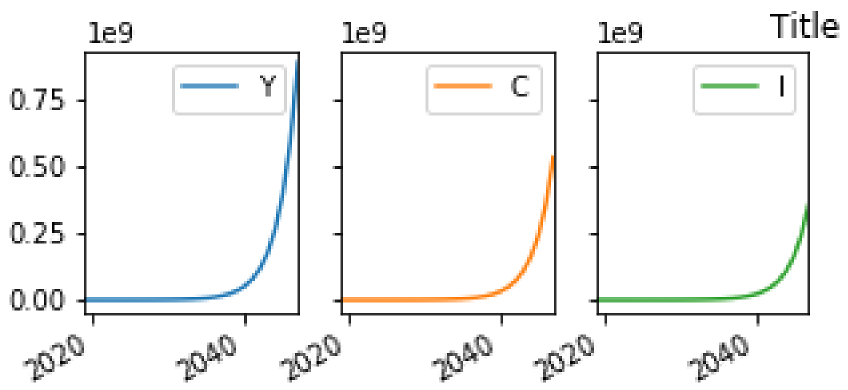
Explosion

At least one eigenvalues above 1 no imaginary part

```
In [48]: geteigen(mul=0.9,acc=2,years=30,show=1)
```

```
Finding derivatives started at :        10:40:28
Finding derivatives took       :    0.0075001717 Seconds
Calculating derivatives started at :        10:40:28
Calculating derivatives took       :    0.0049998760 Seconds
creating dataframe started at :        10:40:28
creating dataframe took       :    0.0000000000 Seconds
```



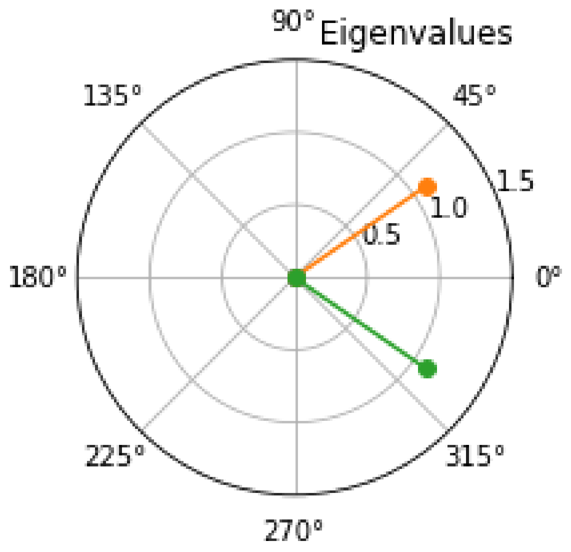


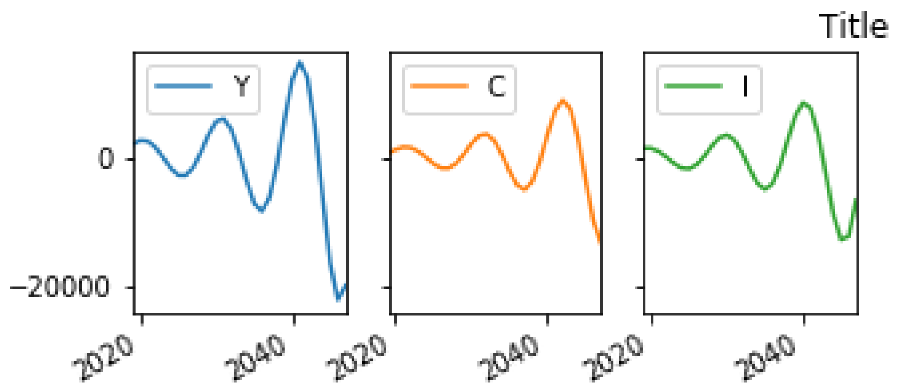
Exploding oscillations

At least one eigenvalue above 1 and imaginary parts

```
In [49]: geteigen(mul=0.6,acc=2,years=30,show=1)
```

```
Finding derivatives started at :        10:40:28
Finding derivatives took       :    0.0075001717 Seconds
Calculating derivatives started at :        10:40:28
Calculating derivatives took       :    0.0050001144 Seconds
creating dataframe started at :        10:40:28
creating dataframe took       :    0.0000000000 Seconds
```



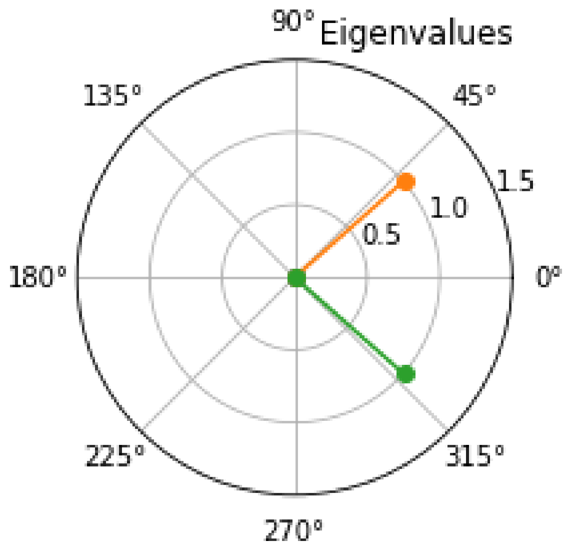


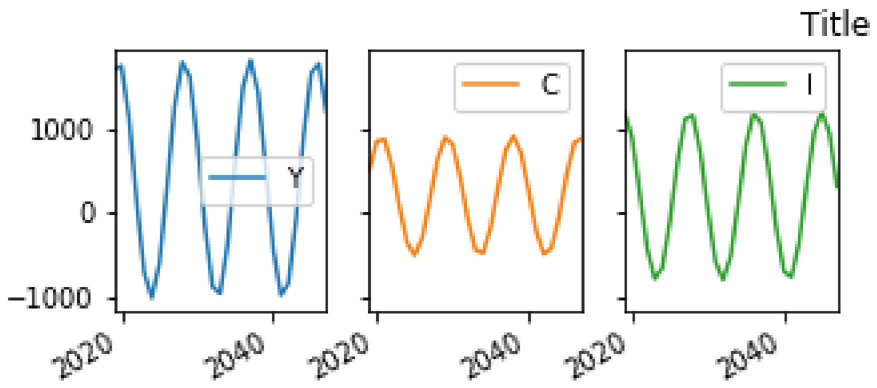
Perpetual oscillations

Eigenvalues at 1 with imaginary part

```
In [50]: geteigen(mul=0.5,acc=2,years=30,show=1)
```

```
Finding derivatives started at :        10:40:29
Finding derivatives took       :    0.0050001144 Seconds
Calculating derivatives started at :        10:40:29
Calculating derivatives took       :    0.0049998760 Seconds
creating dataframe started at :        10:40:29
creating dataframe took       :    0.0000000000 Seconds
```



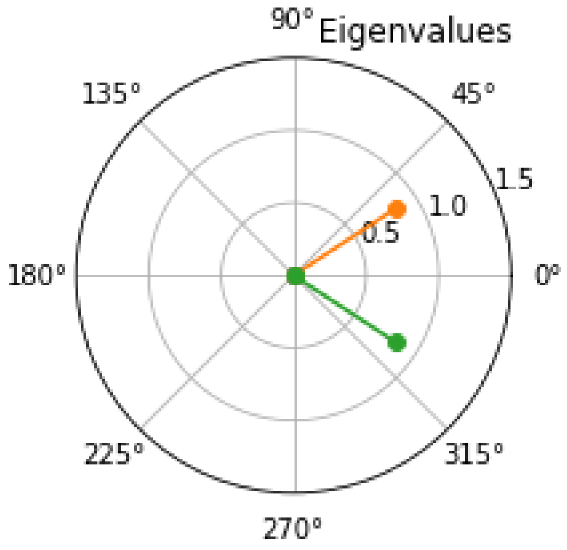


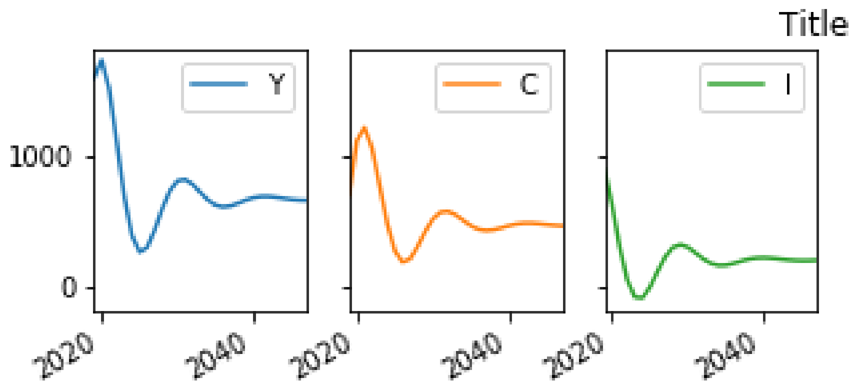
Dampened oscillations

All eigenvalues below 1 and imaginary parts

```
In [51]: geteigen(mul=0.7,acc=1,years=30,show=1)
```

```
Finding derivatives started at :        10:40:29
Finding derivatives took       :    0.0181002617 Seconds
Calculating derivatives started at :        10:40:29
Calculating derivatives took       :    0.0050001144 Seconds
creating dataframe started at :        10:40:29
creating dataframe took       :    0.0000000000 Seconds
```





Using this feature. ModelFlow can be used to analyze how different feedback parameters will impact stability. Of cause it requires that the model is differentiable, and for large models the analysis can take time and a lot of RAM.

Chapter 6

Onboarding a model

We want to present a model to the transpiler, so it can create a model instance including a python function which can solve/calculate the model. The primary method for this is to formulate the model in a Business logic Language.

The purpose of the Business logic language is to enable a clear and concise specification of the models at hand.

A *template model* can flow through a *pre-processor* which returns *business logic*. This is then processed by transpiler to create a python function which can solve the model.

The pre-processer performs text processing:

- Normalize formulas
- Decorate formulas
- Expand arrays
- Expand matrices
- Unlooping do-loops

The inspiration is the modeling language of venerable programs like TSP, Eviews, Troll and SAS/ETS combined with the rules of Python.

If you launch this file as Jupyter notebook, you can play around with the business logic language.

There are three types of statements:

1. Formula statement:
   Defines transformation of variables in the Dataframe
2. List statement:
   Defines lists and associated sub lists. Lists can be used to define repetitive items and to define mappings
3. DO/END DO statements
   Defines repetitive formulas in which the content of list are interpolated

6.1 A Flexible language

The example above shows how to specify a simple model. It looks like python, except that variable names are substituted to references to pandas dataframe cells after Transpiling. This is fine for some problems.

However a **preprocessing** template expanding phase enhance the language in order to make it more powerful, agile, and and still parsimonious.

The expanded business logic language introduce a little more. It consists of commands and comments:

- frml: Define formulas
- list: Define sublist to explode over
- do: start exploding
- enddo: end exploding

6.1.1 Variables

Variable names has to start with a letter or _ then any number of letters _ or digits. Time t is implicit.

A lag can be specified right after a variable as: (-), meaning: horse(-1) is the variable horse lagged one period

A variable can contain a number or an Python object. The Python object can be for instance be a cvxopt matric, a numpy array or even a Pandas dataframe.

Case does not matter. All variables are translated to upper case. And in the Dataframe all variables should be upper case.

Lags cannot be specified at the left hand side of =.

6.2 FRML command, defines calculations

frml left hand side = right hand side $

If it is done for the whole model, the word frml and the <> clause and $ can be ignored. In this case commands including formulas are separated by newline.

6.2.1 Right hand side

6.2.2 Operators

operators: = + - / * ** ( )@
comparison: >= <= == # evaluates to 0 if false, 1 if true

special: $ > < , .] [# used in different python constructions.

6.2.3 Several variables on the left hand side

There can be several left hand variables. Python rules for tuple unpacking apply.

a, b, c = 1, 2, 3 $

or

a, b, b = some_function_which returns_3 values(x,y,z)

Be aware that normalization cannot be combined with this feature.

The feature is useful when packing and unpacking matrices and when applying optimization.

6.2.4 Text processing functions on the right hand side

The functions DIFF and DLOG will be expanded on the right hand side. Their argument can be expressions involving lagged variable, therefore they can't be evaluated directly, so they are expanded in the pre-processing phase.

```
In [52]: explode('a = alfa * diff(b)')

Out[52]: 'FRML <> A = ALFA * ((B)-(B(-1))) $'

In [53]: explode('a = alfa * dlog(b)')

Out[53]: 'FRML <> A = ALFA * ((LOG(B))-(LOG(B(-1)))) $'

In [54]: explode('a = alfa * dlog(a+b+c)')

Out[54]: 'FRML <> A = ALFA * ((LOG(A+B+C))-(LOG(A(-1)+B(-1)+C(-1)))) $'
```

6.2.5 Text processing functions on the left hand of a formular

In some models, it can be useful to specify a function of the endogenous variable as the target on the left-hand side. In order to do the calculation, the endogenous variable has to be isolated on the left-hand side. This is done in a pre-processing step by applying the inverse of the function to the right-hand side. Functions allowed on left-hand side: **diff, log, dlog, logit**.

These functions are only allowed when there are just one left-hand side variable.

```
In [55]: explode('diff(lhv) = 42 * (a+b+c) ')

Out[55]: 'FRML <> LHV=LHV(-1)+(42 * (A+B+C))$'

In [56]: explode('log(lhv) = 42 * (a+b+c) ')

Out[56]: 'FRML <> LHV=EXP(42 * (A+B+C))$'

In [57]: explode('logit(lhv) = 42 * (a+b+c) ')

Out[57]: 'FRML <> LHV=(EXP(42 * (A+B+C))/(1+EXP(42 * (A+B+C))))$'

In [58]: explode('dlog(lhv) = 42 * (a+b+c) ')

Out[58]: 'FRML <> LHV=EXP(LOG(LHV(-1))+42 * (A+B+C))$'
```

6.2.6 Formula name, enhancing formular calculations

The purpose of the formula name/control is to enable relevant additions to the formulas, but avoid boilerplate noise to the formulas.

The control is placed between <> after the 'FRML'. different controls can be separated by ','

Absolute adjustment

```
In [59]: mp.explode('<J> lhv = rhs ')

Out[59]: 'FRML <J> LHV = RHS  +LHV_J$'
```

Relative adjustment

```
In [60]: mp.explode('<jr> lhv = rhs ')

Out[60]: 'FRML <JR> LHV = ( RHS  )*(1+LHV_JR )$'
```

Fixing results

```
In [61]: mp.explode('frml <exo> lhv = rhs ')

Out[61]: 'FRML <EXO> LHV =( RHS )*(1-LHV_D)+LHV_X*LHV_D$'
```

By setting LHV_D the user controls wether LHV is set to the value of the RHS exression or the value LHV_X.

Extract model for calculating identities in before projection period

```
In [62]: tmodel = '''\
         frml <> horse = 1.3 * cow +dgdp(-1)$
         frml <I> dgdp = gdp/gpd(-1) $
         frml <i> dinv = inv/inv(-1) $ '''
         print( mp.explode(tmodel))

FRML <> HORSE = 1.3 * COW +DGDP(-1)$
FRML <I> DGDP = GDP/GPD(-1) $
FRML <I> DINV = INV/INV(-1) $


In [63]: mp.find_hist_model(tmodel)

Out[63]: '  frml <I> dgdp = gdp/gpd(-1) $  frml <i> dinv = inv/inv(-1) $'
```

Make dampening a formular possible

Z : When solving a model with the Gauss-Seidel algorithm the evaluation of this formula can be damped.

6.2.7 Incorporating user defined functions into formulas

The user can incorporate her own routines in the BL language. This is giving the user defined functions as the funks argument to the model creation.

The tokenizer will recognize the name of the function and place call to the function in the solving routine.

```
In [64]: def f1(x):
             return 42

         f2model = '''\
         a      = c * 42
         f      = c(-1) >=2
         answer = f1(a)'''
         m2model = mc.create_model(f2model,funks=[f1])
         m2model(df)
```

```
Will start calculating: indtastet_udtryk
2019  solved
2020  solved
2021  solved
indtastet_udtryk calculated
```

```
Out[64]:         B    C    E  ANSWER    F      A
         2018  1.0  4.0  4.0     0.0  0.0    0.0
         2019  2.0  5.0  4.0    42.0  1.0  210.0
         2020  3.0  7.0  4.0    42.0  1.0  294.0
         2021  4.0  9.0  4.0    42.0  1.0  378.0
```

6.3 LIST command, define list and sublists

Lists are used to direct how the do command explodes a template model. And to define sum functions, and for matrix packing and unpacking.

```
List <listname> = <sublistname0> : <sl00> <sl01> .. <sl0n> [/
                     <sublistname1> : <sl10> <sl11> .. <sl1n> /
                     .............................]
                     $
```

There should always be the same number of elements in sublists
Example with one sublist:

```
LIST BANKLIST = BANK : ATBAW DEHAS DEHSH LUPCK DELBY DEVWF ESSAB  $
```

Example with two sublist:

```
LIST CRCOLIST  =  CRCOUNTRY   : UK DE CH NL BE CA JP KY LU HK  /
                  CRMODELGEO  : UK DE CH NL BE CA JP R20 LU HK $
```

If a list name is created several times, the first time is the used. This makes it more easy to make prototypes.

6.4 DO ... ENDDO, loop over lists

```
do <listname> $
stuff
enddo $
```

Will take take the stuff and make a copy for each element of the sublists in list name.
Remember that all sublists have to have the same number of elements

```
In [65]: mtest = '''
         list banklist = bank : ib soren $
         do banklist $
              profit_{bank} = revenue_{bank} - expenses_{bank} $
         enddo $
         '''

         print(explode(mtest))
```

```
LIST BANKLIST = BANK : IB SOREN $
PROFIT_IB = REVENUE_IB - EXPENSES_IB $
PROFIT_SOREN = REVENUE_SOREN - EXPENSES_SOREN $
```

6.4.1 Use of sublist to inject additional information

```
In [66]: mtest = '''
         list banklist = bank    : ib      soren  marie /
                         country : denmark sweden denmark  $

         do banklist $
             frml <> profit_{bank} = revenue_{bank} - expenses_{bank} $
             frml <> expenses_{bank} = factor_{country} * revenue_{bank} $
         enddo $
         '''

         print(mp.explode(mtest))
```

```
LIST BANKLIST = BANK    : IB      SOREN  MARIE /
                COUNTRY : DENMARK SWEDEN DENMARK  $
FRML <> PROFIT_IB = REVENUE_IB - EXPENSES_IB $
FRML <> EXPENSES_IB = FACTOR_DENMARK * REVENUE_IB $
FRML <> PROFIT_SOREN = REVENUE_SOREN - EXPENSES_SOREN $
FRML <> EXPENSES_SOREN = FACTOR_SWEDEN * REVENUE_SOREN $
FRML <> PROFIT_MARIE = REVENUE_MARIE - EXPENSES_MARIE $
FRML <> EXPENSES_MARIE = FACTOR_DENMARK * REVENUE_MARIE $
```

6.4.2 Nested do

Do loops can be nested.

One just have to take care that there is no duplication of sublist names.

```
In [67]: mtest = '''
         list banklist = bank    : ib      soren  marie /
                         country : denmark sweden denmark  $
```

```
list assetlist = asset  : nfc sme hh             $


do banklist $
  ! bank = {bank}
    profit_{bank} = revenue_{bank} - expenses_{bank}-loss_{bank} $
    expenses_{bank} = factor_{country} * revenue_{bank} $
    do assetlist $
      loss_{asset}_{bank} = 0.02 * stock_{asset}_{bank} $
    enddo $
    ! find the sum
    loss_{bank} = sum(assetlist,loss_{asset}_{bank})    $

enddo $
'''

print(explode(mtest))
```

```
LIST BANKLIST = BANK    : IB      SOREN  MARIE /
                COUNTRY : DENMARK SWEDEN DENMARK  $
LIST ASSETLIST = ASSET  : NFC SME HH             $
BANK = IB
    PROFIT_IB = REVENUE_IB - EXPENSES_IB-LOSS_IB $
EXPENSES_IB = FACTOR_DENMARK * REVENUE_IB $
LOSS_NFC_IB = 0.02 * STOCK_NFC_IB $
LOSS_SME_IB = 0.02 * STOCK_SME_IB $
LOSS_HH_IB = 0.02 * STOCK_HH_IB $
FIND THE SUM
    LOSS_IB = (LOSS_NFC_IB+LOSS_SME_IB+LOSS_HH_IB)    $
BANK = SOREN
    PROFIT_SOREN = REVENUE_SOREN - EXPENSES_SOREN-LOSS_SOREN $
EXPENSES_SOREN = FACTOR_SWEDEN * REVENUE_SOREN $
LOSS_NFC_SOREN = 0.02 * STOCK_NFC_SOREN $
LOSS_SME_SOREN = 0.02 * STOCK_SME_SOREN $
LOSS_HH_SOREN = 0.02 * STOCK_HH_SOREN $
FIND THE SUM
    LOSS_SOREN = (LOSS_NFC_SOREN+LOSS_SME_SOREN+LOSS_HH_SOREN)    $
BANK = MARIE
    PROFIT_MARIE = REVENUE_MARIE - EXPENSES_MARIE-LOSS_MARIE $
EXPENSES_MARIE = FACTOR_DENMARK * REVENUE_MARIE $
LOSS_NFC_MARIE = 0.02 * STOCK_NFC_MARIE $
LOSS_SME_MARIE = 0.02 * STOCK_SME_MARIE $
LOSS_HH_MARIE = 0.02 * STOCK_HH_MARIE $
FIND THE SUM
    LOSS_MARIE = (LOSS_NFC_MARIE+LOSS_SME_MARIE+LOSS_HH_MARIE)    $
```

6.4.3 Use of sublist to do conditional Loop

By using

```
DO listname sublist = value $
```

the looping will only be performed on the members of list where the value of the corresponding sublist match the value in the do statement.

This can be useful for instance when treating portfolios differently depending on some information. An example could be the different treatment of portfolios depending on regulatory approach.

```
In [68]: mtest='''
         list portdic = port  : households ,  NFC ,   RE,  sov  /
                        reg   :       AIRB , AIRB ,  STA,  STA $
             do portdic reg = airb $
             frml x {port}_REA_W = calc_rw({port}_pd ,{port}_lgd) $
             enddo $
             do portdic reg = sta $
             frml x {port}_REA_W = {port}_REA_W(-1) $
             enddo $
          '''
         print(explode(mtest))
```

```
LIST PORTDIC = PORT  : HOUSEHOLDS ,  NFC ,   RE,  SOV  /
               REG   :       AIRB , AIRB ,  STA,  STA $
FRML X HOUSEHOLDS_REA_W = CALC_RW(HOUSEHOLDS_PD ,HOUSEHOLDS_LGD) $
FRML X NFC_REA_W = CALC_RW(NFC_PD ,NFC_LGD) $
FRML X RE_REA_W = RE_REA_W(-1) $
FRML X SOV_REA_W = SOV_REA_W(-1) $
```

6.4.4 Dynamic defined lists to utilize sparsity

Often not all the potential dimensions in a model contains data. For instance all banks don't have positions in all potential countries.

In order to speed up calculations and to avoid bloating dataframes, we want to avoid calculating and carrying a lot of zeros around.

This can be achieved by using dynamic naming of lists. For instance create a separate list of countries for each bank.

The list for each country can be created by using the pandas library to identify in which countries there are non-zero positions - straight forward but beyond the scope of this notebook.

Below is an example where there are two banks which each has exposures in different countries.

```
In [69]:     stest='''
             list BANKDIC = bank : Danske , Nordea $
             list country_danske_dic = country : uk , DK, IR $
             list country_nordea_dic = country: SE , DK $
             do bankdic $
                 frml x {bank}_income = {bank}_a +{bank}_b $
```

```
                do country_{bank}_dic $
                    frml x value_{bank}_{country} = 42 $
                enddo $
                frml <> total_{bank} = sum(country_{bank}_dic,value_{bank}_{country})  $
              enddo $ '''

            print(explode(stest))
```

```
LIST BANKDIC = BANK : DANSKE , NORDEA $
LIST COUNTRY_DANSKE_DIC = COUNTRY : UK , DK, IR $
LIST COUNTRY_NORDEA_DIC = COUNTRY: SE , DK $
FRML X DANSKE_INCOME = DANSKE_A +DANSKE_B $
FRML X VALUE_DANSKE_UK = 42 $
FRML X VALUE_DANSKE_DK = 42 $
FRML X VALUE_DANSKE_IR = 42 $
FRML <> TOTAL_DANSKE = (VALUE_DANSKE_UK+VALUE_DANSKE_DK+VALUE_DANSKE_IR)  $
FRML X NORDEA_INCOME = NORDEA_A +NORDEA_B $
FRML X VALUE_NORDEA_SE = 42 $
FRML X VALUE_NORDEA_DK = 42 $
FRML <> TOTAL_NORDEA = (VALUE_NORDEA_SE+VALUE_NORDEA_DK)  $
```

6.5 Pack values into matrices (a cvxopt class)

```
In [70]: mtest2 = '''
         list banklist  = bank : ib soren $
         list banklist2 = bank2 : ib soren $
         ! vector
         frml <matrix> vbank = to_matrix(banklist,profit_{bank})  $
         ! matrices
         frml <matrix> Mbank = to_matrix(banklist,banklist2,loanfrom_{bank}_to_{bank2})  $
         '''

         print(mp.explode(mtest2))
```

```
LIST BANKLIST  = BANK : IB SOREN $
LIST BANKLIST2 = BANK2 : IB SOREN $
VECTOR FRML <MATRIX> VBANK = MATRIX(
[PROFIT_IB,PROFIT_SOREN])  $
MATRICES FRML <MATRIX> MBANK = MATRIX(
[[LOANFROM_IB_TO_IB,LOANFROM_IB_TO_SOREN],
[LOANFROM_SOREN_TO_IB,LOANFROM_SOREN_TO_SOREN]])  $
```

6.6 Pack values into arrays (a numpy class)

```
In [71]: mtest2 = '''
         list banklist  = bank : ib soren $
         list banklist2 = bank2 : ib soren $
         ! vector
         frml <matrix> vbank = to_array(banklist,profit_{bank})  $
         ! matrices
         frml <matrix> Mbank = to_array(banklist,banklist2,loanfrom_{bank}_to_{bank2})  $
         '''

         print(mp.explode(mtest2))
```

```
LIST BANKLIST  = BANK : IB SOREN $
LIST BANKLIST2 = BANK2 : IB SOREN $
VECTOR FRML <MATRIX> VBANK = ARRAY(
[PROFIT_IB,PROFIT_SOREN])  $
MATRICES FRML <MATRIX> MBANK = ARRAY(
[[LOANFROM_IB_TO_IB,LOANFROM_IB_TO_SOREN],
[LOANFROM_SOREN_TO_IB,LOANFROM_SOREN_TO_SOREN]])  $
```

6.7 Unpack values - Argexpand

```
In [72]: mtest2 = '''
         list banklist  = bank : ib soren $
         list banklist2 = bank2 : ib soren $
         ! vector
         frml <> argexpand(banklist,test_{bank}) = to_matrix(banklist,profit_{bank})  $
         ! matrices
         frml <> argexpand(banklist,argexpand(banklist2,loanfrom_{bank2}_{bank})) = 0  $
         '''

         print(mp.explode(mtest2))
```

```
LIST BANKLIST  = BANK : IB SOREN $
LIST BANKLIST2 = BANK2 : IB SOREN $
VECTOR FRML <> TEST_IB,TEST_SOREN = MATRIX(
[PROFIT_IB,PROFIT_SOREN])  $
MATRICES FRML <> LOANFROM_IB_IB,LOANFROM_SOREN_IB,LOANFROM_IB_SOREN,LOANFROM_SOREN_SOREN = 0  $
```

```
In [73]: mtest2 = '''
         list listdevs  = devs : dev_yer dev_uts $
         list listshocks = shocks : ib soren $
             frml <matrix> ARGEXPAND(listdevs,{devs}) = STE(to_matrix(LISTSHOCKS,{SHOCKS}(-3)),\
            to_matrix(LISTSHOCKS,{SHOCKS}(-2 )), \
            to_matrix(LISTSHOCKS,{SHOCKS}(-1)),
            to_matrix(LISTSHOCKS,{SHOCKS})  ) $
            '''.upper()

        print(mp.explode(mtest2))
```

```
LIST LISTDEVS  = DEVS : DEV_YER DEV_UTS $
LIST LISTSHOCKS = SHOCKS : IB SOREN $
FRML <MATRIX> DEV_YER,DEV_UTS = STE(MATRIX(
[IB(-3),SOREN(-3)]),    MATRIX(
[IB(-2 ),SOREN(-2 )]),     MATRIX(
[IB(-1),SOREN(-1)]),
    MATRIX(
[IB,SOREN])  ) $
```

6.8 Sum, sum over list

sum(LIST,expression) = sums expression over the elements of a list substitution the sublist names.

```
In [74]: mtest = '''
         list banklist = bank    : ib      soren  marie /
                         country : denmark sweden denmark  $

         list assetlist = asset  : nfc sme hh             $


         do banklist $
           !
           ! bank = {bank}
             do assetlist $
               loss_{asset}_{bank} = 0.02 * stock_{asset}_{bank} $
             enddo $
             ! find the sum
             loss_{bank} = sum(assetlist,loss_{asset}_{bank})   $

         enddo $

         loss_total =  sum(banklist,sum(assetlist,loss_{asset}_{bank}))$
         '''

         print(explode(mtest))
```

```
LIST BANKLIST = BANK    : IB      SOREN  MARIE /
                COUNTRY : DENMARK SWEDEN DENMARK  $
```

```
LIST ASSETLIST = ASSET  : NFC SME HH             $
BANK = IB
    DO ASSETLIST $
LOSS_{ASSET}_IB = 0.02 * STOCK_{ASSET}_IB $
BANK = SOREN
    DO ASSETLIST $
LOSS_{ASSET}_SOREN = 0.02 * STOCK_{ASSET}_SOREN $
BANK = MARIE
    DO ASSETLIST $
LOSS_{ASSET}_MARIE = 0.02 * STOCK_{ASSET}_MARIE $
FIND THE SUM
    LOSS_{BANK} = (LOSS_NFC_{BANK}+LOSS_SME_{BANK}+LOSS_HH_{BANK})   $
LOSS_TOTAL =  ((LOSS_NFC_IB+LOSS_SME_IB+LOSS_HH_IB)+(LOSS_NFC_SOREN+LOSS_SME_SOREN+LOSS_HH_SOREN)+(LOSS
```

6.9 Onboarding models from other sources

Python has incredible strong tools both for interacting with other systems like Excel and Matlab, and for text processing. This allows for *recycling* of models from different sources and make them live together (hopefully happy). Some of the sources, from which models has been recycled are:

- Latex,
  - Code from Dynare
  - Model written in Latex - with some rules to allow text processing.
- Excel
  - Calculation model from Excel workbook
  - Grabbing coefficients from excel workbooks
- Matlab
  - Wrapping matlab models into python functions, which can be used in ModelFlow
  - Grabbing coefficients from matlab .mat files.
- Aremos models
- TSP models

Grabbing models and transforming them to Business logic language usually requires a tailor-made Python program. However in the ModelFlow folder there are different examples of such grabbing.

Chapter 7

Onboarding data

The Pandas library really shines when it comes to retrieving and wrangling data. How to get data on board is very specific for each model and data source. There are many great books on using Pandas for data wrangling. So it is beyond the scope of this notebook to go into details on this subject. However, it is useful to go through some naming advice and a broad description of how data can be transformed to the right format.

ModelFlow need to get the data as a wide Pandas DataFrame. Each row is one time frame. Each columns s a variable.

There are no limits to the length of column names. Therefor column names can be constructed meaningful. a column name should start with a letter then it can be followed by any numbers of letters, digits or _. Letters have to be upper case. If lower case letters are used in the Business logic, they will be converted to upper case.

Often variables reflect a number of different dimensions. When naming variables separate the dimensions by a separator string.

$\underbrace{REA\_NON\_DEF}_{type}\_\_\underbrace{DECOM}_{bank}\_\_\underbrace{DE}_{Country}\_\_\underbrace{HH\_OTHER\_NSME\_AIRB}_{Sector}$

Naming variables along dimensions is supported by ModelFlows slicing methods. Slicing allows for filtering of variables with the help of wildcards like * and ?.

model['REA_NON_DEF__*__DE__HH_OTHER_NSME_AIRB'] will return a dataframe with all columns where:

- type = REA_NON_DEF
- banks = all
- country= DE
- sector = HH_OTHER_NSME_AIRB

If we would limit the results to German banks we just write:
model['REA_NON_DEF__DE???__DE__HH_OTHER_NSME_AIRB']

7.1 From tall to wide dataframes:

Typically the necessary wide dataframe is constructed from a tall DataFrame. In this tall dataframe has a row for each value and a column for each dimension - including time. If we just look at the few rows related to the variable mentioned above, it could look like this. (of course mock data)

value	year	type	bank	county	sector
.	.	.	.	.	.
.	.	.	.	.	.
1.00	2018	REA_NON_DEF	DECOM	DE	HH_OTHER_NSME_AIRB
2.22	2019	REA_NON_DEF	DECOM	DE	HH_OTHER_NSME_AIRB
3.33	2020	REA_NON_DEF	DECOM	DE	HH_OTHER_NSME_AIRB
4.44	2021	REA_NON_DEF	DECOM	DE	HH_OTHER_NSME_AIRB
.	.	.	.	.	.
.	.	.	.	.	.

First step is to create a new dataframe with only the value, the year and the variable name. The variable name (varname) is constructed by joining the dimensions and separate them by '__'

value	year	varname
.	.	.
.	.	.
1.00	2018	REA_NON_DEF__DECOM__DE__HH_OTHER_NSME_AIRB
2.22	2019	REA_NON_DEF__DECOM__DE__HH_OTHER_NSME_AIRB
3.33	2020	REA_NON_DEF__DECOM__DE__HH_OTHER_NSME_AIRB
4.44	2021	REA_NON_DEF__DECOM__DE__HH_OTHER_NSME_AIRB
.	.	.
.	.	.

Then the dataframe is pivoted keeping the year dimension. Now the wide dataframe (ignoring all the other columns) for this variable could look like this:

YEAR	...	REA_NON_DEF__DECOM__DE__HH_OTHER_NSME_AIRB	...
2018	...	1.00	...
2018	...	2.22	...
2018	...	3.33	...
2018	...	4.44	...

A wide dataframe like this can be used as input to a ModelFlow model, so we are in business.

Chapter 8

Summary

ModelFlow allows easy implementation of models in Python, Which is a powerful and agile language. ModelFlow leverage on the rich ecosystem of Python in order to:

- Separates the specification of a model and the code which solves the model. So the user can concentrate on the economic and not the implementation of the model.
- Can include user specified Python function in the model definition.
- Can solve very large models
- Can solve simultaneous models.
- Keeps tab on the dependencies of the formulas. This allows for easy Tracing of results.
- Can perform model inversion (goal seek) with multiple targets and instruments
- Can attribute changes in results to input variables. Both for individual formulas and the complete model
- Can include optimizing behavior

The purpose of this notebook has been to give a broad introduction to model management using ModelFlow. Using the tool requires some knowledge of python. The required knowledge depends on the complexity of the model. So ModelFlow can be used in Python training.

To get more in-depth knowledge there is a Sphinx based documentation of the library. There you can find the calling conventions and documentation of all elements.

All suggestions and recommendations are welcome

Chapter 9

Literature:

Aho, Lam, Sethi, Ullman (2006), Compilers: Principles, Techniques, and Tools (2nd Edition), Addison-Wesley

Berndsen, Ron (1995), Causal ordering in economic models, Decision Support Systems 15 (1995) 157-165

Danmarks Nationalbank (2004), MONA -- a quarterly model of Danish economy

Denning, Peter J. (2006), The Locality Principle, Chapter in *Communication Networks and Systems* (J Barria, Ed.). Imperial College Press

Gilli, Manfred (1992), Causal Ordering and Beyond, International Economic Review, Vol. 33, No. 4 (Nov., 1992), pp. 957-971

McKinney, Wes (2011),[pandas: a Foundational Python Library for Data Analysis and Statistics,] Presented at PyHPC2011](http://www.scribd.com/doc/71048089/pandas-a-Foundational-Python-Library-for-Data-Analysis-and-Statistics)

Numba (2015) documentation, http://numba.pydata.org/numba-doc/0.20.0/user/index.html

Pauletto, G. (1997), Computational Solution of Large-Scale Macroeconometric Models, ISBN 9781441947789

Tinberger, Jan (1956), Economic policy: Principles and Design, Amsterdam,

Chapter 10

Footnotes

^{1}: The author has been able to draw on experience creating software for solving the macroeconomic models ADAM in Hansen Econometric.

^{2}: In this work a number of staff in Danmarks Nationalbank made significant contributions: Jens Boldt and Jacob Ejsing to the program. Rasmus Tommerup and Lindis Oma by being the first to implement a stress test model in the system.

^{3}: In ECB Marco Gross, Mathias Sydow and many other collegues has been of great help.

^{4}: The system has benefited from discussions with participants in meetings at: IMF, Bank of Japan, Bank of England, FED Board, Oxford University, Banque de France, Single Resolution Board

^{5}: Ast stands for: Abstract Syntax Tree

^{6}: Re stands for: Regular expression

```
In [ ]:
```