The workflow of building SFC model of the Polish economy

Julian Kacprzak Iwo Augustyński

Table of contents

The main goal of this document is to present the workflow developed and used in creating an empirical SFC model for Poland. It is developed entirely in the R language and, in addition to standard packages for data manipulation, uses the following libraries:

- the SFC `godley` for modeling,
- `eurostat` for downloading data from the Eurostat database,
- and the libraries for transformation and estimation of variables `tempdisagg` and `dynlm`.

We believe that the presented method of working on empirical SFC models will make it easier for other researchers to work on the models and thus accelerate the development of this field of economics.

The next two chapters describe the structure of scripts used to prepare data and build the model, as well as the recommended procedures. The model itself is a slightly modified copy of the quarterly empirical model of the Danish economy and is not presented here. All scripts and files including model equations are attached in a zip file.

The appendix at the end contains the `godley` library manual.

The main contributor to the workflow is Julian Kacprzak, Iwo Augustyński was responsible for scientific supervision and model builidng.

Full list of people involved in work on the model:

- Iwo Augustyński, Wroclaw University of Economics and Business - scientific supervision and model building,
- Michał Możdżeń, Krakow University of Economics - data preparation,
- Maciej Grodzicki, Jagiellonian University - parameter estimation,
- Julian Kacprzak - software engineer.

1. Main scripts and files

1.1 Data Dictionary

The data dictionary is an xlsx file serving as a complete source of information about the data and equations in the model. The dictionary has been designed as a kind of simple interface for issuing commands to automated processes in the tool written in the R language, which is the subject of next chapters.

The dictionary itself contains all the technical information needed to recreate the model, so it can be thought of as a punched card entered into the R tool. What the user "orders" in the dictionary will be executed without the need to interfere with the source code of the tool. At the same time, the spreadsheet format allows for a comprehensive assessment of the model in a more visually friendly manner than would be possible in an R script or a CSV file.

The dictionary displayed in Microsoft Excel is equipped with three sheets:

1. Variables - a list of all model variables along with information that uniquely identifies them in the Eurostat database,
2. Equations - a list of all equations in the model along with the form required for estimating structural parameters,
3. Parameters - a list of all parameter values estimated during the tool's execution.

1.1.1 Variables

The Variables sheet contains information about variables in the model, including their descriptions, properties, and identification in the Eurostat database. In this sheet, you can find the following columns:

- Id - an ordinal column that sorts variables by sector and alphabetically,
- Name - the variable's name in the model, which should comply with the naming guidelines in the godley package. In addition to the package's technical requirements, the following rules are applied:
  - Names are derived from English abbreviations of variable names, as commonly described in economic literature,
  - Real quantities are written in lowercase, while nominal quantities are in uppercase,
  - Parts of names are separated by underscores "_". If a variable clearly belongs to a specific sector, the sector's designation is the last part of its name, e.g., _H for households,
  - Variables not present in the model but used for custom calculations are marked with _ow (on their own) in their names,
  - No variable is named after an R constant or object, i.e., no variable is named NA, NaN, Inf, pi, etc.,
- Sector_name - the sector to which the variable belongs. For variables related to the overall economy, the sector is Non-Financial Corporations,
- Type - information about whether the variable is endogenous (explained by the model and appears on the left-hand side of equations) or exogenous (only appears as an explanatory variable and is located on the right-hand side of equations). This column can be easily filled in using the info = TRUE argument during simulation in the godley package (see Chapter 2.1),
- Description - a brief description of the variable,
- Link_to_source - a link to the variable's view in the Eurostat data browser,
- Columns from G to AB - these columns contain information that uniquely identifies each variable in the Eurostat database. The online_data_code column contains information about the table from which the variable originates, e.g., nasq_10_nf_tr. Each subsequent column is named after a dimension present in that table, and it contains values

of that dimension corresponding to the variable. It should be noted that variables from different tables are present in the dictionary, and some dimension names are common across tables while others are not. To maintain the tabular format of the dictionary, dimensions that do not exist for a particular variable are still included but left empty. Eurostat presents dimension values in two variants: label and code - in the dictionary, the second variant (code) is used,

- freq_td (frequency temporal disaggregation) - information about which function to use for disaggregating an annual variable into quarterly data. Since some variables are only available on an annual basis, the tool automatically performs temporal disaggregation. However, for each annual variable (column freq = A), a transition function must be specified. The names of these functions should come directly from the argument values of the `tempdisagg::td()` function. Currently, these values are: sum, mean, first, and last,
- Calc - not all desired variables are present in the Eurostat database. The calculations column allows creation of a new variable based on Eurostat data series. There are two calculation variants:

1. Combining variables within the same online_data_code table: To combine variables from the same table but different dimensions, calculations should use dimension values as variable names while leaving the column of the merged dimension empty (from G to AB). For example, if you want to calculate the net value (assets - liabilities), you should leave the column from which these values originate (e.g., finpos) empty and enter ASS - LIAB in the calculation field. When combining multiple dimensions in variable names, use underscores "_" to separate them. For example, the net value for two different national account categories F1 and F2 can be written as ASS_F1 + ASS_F2 - LIAB_F1 - LIAB_F2. Both columns from which values originate should be left empty from G to AB. The order of elements in variable names with multiple dimensions is significant and should correspond to the order of columns in the dictionary. For the nasa_10_f_gl table, the correct format is: ASS_F1, while F1_ASS is incorrect. This variant is useful when both the online_data_code column and Calc column are non-empty.
2. Combining variables from different online_data_code tables: To combine variables from different tables, calculations should use the names of other variables present in the dictionary. If a variable does not exist in the model, it can only be created in the dictionary and not included in equations. According to the convention applied, the name of such a variable will receive the suffix _ow (on their own). If you want to create variable X as the sum of series from two different tables, you can create two variables for your calculations: X_ow_1 and X_ow_2, and enter X_ow_1 + X_ow_2 in the calculation field for variable X. Of course, you can also use the names of variables already present in the model. This variant is applicable when the online_data_code column is empty, and the Calc column is not empty. In both cases of custom calculations, formulas should be constructed using standard R operators and functions defined for the columns of a data frame. The formula will be evaluated within the `dplyr::mutate()` function on a data frame with a time column and columns named according to each of the two variants men-

tioned above. Therefore, any expression that meets these requirements can be placed in the calculation column. It is worth noting the use of the `diff()` function in calculations. This function is defined on a vector and, therefore, on a data frame column, but it returns a vector shorter than the original one (depending on the degree of difference). Thus, if you enter `dplyr::mutate(new_variable = diff(old_variable))`, you will get an error. In calculations, you can write `c(NA, diff(old_variable))` - which is correctly defined in the `dplyr::mutate()` function. Another illustrative example is the use of a logical condition in calculations in combination with the time column, which is present in every data frame and named time. The expression `ifelse(time < '2020-01-01', one_variable, another_variable)` is fully valid and will yield different variables before and after the first quarter of 2020,

- Start of Series and End of Series - the time range of variables in the Eurostat database,
- Comment - a user comment field.

1.1.2 Equations

The Equations sheet contains a list of model equations. In this sheet, you can find the following columns:

- Id - an ordinal column that sorts equations by sector and alphabetically,
- Equation - the equation written in a format that will be passed to the simulation in the godley package, thus meeting all the technical requirements of the package (see Appendix). It's worth mentioning the rule regarding variable signs applied during model construction. All variables in the data frame for the model are stored with a positive sign, similar to the Eurostat database. The sign is added in the equation itself based on whether, for example, they represent a sector's liabilities or assets. The only variables in which information about the direction of the operation is stored are variables representing net values. They are designed for a specific sector or entity and do not require an additional sign in the equation,
- Param_equation - the equation written in a form understood by the `dynlm::dynlm()` function. This pertains to parametric equations to be estimated. Filling in this column defines the equation as parametric throughout the tool's operation. Since parameter estimation occurs outside the SFC model and the godley package, parametric equations must be written as R language formulas, calculated directly by the `dynlm::dynlm()` function. The differences are as follows:
  - On the left-hand side, expressions may appear, not just variable names, e.g., the logarithm of variable c: `log(c)`,
  - The tilde sign "~" separates the left and right sides of the equation,
  - On the right-hand side of the equation, expressions that should be treated as single explanatory variables should be enclosed in the I() function, for example, the ratio of variables x and y: I(x/y),

  - Explanatory variables that should be multiplied by estimated parameters do not require the inclusion of these parameters in the equation; that is, the equation y ~ x + z is understood as y ~ parameter_1x + *parameter_2*z, – The absence of the intercept term in the formula is indicated by the expression + 0 added at the end of the equation,
  - The `dynlm::dynlm()` function has its own dynamic column operations functions. For example, for variable x:
    * Delay in R language: `lag(x)`, delay in the godley package: `x[-1]`, delay in the `dynlm::dynlm()` function: `L(x)`
    * Difference in R language: `diff(x)`, difference in the godley package: `d(x)`, difference in the `dynlm::dynlm()` function: `d(x)`
    * Delayed difference in R language: `lag(diff(x))`, delayed difference in the godley package: `d(x[-1])`, delayed difference in the `dynlm::dynlm()` function: `L(d(x))`.

    It's worth mentioning that due to the manual and semi-automatic parameter estimation method in the tool, formulas for parametric equations will be overwritten in the dictionary by new formulas chosen by the user. However, formulas in the Equation column will not be overwritten, and users need to update the representation of equations intended for calculation in the SFC model in the godley package each time they change parametric equations,

- Sector - the sector to which the equation belongs (consistent with the sectors of variables in the Variables sheet),
- Comment - a user comment field.

1.1.3 Parameters

The Parameters sheet contains a list of structural parameter estimates for the model. This sheet is automatically created and completely overwritten in case of changes in parameter values during the tool's execution. In this sheet, you can find the following columns:

- Name - the name of the structural parameter created according to the rule: par_ + the name of the explained variable + parameter index. For example, if an equation explains a variable named T_N, the first parameter in this equation will be named par_T_N_1, the second parameter will be named par_T_N_2, and so on. If an estimated formula contains an intercept term, it will always be given the first index. For example, if the equation explaining variable c is estimated with an intercept term, it will be labeled as par_c_1,
- Parameter - the value of the parameter,
- Explanatory_Variable - the name of the variable or the entire expression where the parameter stands in the equation (with a multiplication sign). Intercept terms are marked as (Intercept) here. If the explanatory variable or expression appears in more than one

equation, an index in the form of consecutive natural numbers will be added to it for the second and subsequent occurrences,
- Comment - a user comment field.

1.2 Data Download

In this script, automatic data retrieval from the Eurostat database is implemented. Input Data:

- dict_variables - the Variables sheet from the dictionary described above. Output Data:
- data_raw - a dataframe of raw time series from the Eurostat database containing variables defined in the Variables tab of the dictionary.

Procedure: Apart from configuring the environment and loading input data, the script consists of four parts:

1. Retrieving all used Eurostat tables - in this part, all necessary Eurostat tables (all present in the online_data_code column of the Variables tab) are retrieved. The tables are saved under their names in the global environment using the `eurostat::get_eurostat()` function. Eurostat provides two methods of retrieving tables: in their entirety (slower solution) or with user-imposed filters (faster solution) - you can read more about it here. However, the filtered method has a data size limit, making it not successful for every combination of filters. The implemented automatic download in the tool will attempt to download subsequent tables with filters (created based on the Variables tab of the dictionary). If this method fails, the table will be downloaded in its entirety.
2. Creating raw variables from Eurostat tables - in this part, variables present in the Eurostat database are created in the form desired by the user, i.e., variables unambiguously defined in the Eurostat dimension columns with an empty calculations column. Each row of the dictionary corresponding to such a variable is combined with the appropriate table using the `merge()` function, which filters the table for that variable. Finally, variables available only in annual frequency are disaggregated to quarterly frequency using the `tempdisagg:td()` function.
3. Custom calculations - in this part, variables are created based on the calculations column from the dictionary and the raw time series created in the previous part. As discussed, custom calculations come in two variants, which are the subject of two sections:

a. Custom calculations within the same Eurostat table - to create variables within this variant, the `custom_variable()` function is used. This function creates a dataframe with columns corresponding to the values of calculated dimensions, then evaluates user-defined calculations on this dataframe. Finally, variables available only in annual frequency are disaggregated to quarterly frequency using the `tempdisagg:td()` function. To ensure the operators and functions work within calculations without additional arguments, the

values of dimensions used in calculations, which are empty in the Eurostat database, will be treated as if they were 0. This allows, for example, to sum multiple national account categories without worrying that the lack of data will result in a result equal to NA. At the same time, to avoid creating 0 values for periods where all data is missing, at the end of the process, all series elements equal to 0 are replaced back with NA,

b. Custom calculations within different Eurostat tables - in this section, calculation formulas are evaluated on a dataframe containing all variables created up to that point. It may happen that dependent equations are found in this set of calculations. To address this, the tool will attempt to evaluate each calculation, and in case of an error (lack of required variables in the dataframe), the next calculation on the list will be evaluated until all calculations are completed.

4. Data saving - dataframes from the script's run are merged and time-limited by the user parameter time_lower_raw. The prepared dataframe data_raw is saved in the /data folder.

1.3 Data Validation

This script implements tools for validating the downloaded data and model equations.

Input Data:

`data_raw` - a dataframe of raw time series data from the Eurostat database containing variables defined in the Variables tab of the dictionary.

Output Data:

This script does not produce any output data. It is used solely for reviewing the results of validation.

Process:

In addition to configuring the environment and loading input data, the script consists of four parts:

1. Review of raw time series - In this part, the user can create plots of the downloaded variables and assess the level of missing data. Initially, completely empty series are identified, and if at least one such series exists, the tool will pause execution with a corresponding message. Then, a list of variables `miss_rates_0.05` is created, indicating variables with missing data at or above 5%. Special attention should be paid to these variables because the imputation implemented in script 3_Data_tranformation will have a significant impact on them.

2. Transaction Matrix - In this part, the transaction matrix `matrix_tr` is calculated. It's important to note the sign rule described above - the sign is added externally to the variable unless it concerns net values, which already include the appropriate sector-specific sign.
3. Balance Sheet Matrix - In this part, the balance sheet matrix `matrix_bs` is calculated.
4. Validation of Model Equations - In this part, the user can check how well the theory expressed in the model equations corresponds to the collected real-world data. Based on the raw data frame, the right-hand side of each equation is calculated and compared to the left-hand side. This allows answering the question: Does the formula for a variable, as written, indeed equal the observations of that variable?

The entire analysis in this section is performed using a custom function called `validate()`. If the function is executed without any arguments, it will automatically calculate each model equation and produce a `diffs` dataframe. This dataframe includes the following columns next to the equation formula:

- `mape` (mean absolute percentage error) calculated between the left and right sides of the equation.
- `miss` - the number of missing periods in theoretical data (right side) compared to actual data (left side).

If the first argument of the function is the name of a variable, a comparison plot will be created to compare theoretical and actual values `validate("Y")`.

It's worth noting that the equation formulas in the `validate()` function are evaluated on columns of real data. This means that true recursion present during the simulation of an SFC model is lost. In the SFC model, the expression denoting the lag [-1] in each period represents a variable calculated by the model in the previous period. Outside the model, this lag is replaced by the `lag()` function, which shifts the entire vector of known real data in time. As a result, in subsequent periods, the calculation "returns" to the observed values of the variable. This regularity can lead to underestimated error estimates in the case of recursive equations, so special attention should be paid to their fit.

1.4 Data Transformation

In the script 23_Data_transformation.R , data transformation from raw data frame to a modeling-ready data frame is implemented.

Input data:

- `data_raw`: Raw time series data frame from the Eurostat database containing variables defined in the Variables tab of the dictionary.

Output data:

- `data`: Data frame for modeling, containing data imputation for missing values, seasonal decomposition of variables (along with seasonal components and original series), and auxiliary variables.
- `seasonality`: Data frame containing information about which variables are seasonal.

Process:

Apart from environment configuration, loading input data, and restricting the time range of analysis, the script consists of four parts:

1. **Data Imputation**: In this part, all missing values in the data frame are filled using the `imputeTS::na_seadec()` function. The imputation method used is StructTS with Kalman smoothing. Additionally, seasonal time series are imputed without the seasonal component.
2. **Seasonal Decomposition of Variables**: In the beginning of this part, the seasonality of each variable is tested. The test used for this purpose is implemented in the `seastest::isSeasonal()` function. This function conducts several classical seasonality tests (QS, Friedman, Kruskall-Wallis, F-test for seasons, Welch) and combines the results. If a variable is found to be seasonal, it is decomposed using the `decompose()` function. For each decomposed variable, three columns are saved in the data frame:
    - De-seasonalized series, labeled with the original variable name.
    - Seasonal component, labeled with a variable name suffixed with `_seas`.
    - Original time series with seasonality, labeled with a variable name suffixed with `_og`.
3. **Creation of Auxiliary Variables**: In this part, the user has the opportunity to define auxiliary variables that will be added to the data frame for modeling. These variables can be dummy variables, for example, indicating the COVID-19 pandemic by taking the value 1 for pandemic periods and 0 for other periods.
4. **Data Saving**: Output data is saved.

1.5 Parameter Estimation

In script 4_Parameter_estimation.R, the process of estimating the structural parameters of the model is implemented.

Input data:

- `data`: Data frame for modeling.
- `dictionary`: The dictionary described in Chapter 1.
- `dict_equations`: The Equations sheet from the dictionary.

- `equations_param`: Parametric equations from the Param_equation column of the Equations sheet in the dictionary.
- `dict_parameters_original`: The Parameters sheet from the dictionary.

Output data:

- `dictionary`: The dictionary with updated Equations and Parameters sheets.

Process:

Apart from environment configuration and loading input data, the script consists of three parts:

1. **Editing Equations**: In this section, the user has the option of semi-automatically editing parametric equations using the interactive custom function `restimate()`. The function operates as follows:

    1. The user selects which equation to analyze.
    2. Parameters of the selected equation are estimated.
    3. A set of information regarding the estimation is displayed: correlation table, results of stationarity tests, standard output from the model with parameter estimates, standard errors, and other metrics, as well as a plot of fitted values.
    4. Based on the displayed information and additional user-conducted tests, the user decides whether to attempt to introduce a different functional form for the equation.
    5. If so, the user inputs the new formula, and a set of information is generated for the new equation.
    6. If not, the equation is accepted and saved.
    7. The process is repeated until all equations are approved.

    The detailed steps after running the `restimate()` function are as follows:

    1. A list of available equations is displayed, and the user selects the equation to be analyzed by entering its position in the list.

    2. The equation is estimated, and information about the estimation is displayed. The user has the following options:

        - 'n' - next, to approve the current equation and move on to the next one (necessary to confirm the selected formula).
        - 'equation' - entering a new formula for the equation (for convenience, the previous formula is copied to the clipboard, so you can use the Ctrl+V shortcut to paste the old formula and edit it).
        - 't' - transform, indicating the need to enter a transformation of the explained variable to be displayed in the correlation table. In the next step, you should specify the transformation suffix according to the description of the 'y_tf' argument in the `estimate()` function.

- 'r' - reset, resetting the current equation to its original formula.
- 'q' - quit, exiting the editor. If changes have been made, they will be displayed, and a question will appear asking whether to save or discard them. After making this choice, the function ends.
- If none of the above options is entered in the command line, R will attempt to evaluate any other expression entered in the function's environment. This allows you to preview objects such as the full correlation table 'corr' or conduct additional tests without leaving the function. If an incorrect expression is entered, the function will report a failed equation estimation and evaluation.

The function saves the vector of new parametric equations 'param_equations' in the global environment, `restimate()`

2. **Parameter Estimation**: If parametric equation formulas have already been established (in the dictionary or as a result of the `restimate()` function), in this part, the parameters contained in them will be estimated using a custom function called `estimate()`. After the complete run, this function saves a data frame named 'parametry' in the global environment, which contains the parameter estimate values.

   The `estimate()` function (which is also used for estimation in the `restimate()` function) uses an estimation engine included in the script '4_manual_Parameter_Estimation', `estimate()`

3. **Saving Data**: If during the script execution, equation formulas and parameter values change, they will be overwritten in the provided dictionary successively in the Equations and Parameters sheets.

1.6 SFC_Model

In this script, scenario simulations of the Stock-Flow Consistent (SFC) model of the Polish economy are implemented using the `godley` package.

Input data:

- `data`: A data frame for modeling.
- `seasonality`: A data frame containing information about which variables are seasonal.
- `equations`: All equations from the 'Equation' column of the Equations sheet in the dictionary.
- `parameters`: Parameter values along with names from the 'Name' and 'Parameter' columns of the Parameters sheet in the dictionary.

Output data:

- `model`: An SFC model object from the 'godley' package saved in Rds format.
- `data_out`: The output data frame of the baseline model scenario simulation.

- `data_out_seas`: The output data frame with added seasonality to variables that were deseasonalized in the '3_Data_Transformation' script.

Process: Apart from configuring the environment and loading input data, the script consists of seven parts:

1. **Model Definition**: In this part, an instance of the SFC model from the `godley` package is created. Variables, parameters (essentially treated as variables), and equations are sequentially added to the model. It's important to note that all variables present in the 'data' data frame are added to the model. However, only the variables present in the equations will be used.
2. **Simulation**: The baseline scenario of the model is simulated for the analyzed period.
3. **Results Overview**: The output data frame 'data_out' of the model is created, and the 'add_seasonality()' custom function is used to create input and output data frames with restored seasonality.

```
data_out_seas <- add_seasonality(data_out, seasonality)
data_seas <- add_seasonality(data, seasonality)
```

The user also has the option to review the model results on a plot using the custom function `see_fit()`. Depending on the input data frame provided to the function, you can compare input and output data, with or without seasonality.

```
see_fit("Y", data, data_out)
see_fit("Y", data_seas, data_out_seas)
```

4. **Scenarios - Shocks**: In this part, sample alternative scenarios with shocks are presented, programmed according to the standards of the 'godley' package (see Appendix). The first example shock involves a five-fold increase in interest rates from the first quarter of 2015 until the end of the simulation period. The chart shows the impact of the shock on investments.

```
shock_1 <- create_shock() %>%
  add_shock(
    c("chi", "r_A_H", "r_N", "r_L_H", "r_G"),
    rate = 5, start = "2015-01-01"
  )
model <- model %>%
  add_scenario("rates", shock = shock_1) %>%
  simulate_scenario()

plot_simulation(model, c("baseline", "rates"), expressions = "I")
```

5. **Sensitivity Analysis**: In this part, a sample sensitivity analysis of a parameter implemented in the `godley` package is presented. The parameter expressing household taxation is varied from 0.02 to 0.08. The chart shows the impact of changing the parameter on consumption.

```
model_sen <- model %>%
  create_sensitivity(
    variable = "par_T_H_1", lower = 0.02, upper = 0.08, step = 0.01
  ) %>%
  simulate_scenario(start_date = data$time[1], periods = nrow(data))
plot_simulation(
  model_sen,
  scenario = "sensitivity", take_all = TRUE, expressions = "C"
)
```

6. **Saving the Model and Data**: Output data is saved.

2 Procedures

2.1 Adding Equations and Variables

The procedure for adding new equations and variables to the model of the Polish economy is described below:

1. Add and complete the rows of the "Equations" worksheet in the dictionary.xlsx file.
2. If the equation includes new variables, add and complete the rows of the "Variables" worksheet in the dictionary.
3. Save the new dictionary file in /data/dictionary.xlsx.
4. If an automatic run of the tool is desired, execute the main script (see Chapter 4.1) and skip the remaining steps. Simulation results can be viewed using the custom function `see_fit()`. Otherwise, proceed to the next step.
5. If new variables have been added, run the script 1_Data_download.
6. Run the script 2_Data_Validation. Validate new equations using the custom function `validate()`.
7. Run the script 3_Data_Transformation.
8. If new parametric equations have been added, estimate the parameters using the scripts 4_Parameter_Estimation.
9. Run the script 5_Model_SFC (see Chapter 4.9) and display the results of the baseline scenario simulation using the custom function `see_fit()`.

2.2 Adding New Dimensions to Eurostat Tables

When adding variables from new online_data_code tables of the Eurostat database to the model, it may happen that the new table contains dimensions not present in columns G to AB. In this case, add a new column with the name of the new dimension to the dictionary among the current dimension columns. The convention for arranging columns was their order in the database, but in practice, this does not matter as long as the order of members in the names of variables of the first variant of calculations matches this order.

2.3 Model Calibration

Fitting large macroeconomic models to data is a complex problem, especially when it is not possible to fit structural parameters in a multiequational model but is done on real data and individual equations outside the model, as is the case in the described model of the Polish economy. The following steps have been recorded to facilitate the "bottom-up" model fitting method, i.e., the assessment of the smallest elements of the model. "Top-down" fitting is also possible and involves evaluating from the results of the model, but it is much more difficult and requires a very good understanding of the model.

Steps for fitting the model "bottom-up":

1. Review and assess equations and variables in the "Equations" and "Variables" tabs of the dictionary. Answer questions such as: Is each equation consistent with the theory to be implemented? Is each variable documented according to its interpretation and role in the equation? For the initial assessment of variables, you can use the Eurostat data browser (the "Link_to_source" column in the "Variables" worksheet).
2. Review and evaluate equations and variables in the script 2_Data_Validation. Answer questions such as: Are the values of the retrieved variables consistent with expectations? Are real-world data consistent with the theory implemented in the equations?
3. Estimation of structural parameters in scripts 4_Parameter_Estimation. Answer the question: Do the equation formulas, variable transformations, and parameter estimates reflect real relationships between economic variables?
4. Run and evaluate the SFC model in the script 5_Model_SFC only for non-parametric equations. This can be achieved by replacing the value of the equation variable in this script with the value of the variable "equation_sfc" from the script 2_Data_Validation. This way, you can assess the model before behavioral equations are calculated. Any errors existing at this stage are independent of structural parameters, so you can address them more transparently.
5. Run and evaluate the full SFC model in the script 5_Model_SFC_Poland.

2.4 Endogenous Variable Shocks

Within the godley package, it is possible to program shocks to exogenous variables. Shocks to endogenous variables will not have an effect because, regardless of the initial values assigned to them in the starting frame, they will be overwritten during the simulation (except for the first few periods, depending on the number of lags).

Shocks to endogenous variables can be introduced by applying shocks to exogenous variables present in the equations explaining these endogenous variables. For example, in behavioral equations estimated with an intercept term, a shock to the intercept term can be interpreted as an external effect on the endogenous variable explained by the equation. If the equation lacks independent exogenous variables that can be modified without losing interpretation, you can add a dummy variable to the equation and apply the shock to it. The process of adding auxiliary variables can be found in the script 3_Data_Transformation.

As an example of a shock to an endogenous variable in Section 4.4 of the script 5_Model_SFC_Poland, an increase in wages was implemented by raising the value of the intercept term.

```
shock_4 <- create_shock() %>%
 add_shock("par_W_1", rate = 0.1, start = "2015-01-01")
model <- model %>%
 add_scenario("wages", shock = shock_4) %>%
 simulate_scenario()
```

2.5 Forecasting

The tool and the godley package described in this document do not inherently possess forecasting functions. All values shorter than the interval they are meant to fill are extended with the last element. Therefore, the tool, without additional functionalities, is only capable of performing naive forecasting. The time series that are extended with the last value are all the series of exogenous variables in the input frame of the model. If, before providing the data frame to the model, values of exogenous variables are independently filled with forecasts for future periods, then the SFC model's outcome can be considered a forecast because, in future periods, it will rely on the values of exogenous variables from the forecasted values.

Alternatively, you can obtain an extrapolative forecast relatively easily using the tool. This requires modifying the script 3_Data_Transformation. It's worth noting that if empty future periods (i.e., with NA values) are added to the data frame 'data' after its creation, they will be filled during the data imputation step. This is not a recommended forecasting method, as it shares the drawbacks of extrapolation and related methods, but it may be better than a naive forecast.

3 Development Directions

Below are suggested development directions for the tool, which can be pursued as the first steps in its expansion:

1. Implementation of true recursiveness in the equation validation stage (problem described in Chapter 1.3).
2. Addition of auxiliary variable selection criteria other than correlation coefficients, e.g., Granger causality test. The estimation process can also include the automatic creation of parametric equation formulas based on variable selection analysis results.
3. Currently, the only manual step when changing the formula of a parametric equation is to rewrite it in the dictionary to conform to the `godley` package standard. An improvement could be the automation of this process.
4. Extending the forecasting capabilities of the tool beyond naive forecasting and extrapolation.

Appendix

A1 The `godley` Package

Below, we present a minimal use case of the package, demonstrating a simulation of the baseline scenario in the classical SIM model (Godley & Lavoie, Monetary Economics, 2007).

To begin, you should create an empty model using the `create_model()` function. This model is an S3 object (a list) with the SFC class.

```
model_sim <- create_model(name = "SFC SIM")
```

Now you can add variables to the model using the `add_variable()` function. This will add a tibble called `$variables` to the model. Please note that the following characters are not allowed in variable names: "§£@#${};:'~'? !%^&*()-+=[]|<,>/" It is considered good practice to use letters, numbers, and underscores "_" when creating variable names.

For each variable, in addition to its name and description, you can specify an initial value using the `init` argument. This argument can be a single number or a vector. In either case, the specified values will appear in the initial matrix `initial_matrix` of the model, which will be used for the simulation.

It is worth noting that providing vectors for endogenous variables is allowed, but apart from the first few values (depending on the chosen number of lags in the equation), the remaining elements of the vector will not be used (they will be calculated using the equation and overwritten in subsequent periods).

If no initial value is provided for an endogenous variable, it will default to 1e-05. For exogenous variables, the absence of at least one initial value is not permissible and will be signaled with an appropriate message.

Throughout the entire package, any values shorter than the required range will be extended with the last value. This applies to initial values of variables as well.

```
model_sim <- model_sim |>
add_variable(
"C_d", desc = "Consumption demand by households",
"C_s", desc = "Consumption supply",
"G_s", desc = "Government supply",
"H_h", desc = "Cash money held by households",
"H_s", desc = "Cash money supplied by the government",
"N_d", desc = "Demand for labor",
"N_s", desc = "Supply of labor",
"T_d", desc = "Taxes, demand",
"T_s", desc = "Taxes, supply",
"Y", desc = "Income = GDP",
"Yd", desc = "Disposable income of households",
"alpha1", init = 0.6, desc = "Propensity to consume out of income",
"alpha2", init = 0.4, desc = "Propensity to consume out of wealth",
"theta", init = 0.2, desc = "Tax rate",
"G_d", init = 20, desc = "Government demand",
"W", init = 1, desc = "Wage rate"
)
model_sim$variables
```

Next, you can add the appropriate equations to the model. You can do this using the `add_equation()` function by providing equations in text form. This will add a tibble called `$equations` to the model.

Equations must meet the following requirements:

1. The following characters are not allowed: “§£@#${};:‘~’? The following characters are allowed but will be treated as operators (ignored when reading variable names): !%^&*()-+=[]|<,>/
2. On the left-hand side, there should be a variable explained by the equation in its untransformed form. For example, C_s represents consumption. Not allowed are expressions like C_s*2 or log(C_s). If an operator or function appears on the left-hand side, the entire thing will be treated as a variable name.
3. There should be an equal sign”=” between the left and right sides.

4. On the right side, you should have the rest of the equation formula written using variable names and standard operators and functions in the R language. All specified operations will be performed on successive numeric rows of the matrix. Therefore, any operations that meet these requirements will also be correct in the equations of the godley package.

   For user convenience, frequently used operations on matrix columns have also been introduced - delays and differences:

   - **Delays** of variables should be written by adding a negative delay degree in square brackets on the right side of the variable. For example, the first lag of consumption: C_s[-1], the third lag of consumption: C_s[-3]. Currently, the package supports delays of up to 4 degrees (useful for quarterly data). This notation corresponds to the `lag()` function for vectors in the R language.
   - **First difference**, i.e., C_s - C_s[-1], can be written as d(C_s). This is equivalent to the `diff()` function for vectors in the R language. Please note that this operation is defined only for the first difference - for, say, the third difference, you need to use the expression C_s - C_s[-3].
   - **Delayed difference** can be obtained by combining the above, i.e., d(C_s[-1]).

5. One explained variable has only one equation in which it appears on the left-hand side. Two different equations describing the same variable are not allowed (exact duplicates are, however, signaled with an appropriate message).

```
model_sim <- model_sim |>
add_equation(
"C_s = C_d", desc = "Consumption",
"G_s = G_d",
"T_s = T_d",
"N_s = N_d",
"Yd = W * N_s - T_s",
"T_d = theta * W * N_s",
"C_d = alpha1 * Yd + alpha2 * H_h[-1]",
"H_s = G_d - T_d + H_s[-1]",
"H_h = Yd - C_d + H_h[-1]",
"Y = C_s + G_s",
"N_d = Y/W",
"H_s = H_h", desc = "Money equilibrium", hidden = TRUE
)
model_sim$equations
```

After defining all the variables and equations, you can proceed to run the simulation using the `simulate_scenario()` function. As part of its operation, it will perform a validation

check of the user-input model using the `prepare()` function (also available in the exported package environment), followed by the simulation. The result of the simulation will be a tibble called `$result` added to the model under the simulated scenario (for the base scenario, it is `$baseline`).

As part of the simulation arguments, you can choose the number of periods (`periods`) for which you want to simulate and the start date (`start_date`). The date is optional; if not provided, the periods will be numbered with consecutive natural numbers. However, if a date is provided, currently supported frequencies are quarters, e.g., the second quarter of 2022 would be "2022-04-01".

You can also choose from two algorithms (`method`) for solving systems of simultaneous equations: Gauss-Seidel and Newton-Raphson (currently only available for systems with a lag degree no greater than 1). By default, equations will be solved using the Gauss method, which involves iterative computation of equations. Independent equations for a given period will be calculated in a single iteration. Interdependent equations will be combined in a loop and solved iteratively. In the first iteration for a given loop, the initial values (`init`) will be used for calculations, and in subsequent iterations, the results from the previous iteration will be used for all variables until convergence is reached with the specified tolerance (`tol`) or the maximum number of iterations (`max_iter`) is reached. Iterations continue until convergence is achieved for each period. If a value of Inf, NaN, or NA is reached in any period, the computation will be suspended along with an appropriate message.

Before calculating independent and interdependent equations, an order is established for all equations that (1) allows the calculation of an endogenous variable first, which will later be used as an exogenous variable, and (2) combines interdependent equations in a loop to ensure the substitutability of exogenous variables.

The Newton method follows a similar process, but for solving interdependent equations, it uses the Newton-Raphson algorithm for finding zero points implemented in the `rootSolve::multiroot()` function.

During the model construction stage, it may be helpful for the user to set the `info = TRUE` argument. This will return the result of the `prepare()` function, which includes information about the types of all variables (endogenous/exogenous) and other summary information about the input data.

```
model_sim <- model_sim |>
simulate_scenario(
periods = 100, start_date = "2015-01-01",
method = "Gauss", max_iter = 350, tol = 1e-05
)
model_sim$baseline$result
```

You can visualize the simulation results on a plot using the `plot_simulation()` function. You can select the scenarios (`scenario`), the time range of the plot (`from` and `to`), and a list of expressions (`expressions`) which can be variable names or expressions created from them (e.g., G_s/Y).

```
plot_simulation(
model_sim, scenario = "baseline",
from = "2015-01-01", to = "2023-01-01",
expressions = c("Y", "G_s", "T_s")
)
```

A2 Shocks

The godley package also allows for the creation and simulation of shocks. Below, we'll demonstrate how to introduce an increase in government spending in a sample model.

An alternative scenario with a shock in the godley package consists of three elements:

1. Shock Parameters (which variable is affected? when will it occur? what value will it take?)
2. Transformation of the initial matrix (which part of the matrix from the original scenario will the new simulation be based on?)
3. Simulation (how many periods ahead should the new scenario be calculated?)

Designing a shock should begin with creating an empty S3 object (list) of the SFC_shock class using the `create_shock()` function. Then, you can add shock parameters for selected variables (`variable`) to it using the `add_shock()` function. You can specify the value that the variable should take during the shock period in three ways: by providing an exact vector of values (`value`), by modifying the variable by a relative part (`rate`), or by modifying the variable by an absolute value (`absolute`).

Let's introduce a 20% increase in government spending from 2017 to 2020.

```
sim_shock <- create_shock() |>
add_shock(
variable = "G_d", rate = 0.2,
start = "2017-01-01", end = "2020-10-01",
desc = "permanent increase in government expenditures"
)
```

Let's now add a new scenario with this shock to the model using the `add_scenario()` function and simulate it. You need to specify which scenario (`origin`) you want to base it on and which

portion of it (`origin_start`, `origin_end`) will serve as the initial matrix for the new scenario. During the simulation of the scenario, you can specify the number of periods (`period`) for the new scenario - by default, it corresponds to the original scenario.

```
model_sim <- model_sim |>
add_scenario(
name = "expansion", origin = "baseline",
origin_start = "2015-01-01", origin_end = "2023-10-01",
shock = sim_shock
) |>
simulate_scenario(periods = 100)
model_sim$expansion$result
```

On the chart, you can see the result of simulating the new scenario with the shock. The increase in government spending had a positive impact on income and, in the short term, reduced the government balance.

```
plot_simulation(
model_sim, scenario = "expansion",
to = "2025-01-01", expressions = c("Y", "G_s", "T_s")
)
```

A3 Sensitivity

The godley package allows users to examine how changes in input parameters affect simulation results. Let's explore how small changes in the parameter `alpha1` affect the short-term dynamics of the model.

The `create_sensitivity()` function creates a new object of the SFC class based on an existing model and automatically adds multiple scenarios that differ only in the value of the selected parameter `variable` within the specified range (`lower`, `upper`, `step`).

```
model_sens <- model_sim |>
create_sensitivity(
variable = "alpha1", lower = 0.1, upper = 0.8, step = 0.1
) |>
simulate_scenario(periods = 100, start_date = "2015-01-01")
```

You can view the results on a chart. To simultaneously plot multiple scenarios that have the same `scenario` component in their name, you should use the `take_all = TRUE` argument.

```
plot_simulation(
model_sens, scenario = "sensitivity", take_all = TRUE,
to = "2028-01-01", expressions = c("Y")
)
```

A4 Functions

Below is a list of the most important functions of the package:

- `create_model()` - creates an SFC model,
- `add_variable()` - adds variables,
- `add_equation()` - adds equations,
- `simulate_scenario()` - simulates scenarios,
- `plot_simulation()` - creates a chart of simulation results,
- `create_shock()` - creates a shock object, - `add_shock()` - adds shock specifications,
- `add_scenario()` - adds a scenario to an existing model,
- `create_sensitivity()` - creates a new SFC model with sensitivity scenarios.