ModelFlow_sandbjerg_2021-Copy3

November 16, 2021

[41]:
```
%matplotlib inline
```

[42]:
```
import matplotlib.pyplot as plt
import pandas as pd                # Python data science library
#import numpy as np
#import re
#import sys

from modelclass import model
#import modelpattern as pt
#import modelmanipulation as mp    # Module for model text processing
#from modelmanipulation import explode
```

1 ModelFlow, A library to manage and solve Models

Sandbjerg november 2021

Ib Hansen Ib.hansen.iv@gmail.com

Work done at Danmarks Nationalbank, ECB and Hansen Ølkonometri.

Problem: Stress-test model for banks Complicated and slow **Excel** workbook Difficult to maintain and change

Solution: Separate specification (at a high level of abstraction) and solution code. Python comes **batteries included**. Data management, text processing, visualization …

Implementation of **minimum workable toolkit** A **transpiler**: Takes a model in a **domain specific** Business logic language: Create **solver** and **utility** functions using Python libraries.

Refactor and refine Larger models, Faster transpiler, Newton and Gauss solvers, Logical structure, Derivatives, Visualization, Front ends …

1.1 Refine and refactor

To suit the needs of the different models thrown at the toolkit.

Specify very large (or small) models as concise and intuitive equations. 1 million equation and more can be handled.

Agile model development Model are specified at a high level of abstraction and are processed fast. Experiments with model specification are agile and fast.

Onboarding models and combining from different sources. Recycling and combining models specified in different ways: Excel, Latex, Dynare, Python or other languages.

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 What is a Model in ModelFlow

ModelFlow is created to handle models. The term **model** can mean a lot of different concepts.

The scope of models handled by ModelFlow is **discrete** models which is the same for each time frame, can be formulated as **mathematical equations** and *can* have **lagged** and **leaded** variables. This allows the system to handle quite a large range of models.

A model with:

- **n** number of endogeneous variables
- **k** number of exogeneous variables
- **u** max lead of endogeneous variables
- **r** max lag of endogeneous variables
- **s** max lag of exogeneous variables
- t time frame (year, quarter, day second or another another unit)

can be written in two ways, **normalized** or **un-normalized** form

1.2.1 Normalized model

Each endogenous variable is on the left hand side one time - and only one time.

$$\begin{aligned}
y_t^1 &= f^1(y_{t+u}^1 ..., y_{t+u}^n ..., y_t^2 ..., y_t^n ... y_{t-r}^1 ..., y_{t-r}^n, x_t^1 ... x_t^k, ... x_{t-s}^1 ..., x_{t-s}^k) && (1)\\
y_t^2 &= f^2(y_{t+u}^1 ..., y_{t+u}^n ..., y_t^1 ..., y_t^n ... y_{t-r}^1 ..., y_{t-r}^n, x_t^1 ... x_t^k, ... x_{t-s}^1 ..., x_{t-s}^k) && (2)\\
&\vdots && (3)\\
y_t^n &= f^n(y_{t+u}^1 ..., y_{t+u}^n ..., y_t^1 ..., y_t^{n-1} ... y_{t-r}^1 ..., y_{t-r}^n, x_t^1 ... x_t^r, x ..._{t-s}^1 ..., x_{t-s}^k) && (4)
\end{aligned}$$

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

$$\mathbf{y}_t = \mathbf{F}(\mathbf{y}_{t+u} \cdots \mathbf{y}_t \cdots \mathbf{y}_{t-r}, \mathbf{x}_t \cdots \mathbf{x}_{t-s}) \quad (5)$$

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

1.2.2 Un-normalized form

Some models can not easy be specified as normalized formulas. Especially models with equilibrium conditions can more suitable be specified in the more generalized un-normalized form.

Written in matrix notation like before:

$$\mathbf{0} \quad = \quad \mathbf{F}(\mathbf{y}_{t+u} \cdots \mathbf{y}_t \cdots \mathbf{y}_{t-r}, \mathbf{x}_t \cdots \mathbf{x}_{t-s}) \tag{6}$$

The number of endogenous variables and equations should still be the same.

Model solution For a normalized model:

$$\mathbf{y}_t \quad = \quad \mathbf{F}(\mathbf{y}_{t+u} \cdots \mathbf{y}_t \cdots \mathbf{y}_{t-r}, \mathbf{x}_t \cdots \mathbf{x}_{t-r}) \tag{7}$$

a solution is $\mathbf{y}_t^*$ so that:

$$\mathbf{y}_t^* \quad = \quad \mathbf{F}(\mathbf{y}_{t+u} \cdots \mathbf{y}_t^* \cdots \mathbf{y}_{t-r}, \mathbf{x}_t \cdots \mathbf{x}_{t-r}) \tag{8}$$

For the un-normalized model:

$$\mathbf{0} \quad = \quad \mathbf{F}(\mathbf{y}_{t+u} \cdots \mathbf{y}_t \cdots \mathbf{y}_{t-r}, \mathbf{x}_t \cdots \mathbf{x}_{t-s}) \tag{9}$$

a solution $\mathbf{y}_t^*$ is

$$\mathbf{0} \quad = \quad \mathbf{G}(\mathbf{y}_{t+u} \cdots \mathbf{y}_t^* \cdots \mathbf{y}_{t-r}, \mathbf{x}_t \cdots \mathbf{x}_{t-r}) \tag{10}$$

Some models can have more than one solution. In this case the solution can depend on the starting point of the solution algorithm.

1.3 Model derivatives

Both for solving and for analyzing the causal structure of a model it can be useful to define different matrices of derivatives for a model $\mathbf{F}()$ like this:

$$\mathbf{A}_t = \quad \frac{\partial \mathbf{F}}{\partial \mathbf{y}_t^T} \qquad \text{Derivatives with respect to current endogeneous variables} \tag{11}$$

$$\tag{12}$$

$$\mathbf{E}_t^i = \quad \frac{\partial \mathbf{F}}{\partial \mathbf{y}_{t-i}^T} \quad i = 1, \cdots, r \;\; \text{Derivatives with respect to lagged endogeneous variables} \tag{13}$$

$$\tag{14}$$

$$\mathbf{D}_t^j = \quad \frac{\partial \mathbf{F}}{\partial \mathbf{y}_{t+j}^T} \quad j = 1, \cdots, u \;\; \text{Derivatives with respect to leaded endogeneous variables} \tag{15}$$

$$\tag{16}$$

$$\mathbf{F}_t^k = \quad \frac{\partial \mathbf{F}}{\partial \mathbf{x}_{t-i}^T} \quad k = 0, \cdots, s \;\; \text{Derivatives with respect to current and lagged exogeneous variables}$$

$$\tag{17}$$

$$\tag{18}$$

For un-normalized models the derivative matrices are just the dervatives of $\mathbf{G}$ instead of $\mathbf{F}$

1.4 Model solutions

There are numerous methods to solve models (systems) as mentioned above. ModelFlow can apply 3 different types of model solution methods:

1. If the model has **no contemporaneous feedback**, the equations can be sorted Topological and then the equations can be calculated in the topological order. This is the same as a spreadsheet would do.

2. If the model has **contemporaneous feedback** model is solved with an iterative method. Here variants of well known solution methods are used:
   1. Gauss-Seidle (**Gauss**) which can handle large systems, is fairly robust and don't need the calculation of derivatives
   2. Newthon-Raphson (**Newton**) which requires the calculation of derivatives and solving of a large linear system but typically converges in fewer iterations.

Nearly all of the models solved by ModelFlow don't contain leaded endogenous variables. Therefor they can be solved one period at a time. For large sparse nonlinear models Gauss works fine. It solves a model quite fast and we don't need the additional handiwork of handling derivatives and large linear systems that Newton methods require. Moreover many models in question do not have smooth derivatives. The order in which the equation are calculated can have a large impact on the convergence speed.

For some models the Newton algorithm works better. Some models are not able to converge with Gauss-Seidle other models are just faster using Newton. Also the ordering of equations does not matter for the convergence speed.

However some models like FRB/US and other with **rational expectations** or **model consistent expectations** contains leaded endogenous variables. Such models typical has to be solved as one system for for all projection periods. In this case, the Gauss variation Fair-Taylor or Stacked-Newton Method. The **stacked Newton** methods can be used in all cases, but if not needed, it will usually use more memory and be slower.

Model	No contemporaneous feedback	Contemporaneous feedback	Leaded variables
Normalized	Calculate	Gauss or Newton	Fair Taylor or Stacked Newton
Un-normalized	Newton	Newton	Stacked Newton

2 Implementation of solving algorithms in Python

Solving a model entails a number of steps:

1. Specification of the model
2. Create a dependency graph.
3. Establish a solve order and separate the the model into smaller sub-models
4. Create a python function which can evaluating $f_i(y_1^k, \cdots, y_{i-1}^k, y_{i+1}^{k-1}, \cdots, y_n^{k-1}, z)$
5. If needed, create a python function which can evaluate the Jacobimatrices: $\mathbf{A}, \mathbf{E}, \mathbf{D}$ or $\mathbf{A}, \mathbf{E}, \mathbf{D}$
6. Apply a solve function using the elements above to the data.

2.0.1 Normalized model

Calculation, No contemporaneous feedback In systems with no lags each period can be solved in succession The equations has to be evaluated in a logical (topological sorted) order.

Let z be all predetermined values: all exogenous variable and lagged endogenous variable.

Order the n endogeneous variables in topological order.

For each time period we can find a solution by

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

The Gauss-Seidel algorithm. Normalized models with contemporaneous feedback 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. n be the number of endogenous variables. α dampening factor which can be applyed to selected equations

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

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

The Newton-Raphson algorithme. Models with contemporaneous feedback Let $\mathbf{z}$ be a vector all predetermined values: all exogenous variable and lagged endogenous variable.

For each time period we can find a solution by doing Newton-Raphson iterations:

```
for $k = 1$ to convergence
    $\mathbf{y} = \mathbf{F}(\mathbf{y^{k-1}}, \mathbf{z})$
    $\mathbf{y^k} = \mathbf{y} - \alpha \times (\mathbf{A} - \mathbf{I})^{-1} \times (\mathbf{y} - \mathbf{y^{k-1}})$
```

The expression: $(\mathbf{A} - \mathbf{I})^{-1} \times (\mathbf{y} - \mathbf{y^{k-1}})$ is the same as the solution to:

$\mathbf{y} - \mathbf{y^{k-1}} = (\mathbf{A} - \mathbf{I}) \times \mathbf{x}$

This problem can be solved much more efficient than performing $(\mathbf{A} - \mathbf{I})^{-1} \times (\mathbf{y} - \mathbf{y^{k-1}})$

The Scipy library provides a number of solvers to this linear set of equations. There are both solvers using factorization and iterative methods, and there are solvers for dense and sparce matrices. All linear solvers can easily be incorporated into ModelFlows Newton-Raphson nonlinear solver.

Stacked Newton-Raphson all periods in one go. Models with both leaded and lagged endogeneous variable If the model has leaded endogenous variables it can in general not be solved one time period at a time. We have to solve the model for all time frames as one large model.

$$\mathbf{A}=\begin{bmatrix}\mathbf{A_1} & \mathbf{D_1^1} & \mathbf{D_1^2} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0}\\ \mathbf{E_2^1} & \mathbf{A_2} & \mathbf{D_2^1} & \mathbf{D_2^2} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0}\\ \mathbf{E_3^2} & \mathbf{E_3^1} & \mathbf{A_3} & \mathbf{D_3^1} & \mathbf{D_3^2} & \mathbf{0} & \mathbf{0} & \mathbf{0}\\ \mathbf{E_4^3} & \mathbf{E_4^2} & \mathbf{E_4^1} & \mathbf{A_4} & \mathbf{D_4^1} & \mathbf{D_4^2} & \mathbf{0} & \mathbf{0}\\ \mathbf{0} & \mathbf{E_5^3} & \mathbf{E_5^2} & \mathbf{E_5^1} & \mathbf{A_5} & \mathbf{D_5^1} & \mathbf{D_5^2} & \mathbf{0}\\ \mathbf{0} & \mathbf{0} & \mathbf{E_6^3} & \mathbf{E_6^2} & \mathbf{E_6^1} & \mathbf{A_6} & \mathbf{D_6^1} & \mathbf{D_6^2}\\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{E_7^3} & \mathbf{E_7^2} & \mathbf{E_7^1} & \mathbf{A_7} & \mathbf{D_7^1}\\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{E_8^3} & \mathbf{E_8^2} & \mathbf{E_8^1} & \mathbf{A_8}\end{bmatrix} \mathbf{y}=\begin{bmatrix}\mathbf{y_1}\\ \mathbf{y_2}\\ \mathbf{y_3}\\ \mathbf{y_4}\\ \mathbf{y_5}\\ \mathbf{y_6}\\ \mathbf{y_7}\\ \mathbf{y_8}\end{bmatrix} \mathbf{F}=\begin{bmatrix}\mathbf{F}\\ \mathbf{F}\\ \mathbf{F}\\ \mathbf{F}\\ \mathbf{F}\\ \mathbf{F}\\ \mathbf{F}\\ \mathbf{F}\end{bmatrix}$$

Now the solution algorithme looks like this.

Again let $\mathbf{z}$ be a vector all predetermined values: all exogenous variable and lagged endogenous variable.

$$\text{for } k = 1 \text{ to convergence } > \mathbf{y} = \mathbf{F}(\mathbf{y^{k-1}}, \mathbf{z}) > \mathbf{y^k} = \mathbf{y} - \alpha \times (\mathbf{A} - \mathbf{I})^{-1} \times (\mathbf{y} - \mathbf{y^{k-1}})$$

Notice that the model $\mathbf{F}$ is the same for all time periods. However, as time can be an exogenous variable the result of $\mathbf{F}$ can depend on time. This allows us to specify terminal conditions.

The update frequency of $\mathbf{A}$ and α and the value of α can be set to manage the speed and stability of the algorithm.

We solve the problem:

$$(\mathbf{y} - \mathbf{y^{k-1}}) = (\mathbf{A} - \mathbf{I}) \times \mathbf{x}$$

instead of inverting $\mathbf{A}$.

Python gives access to very efficient sparse libraries. The Scipy library utilizes the Intel® Math Kernel Library. Any of the available routines for solving linear systems can easily be incorporated.

2.1 Create a model instance which calculates the Jacobi matrices.

The derivatives of all formulas with respect to all endogenous variables are needed.

First step is to specifying a model in the business logic language which calculate all the non-zero elements In ModelFlow this can be done by **symbolic**, by **numerical differentiation** or by a combination.

The formula for calculating $\dfrac{\partial numerator}{\partial denominator(-lag)}$ is written as:

< numerator >___p___< denominator >___lag___< lag> = derivative expression

Just another instance of a ModelFlow model class.

2.1.1 A small Solow model to show the construction of the Jacobi matrix.

An example can be helpful First a small model is defined - in this case a solow growth model:

[43]:
```
fsolow = '''\
Y         = a * k**alfa * l **(1-alfa)
C         = (1-SAVING_RATIO)  * Y
I         = Y - C
diff(K)   = I-depreciates_rate * K(-1)
diff(l)    = labor_growth * L(-1)
K_i= K/L '''
msolow = model.from_eq(fsolow)
```

[44]:
```
print(msolow.equations)
```

```
FRML <> Y          = A * K**ALFA * L **(1-ALFA)  $
FRML <> C          = (1-SAVING_RATIO)  * Y  $
FRML <> I          = Y - C  $
FRML <> K=K(-1)+(I-DEPRECIATES_RATE * K(-1))$
FRML <> L=L(-1)+(LABOR_GROWTH * L(-1))$
FRML <> K_I= K/L  $
```

2.1.2 Create some data and solve the model

[45]:
```
N = 100
df = pd.DataFrame({'L':[100]*N,'K':[100]*N})
df.loc[:,'ALFA'] = 0.5
df.loc[:,'A'] = 1.
df.loc[:,'DEPRECIATES_RATE'] = 0.05
df.loc[:,'LABOR_GROWTH'] = 0.01
df.loc[:,'SAVING_RATIO'] = 0.05
msolow(df,max_iterations=100,first_test=10,silent=1);
```

2.1.3 Create an differentiation instance of the model

Use symbolic differentiation when possible else use numerical differentiation.

[46]:
```
from modelnewton import newton_diff
msolow.smpl(3,5);   # we only want a few years
```

2.1.4 Symbolic differentiation

[47]:
```
newton = newton_diff(msolow)
print(newton.diff_model.equations)
```

```
FRML  <> C__p__Y___lag___0 = 1-SAVING_RATIO   $
FRML  <> I__p__C___lag___0 = -1   $
FRML  <> I__p__Y___lag___0 = 1   $
FRML  <> K__p__I___lag___0 = 1   $
FRML  <> K__p__K___lag___1 = 1-DEPRECIATES_RATE   $
FRML  <> K_I__p__K___lag___0 = 1/L   $
FRML  <> K_I__p__L___lag___0 = -K/L**2   $
FRML  <> L__p__L___lag___1 = LABOR_GROWTH+1   $
FRML  <> Y__p__K___lag___0 = A*ALFA*K**ALFA*L**(1-ALFA)/K   $
FRML  <> Y__p__L___lag___0 = A*K**ALFA*L**(1-ALFA)*(1-ALFA)/L   $
```

2.1.5 Numerical differentiation

```
[48]: newton2 = newton_diff(msolow,forcenum=1)
      print(newton2.diff_model.equations)
```

```
FRML  <> C__p__Y___lag___0 =
(((1-SAVING_RATIO)*(Y+0.0025))-((1-SAVING_RATIO)*(Y-0.0025)))/0.005   $
FRML  <> I__p__C___lag___0 = ((Y-(C+0.0025))-(Y-(C-0.0025)))/0.005   $
FRML  <> I__p__Y___lag___0 = (((Y+0.0025)-C)-((Y-0.0025)-C))/0.005   $
FRML  <> K__p__I___lag___0 = ((K(-1)+((I+0.0025)-DEPRECIATES_RATE*K(-1)))-(K(-1)
+((I-0.0025)-DEPRECIATES_RATE*K(-1))))/0.005   $
FRML  <> K__p__K___lag___1 = (((K(-1)+0.0025)+(I-DEPRECIATES_RATE*(K(-1)+0.0025)
))-((K(-1)-0.0025)+(I-DEPRECIATES_RATE*(K(-1)-0.0025))))/0.005   $
FRML  <> K_I__p__K___lag___0 = (((K+0.0025)/L)-((K-0.0025)/L))/0.005   $
FRML  <> K_I__p__L___lag___0 = ((K/(L+0.0025))-(K/(L-0.0025)))/0.005   $
FRML  <> L__p__L___lag___1 = (((L(-1)+0.0025)+(LABOR_GROWTH*(L(-1)+0.0025)))-((L
(-1)-0.0025)+(LABOR_GROWTH*(L(-1)-0.0025))))/0.005   $
FRML  <> Y__p__K___lag___0 =
((A*(K+0.0025)**ALFA*L**(1-ALFA))-(A*(K-0.0025)**ALFA*L**(1-ALFA)))/0.005   $
FRML  <> Y__p__L___lag___0 =
((A*K**ALFA*(L+0.0025)**(1-ALFA))-(A*K**ALFA*(L-0.0025)**(1-ALFA)))/0.005   $
```

2.1.6 Display the full stacked matrix

To make the sparcity clear all zero values are shown as blank

```
[49]: stacked_df = newton.get_diff_df_tot()
      stacked_df.applymap(lambda x:f'{x:,.2f}' if x != 0.0 else ' ')
```

```
[49]: per          3                                         4                        \
      var          C      I      K    K_I      L      Y      C      I      K    K_I
      per var
      3   C    -1.00                                0.95
          I    -1.00  -1.00                         1.00
          K             1.00  -1.00
          K_I                  0.01  -1.00  -0.01
          L                                -1.00
          Y                    0.51          0.49  -1.00
      4   C                                               -1.00
          I                                               -1.00  -1.00
          K                    0.95                               1.00  -1.00
          K_I                                                            0.01  -1.00
          L                                  1.01
          Y                                                              0.51
      5   C
          I
          K                                                              0.95
          K_I
          L
```

```
    Y

per                        5
var          L      Y      C      I      K    K_I      L      Y
per var
3   C
    I
    K
    K_I
    L
    Y
4   C           0.95
    I           1.00
    K
    K_I  -0.01
    L    -1.00
    Y     0.49  -1.00
5   C                  -1.00                               0.95
    I                  -1.00  -1.00                        1.00
    K                          1.00  -1.00
    K_I                               0.01  -1.00  -0.01
    L     1.01                                     -1.00
    Y                                 0.51          0.49  -1.00
```

2.2 Speeding up solving through Just In Time compilation (Numba)

Python is an interpreted language. So slow

Numba is a Just In time Compiler. Experience with a Danish model (1700 equations) shows a speedup from 5 million floating point operations per second (MFlops) to 800 MFlops. But compilation takes time.

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.

Also matrices can be used. This will force the use of the highly optimized routines in the Numpy library.

2.3 Specification of model in Business Logic Language

The Business logic Language is a Python like language, where each function f_i from above is specified as:

`FRML <options> <left hand side> = <right hand side> $ ...`

The `<left hand side>` should not contain transformations, but can be a tuple which match the `<right hand side>`. A $ separates each formular.

Time is implicit, so var_t is written as `var`, while var_{t-1} is written as `var(-1)` and var_{t+1} is written as `var(+1)`. Case does not matter. everything is eventual made into upper case.

It is important to be able to create short and expressive models, therefor. Stress test models should be able to handle many bank and sectors without repeating text. So on top of the **Business logic language**. there is a **Macro Business Logic language**. The primary goal of this is to allow (conditional) looping and normalization of formulas.

The user can specify any conforming python function on the right hand side

3 Onboarding a model

**Python has incredible strong tools both for interacting with other systems like Excel and Matlab. Some of the sources, from which models has been recycled are:

- Latex
  - Model written in Latex - with some rules to allow text processing.
- Eviews
- 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.

3.1 The model structure

The logical structure of a model is useful for several reasons.

The structure of **contemporaneous endogenous variable** is used to establish the calculation sequence and identify simultaneous systems (strong graphcomponents).

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 b to a.

3.1.1 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

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

Naturally: $E \subseteq S$

3.1.2 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

3.1.3 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.

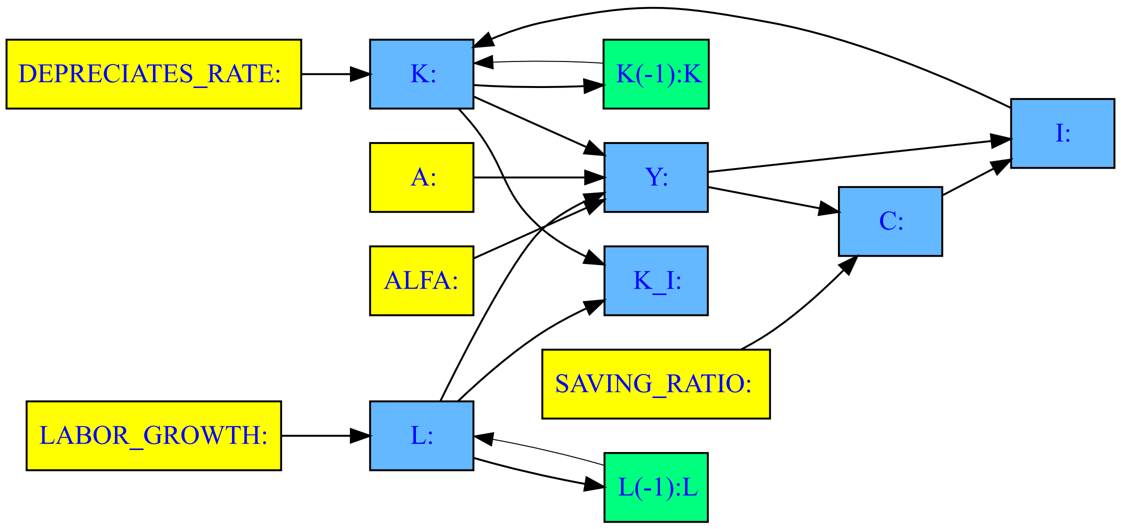
If TE, the dependency graph associated with F is **not** a Directed Acyclical Graph (A DAG). Then F has contemporaneous feedback and is simultaneous. Or - in Excel speak - the model has circular references. And we need to use an iterative methods to solve the model. Sometime a model contains several 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.

Total dependency graphs This shows TG mentioned above.

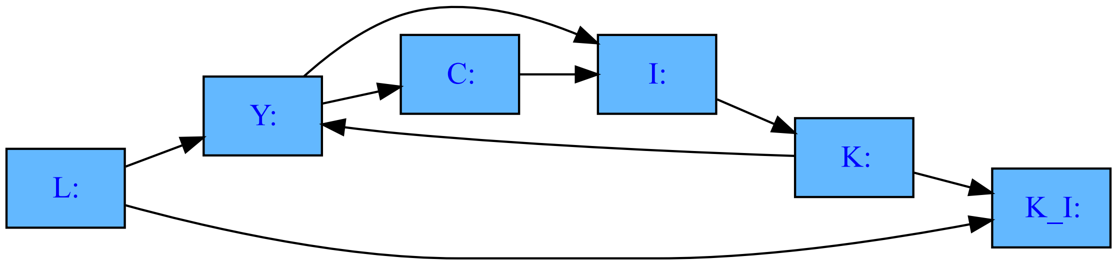
[50]:
```
msolow.drawmodel(title='Total graph',all=0)
```



Total graph

3.1.4 The dependency graph for contemporaneous endogenous variables (TE)

[51]:
```
msolow.drawendo(title='Contemporaneous endo')
```

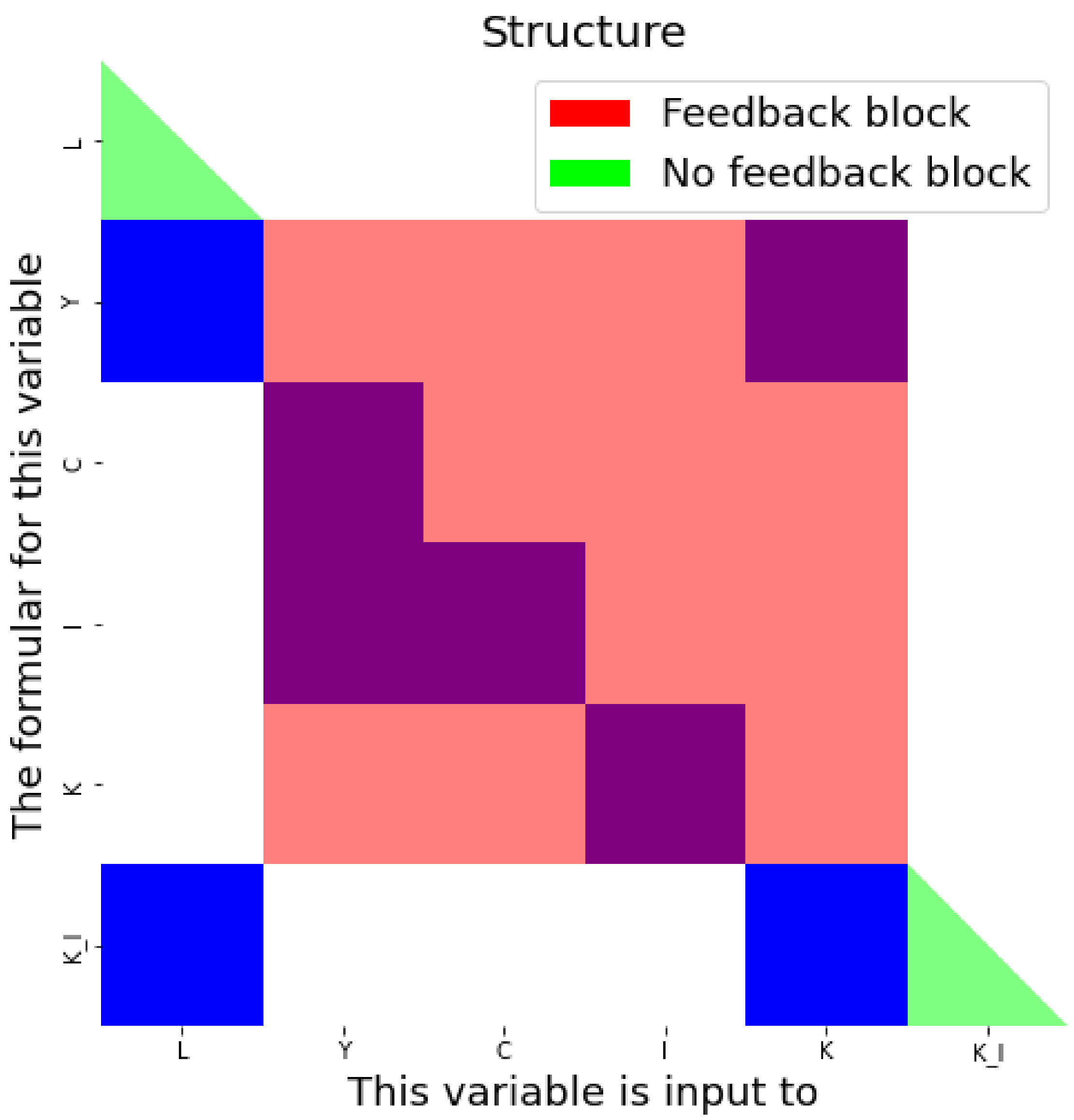


Contemporaneous endo

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.

[52]:
```
a = msolow.plotadjacency(size=(8,8))
```



3.2 Solution ordering

3.2.1 For normalized models:

For a model **without contemporaneous feedback**, the topoligical sorted order is then used as calculating order.

For a model **with contemporaneous feedback** and no leaded variables, ModelFlow divides a model into three parts. A recursive **prolog** model, a recursive **epilog** model, the rest is the simultaneous **core** model. Inside the core model the ordering of the equations are preserved. It may be that the core model contains several strong componens, which each could be solved as a simultanous system, however it is solved as one simultanous system.

Only the core model is solved as a simultaneous system. The prolog model is calculated once before the solving og the simultaneous system, the epilog model is calculated once after the solution of the simultanous system. For most models this significantly reduce the computational burden of solving the model.

For a model with leaded variables where the model is stacked. All equations are created equal.

[53]:
```python
# The preorder
print(f'The prolog variables {msolow.preorder}')
print(f'The core   variables {msolow.coreorder}')
print(f'The epilog variables {msolow.epiorder}')
```

```
The prolog variables ['L']
The core   variables ['Y', 'C', 'I', 'K']
The epilog variables ['K_I']
```

4 Some Model manipulation capabilities

4.1 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 ?
- What probability of transmission result in infected 2 weeks later

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: $\mathbf{y}_t = \mathbf{F}(\mathbf{x}_t)$. Here $\mathbf{x}_t$ are all predetermined variables - lagged endogenous and all exogenous variables.

It can be useful to allow a **delay**, when finding the instruments. In this case we want to look at $\mathbf{y}_t = \mathbf{F}(\mathbf{x}_{t-delay})$

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

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

The approximated Jacobi matrix of $\mathbf{G}: \mathbf{J}_t \approx \frac{\Delta \mathbf{G}}{\Delta \bar{\mathbf{x}}_{t-delay}}$ is used to find the instruments

4.1.1 And how to solve for the instruments

For most models $\bar{\mathbf{x}}_{t-delay} = \mathbf{G}^{-1}(\bar{\mathbf{y}}_t)$ do not have a nice close form solution. However it can be solved numerically. We turn to Newton–Raphson method.

So $\bar{\mathbf{x}}_{t-delay} = \mathbf{G}^{-1}(\bar{\mathbf{y}_t^*})$ will be found using :

for $k = 1$ to convergence

$$\bar{\mathbf{x}}_{t-delay,end}^{k} = \bar{\mathbf{x}}_{t-delay,end}^{k-1} + \mathbf{J}_t^{-1} \times (\bar{\mathbf{y}_t^*} - \bar{\mathbf{y}_t}^{k-1})$$

$$\bar{\mathbf{y}}_t^k = \mathbf{G}(\bar{\mathbf{x}}_{t-delay}^k)$$

convergence: $\mid \bar{\mathbf{y}_t^*} - \bar{\mathbf{y}}_t \mid \leq \epsilon$

Now we just need to find:

$\mathbf{J}_t = \frac{\partial \mathbf{G}}{\partial \bar{\mathbf{x}}_{t-delay}}$

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

$\mathbf{J}_t \approx \frac{\Delta \mathbf{G}}{\Delta \bar{\mathbf{x}}_{t-delay}}$

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.

You will notice that the level of $\bar{\mathbf{x}}$ is updated (by $\mathbf{J}_t^{-1} \times (\bar{\mathbf{y}_t^*} - \bar{\mathbf{y}_t}^{k-1})$) in all periods from $t - delay$ to *end*, where *end* is the last timeframe in the dataframe. This is useful for many applications including calibration of disease spreading models and in economic models, where the instruments are level variable (i.e. not change variables). If this is not suitable, it can be changed in a future release.

The target/instrument functionality is implemented in the python class `targets_instruments` specified in **ModelFlows** `modelinvert` module.

4.1.2 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.

```
[54]: msolow.basedf
```

[54]:
```
             L           K  ALFA    A  DEPRECIATES_RATE  LABOR_GROWTH  \
0   100.000000  100.000000   0.5  1.0              0.05          0.01
1   101.000000  100.025580   0.5  1.0              0.05          0.01
2   102.010000  100.076226   0.5  1.0              0.05          0.01
3   103.030100  100.151443   0.5  1.0              0.05          0.01
4   104.060401  100.250762   0.5  1.0              0.05          0.01
..         …           …   …  …               …           …
95  257.353755  185.913822   0.5  1.0              0.05          0.01
96  259.927293  187.661027   0.5  1.0              0.05          0.01
97  262.526565  189.428077   0.5  1.0              0.05          0.01
98  265.151831  191.215119   0.5  1.0              0.05          0.01
99  267.803349  193.022302   0.5  1.0              0.05          0.01

    SAVING_RATIO           Y           C          I       K_I
0           0.05    0.000000    0.000000   0.000000  0.000000
1           0.05  100.511609   95.486029   5.025580  0.990352
2           0.05  101.038487   95.986562   5.051924  0.981043
3           0.05  101.580575   96.501546   5.079029  0.972060
4           0.05  102.137821   97.030930   5.106891  0.963390
..           …         …         …        …       …
95          0.05  218.736417  207.799596  10.936821  0.722406
96          0.05  220.857924  209.815028  11.042896  0.721975
97          0.05  223.002024  211.851922  11.150101  0.721558
98          0.05  225.168912  213.910466  11.258446  0.721153
99          0.05  227.358789  215.990850  11.367939  0.720761

[100 rows x 11 columns]
```

Define Targets

[55]:
```
target = msolow.basedf.loc[50:,['L','K']]+[30,10]
target.head()
```

[55]:
```
             L           K
50  194.463182  135.971544
51  196.107814  136.933236
52  197.768892  137.911105
53  199.446581  138.905161
54  201.141047  139.915414
```

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 an 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.

Define Instruments

[56]:
```
instruments = [ [('LABOR_GROWTH',0.001)] , [('DEPRECIATES_RATE',0.001)]]
```

Run the experiment

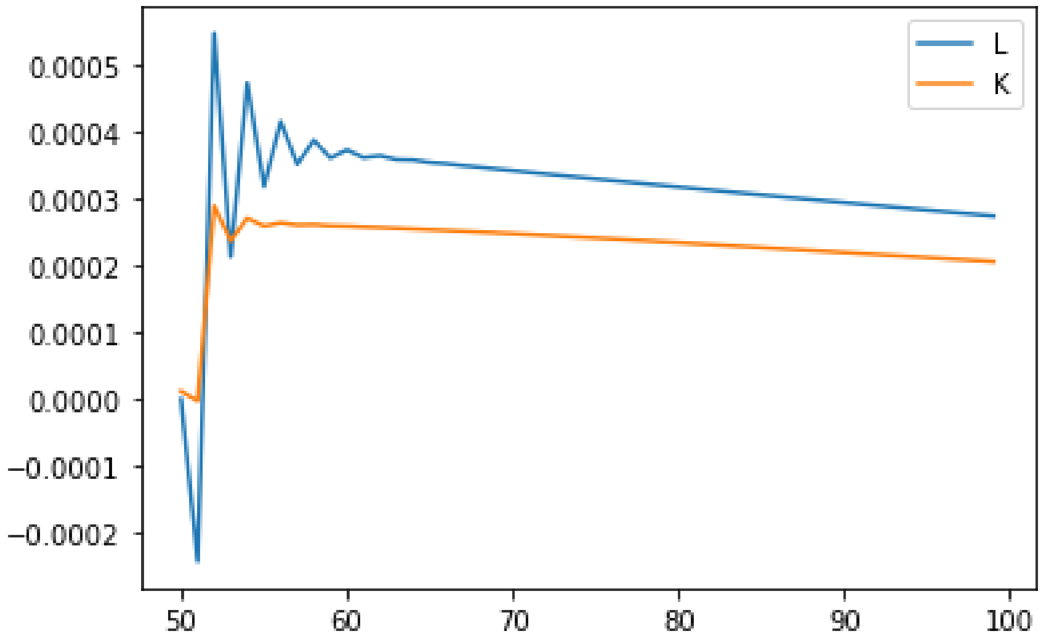
For models which are relative linear we don't need to update for each iteration and time frame. As our small toy model is nonlinear, the jacobi matrix has to be updated frequently. This is controlled by the nonlin=True option below.

[57]:
```
result = msolow.invert(msolow.lastdf,target,instruments,nonlin=True)
```

```
Finding instruments :   0%|          | 0/50
```

And do the result match the target?

[58]:
```
(result-target).loc[50:,['L','K']].plot();
```



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

[59]:
```
msolow.smpl(90,100)  # we only want a few years

msolow.basedf.Y
```

[59]:
```
0      0.000000
1    100.511609
2    101.038487
3    101.580575
4    102.137821
```

```
          …
95    218.736417
96    220.857924
97    223.002024
98    225.168912
99    227.358789
Name: Y, Length: 100, dtype: float64
```

[60]:
```
msolow.lastdf.Y
```

[60]:
```
0       0.000000
1     100.511609
2     101.038487
3     101.580575
4     102.137821
          …
95    237.269232
96    239.389730
97    241.532869
98    243.698846
99    245.887858
Name: Y, Length: 100, dtype: float64
```

4.1.3 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.

4.2 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:

$$\begin{aligned} y_0 &= f(a_0, b_0) && (19) \\ y_1 &= f(a_0 + \Delta a, b_0 + \Delta b) && (20) \end{aligned}$$

So Ωa and Ωb can be determined:

$$\begin{aligned} \Omega f_a &= f(a_1, b_1) - f(a_1 - \Delta a, b_1) && (21) \\ \Omega f_b &= f(a_1, b_1) - f(a_1, b_1 - \Delta b) && (22) \end{aligned}$$

And:

$$residual = \Omega f_a + \Omega f_b - (y_1 - y_0) \quad (23)$$

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

4.2.1 Formula attribution

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

4.2.2 Model Attribution

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

4.3 Python functions can be incorporated

4.3.1 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:

$$
\begin{align}
\mathbf{x} & \quad \text{Position in each asset(+)/liability(-) type} \tag{24}\\
\mathbf{x} & \quad \text{Position in each asset(+)/liability(-) type} \tag{25}\\
 & \quad \text{Covariance matrix} \tag{26}\\
\mathbf{r} & \quad \text{Return vector} \tag{27}\\
\lambda & \quad \text{Risk aversion} \tag{28}\\
\mathbf{riskweights} & \quad \text{Vector of risk weights, liabilities has riskweight} = 0 \tag{29}\\
Capital & \quad \text{Max of sum of risk weighted assets} \tag{30}\\
\mathbf{lcrweights} & \quad \text{Vector of LCR weights, liabilities has lcrweight} = 0 \tag{31}\\
LCR & \quad \text{Min of sum of lcr weighted assets} \tag{32}\\
\mathbf{leverageweight} & \quad \text{Vector of leverage weights, liabilities has leverageweight} = 0 \tag{33}\\
Equity & \quad \text{Max sum of leverage weighted positions} \tag{34}\\
Budget & \quad \text{initial sum of the positions} \tag{35}\\
 & \tag{36}
\end{align}
$$

$$
\begin{align}
\text{minimize:} \quad & \lambda \mathbf{x}^T\ \mathbf{x} - (1-\lambda)\mathbf{r}^T\mathbf{x} && \text{If } \lambda = 1 \text{ minimize risk, if } \lambda = 0 \text{ maximize return} \tag{37}\\
\text{subject to:} \quad & \mathbf{x} \succeq \mathbf{x}^{\mathbf{min}} && \text{Minimum positions} \tag{38}\\
& \mathbf{x} \preceq \mathbf{x}^{\mathbf{max}} && \text{Maximum positions} \tag{39}\\
& \mathbf{riskweights}^T\mathbf{x} \leq Capital && \text{Risk weighted assets <= capital} \tag{40}\\
& \mathbf{lcrweights}^T\mathbf{x} \geq LCR && \text{lcr weighted assets >= LCR target} \tag{41}\\
& \mathbf{leverageweight}^T\mathbf{x} \leq equity && \text{leverage weighted assets <= equity} \tag{42}\\
& \mathbf{1}^T\mathbf{x} = Budget && \text{Sum of positions = B} \tag{43}
\end{align}
$$

4.3.2 The mean variance problem in the business language language

Wrap optimizing in the CVX library into a Python function: 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.

4.4 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.

4.5 Live models

Showtime

5 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 m,odels
- 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

6 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

E. Petersen, Christian & A. Sims, Christopher. (1987). Computer Simulation of Large-Scale Econometric Models: Project Link. International Journal of High Performance Computing Applications (http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.853.6387&rep=rep1&type=pdf)

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

7 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

```
[ ]:
```