Gekko 3.1.x help system

Gekko Timeseries & Modeling

T-T Analyse

Table of Contents

Part I

1 Introduction

Last modified: 1/6 2025

Gekko Timeseries and Modeling Software is a free and open-source software system for managing and analyzing timeseries data, and for solving and analyzing large-scale economic models. See the Gekko homepage: www.t-t.dk/gekko. Read more about the status of different Gekko versions on the Gekko versions overview page.

See the search page if you need to search the Gekko help system.

Short-cuts to oft-used sections:

- Gekko statements
- Gekko functions
- Gekko options

- Data Management User Guide
- Modeling User Guide

The Gekko help system contains the following main sections:

- **Introduction** (this page). General info on new features, installation, user interface, versions, etc.
- Search. Search the Gekko help system.
- Gekko User Guide. New users are guided through the software with descriptions, practical examples, etc.
- Gekko User Manual. Comprehensive and detailed description of statements, functions, etc.
- Excel add-in (Gekcel). A component that makes it possible to issue Gekko statements from within Excel.
- Appendix

The help system occasionally uses toggles: embedded text that can be opened and closed quickly like the following: +toggle (try to click it).

Some explanations and examples are not relevant for all readers, or different readers may prefer different levels of details. Toggles are light-weight and quick to open and close and therefore practical for showing bits of extra information of the nice-to-know rather than need-to-know kind.

In the pdf version of the help system, all toggles are shown per default.

1.1 New features

This version of Gekko is part of the 3.1.x development series, extending the official Gekko 3.0. The 3.1.x development versions are intended to be non-breaking and just as stable as Gekko 3.0. The development versions are published on the Gekko website here (see also the changelog for more details, or Github for even more details).

Gekko 3.1.x

Care is taken to ensure that the 3.1.x versions are stable and backwards compatible with the official Gekko 3.0 (from April 2019). Compared to Gekko 3.0, the latest 3.1.x version implements the following:

- New DECOMP with many new features, among other things recursive decomposition and dimensionality. Also supports GAMS models.
- Data tracing for timeseries, storing info on how the timeseries data have been constructed (cf. TRACE2). Also program dependency tracking: which (data) files are read/written?
- GAMS equations/models are now fully supported (via so-called 'unrolled' equations) and can be used with the new DECOMP (cf. MODEL). For GAMS models, SIM<res> works, too.
- Autocomplete on series names, including array-series. Is triggered with `[Tab]` or `[Ctrl+Space]`.
- Weekly and daily frequencies (`w` and `d`) are introduced, with dates like `2019w50` or `2019m12d24`. There are also new functions to bridge between Gekko daily dates and Excel dates.
- Gekcel: an Excel add-in (xll file) that makes it possible to (among other things) read/write to Gekko databanks via Excel functions from within Excel, without starting up Gekko. This component is downloaded separately, at the same download pages as the normal Gekko system.
- Improved and less voluminous error messages.
- Local options: normal options that only apply to the current Gekko statement.
- User-defined functions and procedures allow prompting and default values for arguments. For functions and procedures, you may omit default arguments, or you may use `?` to make Gekko prompt (input window) regarding such default arguments, for instance `p?;` to call a procedure `p` with prompting, or `f?();` to call a function `f` with prompting. See more under FUNCTION and PROCEDURE.
- User-defined functions and procedures can be stored in library zip files (cf. the LIBRARY statement). Gekko libraries use lazy loading so that you may load a large library quickly without slowing down Gekko. Normal files can also be accessed from the `\data` subfolder of a library. Libraries can be loaded in a hierarchical fashion, and similarly to databanks, you may use colon `:` to refer to a function/procedure/file from a particular library (like `lib1:f()` or `lib1:data.csv`). There are capabilities regarding encapsulation (`this:` designation and private `__` prefixes).

- OLS improvements, showing parameter stability, fit and decomposed values. Trend polynomials can be added automatically, and endpoint restrictions can be stated easily (so-called 'Finnish' trends). OLS<dump> will transform the estimation into an equation suitable for Gekko models (FRML).
- New equation types for models. In addition to the usual Y-type equations (typically containing reverted J-factors and similar technical stuff), you may use T-type equations, too. The difference is purely semantic, but the idea is that T-type may denote that the equations are 'interesting' (for instance used for tabelling), rather then purely technical. Equations may also be P-type: these are not not be part of the model at all, but can be called with the new PREDICT statement. In addition, via RUNBEFORE$ and RUNAFTER$ tags, you may put normal Gekko program statements inside the model file, to be run before or after the simulation proper. These features are experimental at the moment.
- PREDICT statement to call individual model equations.
- IMPORT<aremos> can be used to transfer timeseries from AREMOS with higher precision (significant digits) than the tsd format.
- Python interface implemented, see RUN_PYTHON and EXPORT<python>. This is similar to the R interface, which has been simplified.
- EXPORT<array> implemented, using the Apache Arrow data format for easy use in for instance R or Python (and others). See more under R_RUN and PYTHON_RUN.
- Better selection and indexing possibilities regarding nested lists. This makes it possible to import an Excel sheet as a nested list, and handle the cells in a matrix-like fashion (similar to how NumPy deals with n-dimensional arrays).
- SHEET<import list> and SHEET<import map> to transfer an Excel sheet into a nested list or map for further processing.
- Dynamics checking from version 3.1.7 an onwards. A statement like `x = x[-1] + 1` with the lagged dependent/endogenous `x[-1]` on the right-hand side will fail with a 'dynamics error' if the statement is not decorated with `<dyn>`, `<dyn = no>`, or is put inside a `block series dyn = yes|no`. See more here.
- Possibility to restore statements from a previous interactive Gekko session easily (by clicking a link).
- Improvements in the GAMS to Gekko equation translator, can also be used with mark and right-clicking text.
- In the Gekko main window, you may choose how `[Enter]` and `[Ctrl+Enter]` works in the editor (lower part of the main window). For instance, RStudio style is possible, where `[Enter]` issues a newline, and `[Ctrl+Enter]` executes the line. See OPTION interface edit style = ...
- Use the root() function to avoid hard-coded paths in your system of Gekko programs, databanks, etc. Using root() simplifies portability: the possibility of painless copying/moving a folder Gekko files to another location.
- File paths allow so-called "zip paths", for instance `g:\projects\storage.zip\data.csv` to refer to the file `data.csv` inside `storage.zip`. (the .csv file is automatically extracted). This only works for file reading, not writing.
- SPLICE supports multiple series, date intervals, and different methods. Function splice() is also implemented.
- INTERPOLATE now has the Denton, Cholette and 'Olsette' methods, too.
- INDEX supports array-series.
- PRT<view> to show nested lists in a list window.
- IMPORT<px> supports new short variable codes.

- CLOSEALL statement (= `CLOSE *; CLEAR;`).
- ELSEIF statement.
- Function laspchain() with `annualoverlap` argument, for non-annual chain indexes.
- Function comparefolders() to compare the contents of to file folders.
- Drag & drop of files into the main Gekko window.
- Loading of models and databanks uses multi-core cache files. For large gdx files speedup can be a factor 3-5x.
- Fencing options, restricting read/write of files to certain folders. See options `option global fence ...` .
- Better way to show warning, so they do not flood the main Gekko window. Also, warnings can be inspected at the end of a Gekko run.
- New `gekko version` statement to check Gekko version validity in .gcm files.
- Better way of handling warning messages in a unified way.
- 64-bit version available (both Gekko and Gekcel).
- Quite a lot of smaller fixes and improvements regarding statements and inbuilt functions, and the consistency of these.
- New functions: collapse(), fromBank(), getSpecialDay(), getWeek(), getWeekday(), interpolate(), isLibraryLoaded(), If(), mod(), observations(), rebase(), root(), smooth(), tic(), toc(), upperFirst().
- New options: option collapse method, option collapse missing d, option interpolate method, option smooth method, option copy respect.

Gekko 3.0

Syntax-wise the syntax changes from 2.0/2.2/2.4 to 3.0 are not quite as dramatic as the changes from 1.8 to 2.0. Version 3.0 is more a question of new capabilities, improving upon existing capabilities, cleaning up the syntax, and providing general consistency. Regarding model solving and the way databanks are opened and closed, nothing has been changed from 2.4 to 3.0, in order to keep these parts of Gekko stable. There is an automatic translator from 2.0/2.2/2.4 available, cf. TRANSLATE. The most significant changes from 2.4 to 3.0 are the following:

- All variables types, series, value, string, date, list, map, and matrix, now reside in databanks, and all variable types can be stored in .gbk databank files.
- Assignment of variables no longer needs to include type. So `series x = 5; val %v = 100; matrix #m = [1, 2];` can now be written more compactly `x = 5; %v = 100; #m = [1, 2];`. To be completely sure of the type of for instance `%v`, you can still use for instance `val %v = 100;`. Note also that in Gekko 3.0, you must use `%` or `#` type symbols on the left-hand side, so for instance `val v = 100;` must be `val %v = 100;` in Gekko 3.0.
- Series variables all use frequency indicator `!a` (annual), `!q` (quarterly), `!m` (monthly), `!d` (daily) or `!u` (undated). These indicators can often be omitted, for instance `prt x;` to print out `x` of the current frequency. Use the indicators to access a series of another frequency than the current, or for mixed frequency use.
- Map is a new variable type that stores variables by name. Maps are like mini-databanks and are among other things handy for bundling variables together, for instance when getting variables in and out of user-defined functions.

- Lists may now store any type of variables, not just strings. The list functionalities have been augmented, including list functions. Two-dimensional listfiles (lists of lists) are supported, using a .csv-like format. Note that list definitions in Gekko 3.0 generally include parentheses, for instance `('a', 'b')` for a list of two strings. However, in the LIST and FOR statements, you may use a 'naked' list definition, for instance `#m = a, b;` being equivalent to `#m = ('a', 'b')`, or `#m = 1, 2;` being equivalent to `#m = (1, 2);`. Instead of LIST<direct> from Gekko 2.4, the user can just use a naked list in Gekko 3.0. See more on naked lists.
- Introduction of local and global databanks. The local databank is used for temporary/discardable variables (for instance inside functions/procedures), and the global databank can be used for permanent storage of settings etc. that are intended to survive for instance READ and CLEAR statements. In combination with these banks, there are the new statements LOCAL and GLOBAL to denote such variables. Apart from that, the databank logic is exactly the same as in Gekko 2.4. The local databank is searched first, and the global databank last (also in sim-mode).
- BLOCK ... END is a new structure to set time period and/or other options temporarily, setting them back after the block is finished.
- Since series statements are calculated in a vector-like fashion in Gekko 3.0/3.1.x (in contrast to the Gekko 2.x versions), lags no longer automatically accumulate period-for-period, if the left-hand side variable is present with lags on the right-hand side (lagged dependent/endogenous variable). See more on the help page on dynamic statements. In Gekko versions >= 3.1.7, a statement like for instance `x = x[-1] + 1` will need to be decorated with an indication regarding whether it should accumulate (`<dyn>`) or not (`<dyn = no>`), cf. examples on the linked help page. Note that for absolute accumulations like `x <dyn> = x[-1] + 1`, you may use `x <d>= 1`, `x ^= 1` or `dif(x) = 1` instead (and there are also variants regarding relative accumulations, see below). Note that setting `<dyn>` (or `block series dyn = yes`) unnecessarily entails a speed penalty and should therefore not be used unless needed.
- Wildcard lists are syntactically changed from for instance `[a*x]` to `['a*x']` to obtain a list of strings matching the wildcard, or `{'a*x'}` if the matched strings are going to be used as variable names (for instance in PRT, etc.). In statements that do not accept expressions, for instance COPY, INDEX, DISP, etc., the shorter 'naked' wildcard `a*x` is legal too.
- PRT and PLOT now handle mixed frequencies in the same plot/print.
- PRT/PLOT works on non-indexed array-series, for instance `prt x;` instead of `prt x[#i, #j];` (where x is an array-series).
- User-defined functions and procedures have been reworked for Gekko 3.0: they can be accessed from anywhere, as long as they have been defined chronologically before the call. The old `option library file` is now obsolete. A Gekko program file (.gcm) with user functions/procedures can just be loaded with RUN in gekko.ini, and subsequently the functions/procedures will be available until the next RESET/RESTART (functions/procedures are not stored in databanks). Functions/procedures support default values and prompting.
- User-defined functions and procedures may use a <>-field to indicate time parameters. Hence, a user-defined function `scale()` may be called like `scale(<2010 2020>, x)`, and a procedure `scale` may be called like `scale <2010 2020> x;`. Inside the functions/procedures, these time parameters can be accessed as dates, and if they are omitted in the call, the time parameters are set to

correspond to the the local/global Gekko time period. The new <>-field is also used in some of the inbuilt functions: avgt(), sumt(), hpfilter(), laspchain(), laspfixed(), pack(), unpack(), time().

- All functions (including user-defined functions) can be called as object functions on the first argument (not counting time parameters): this is called Uniform Function Call Syntax (UFCS). So a function like for instance `f(x, y)` can generally be written as `x.f(y)`. Therefore, instead of for instance `f(#m, %s)`, you may use `#m.f(%s)`. Such functions can be chained, for instance `#m.extend(#m1).remove('a').sort()`, providing a more fluent syntax than the equivalent and 'backwards' `sort(remove(extend(#m, #m1), 'a'))`.
- A new name type is introduced for function and procedure arguments, in order to avoid unnecessary single quotes. You may define for instance `procedure f name %x; prt ref:{%x}; end; f a1;`. After this, `f a1;` will print out `a1` from the Ref databank, which is more convenient than having to type `f 'a1';`. Inside the function/procedure, the name `%x` works just like a string `%x`.
- The NAME statement is obsolete, so for instance `NAME %s = 'a';` is not legal. Instead, use STRING and refer to the string with {}-curlies. For instance `%s = 'bvat'; prt {%s};`. Note that there is a name-type for functions/procedures in order to avoid quoted argument strings.
- A lot of new in-built functions are added to deal with variable names represented as strings, for instance the string `'b2:x[a, y]'`. There are functions to add/set/get/remove the databank, frequency, index, etc. part of such a string.
- Faster gdx read/write. Optional equation browser for GAMS equations. In `option model type = gams` mode, ENDO/EXO has been reworked to interface with GAMS.
- Alias names may be used, for instance providing a mapping from old to new variable names. See `option interface alias`.
- Better abort red button that should work in all cases where Gekko needs to be stopped.
- PLOT can export to pdf.
- m() can be used instead of miss() to indicate missing value.
- OLS<dump> can dump results as FRML equation for use in models.
- Enhanced format() function that can control width and alignment, and {}-curlies inside strings can be formatted.
- New statements: BLOCK, CUT, LOCAL, GLOBAL, MAP, VAR.
- Removed statements: NAME (use strings and {}-curlies), SHOW (use PRT), UNSWAP.
- New functions: addbank(), addfreq(), append(), contains(), count(), data(), dates(), except(), extend(), flatten(), getbank(), getdomains(), getendoexo(), getfreq(), getfullname(), getindex(), getmonth(), getname(), getnameandfreq(), getquarter(), getsubper(), getyear(), index(), isopen(), map(), pop(), preextend(), prefix(), prepend(), readfile(), remove(), removebank(), removefreq(), removeindex(), replacebank(), replacefreq(), replaceinside(), rotate(), seq(), series(), setbank(), setdomains(), setfreq(), setname(), setnameprefix(), setnamesuffix(), sort(), strings(), stripend(), stripstart(), substring(), suffix(), timeless(), unique(), vals(), writefile().
- Removed (renamed) functions: difference() is now except(), piece() is now substring(), search() is now index(), strip() is now replace(), trim() is now strip(). The following functions have reordered parameters regarding dates: avgt(), sumt(), hpfilter(), fromseries(), unpack().

- New options: option decomp maxlag, option decomp maxlead, option gams time freq, option interface alias, option interface remote file, option interface table operators (renamed from 'printcodes'), option model type, option plot elements max, option plot using, option print elements max, option print split, option series array calc missing, option series array print missing, option series dyn, option series normal calc missing, option series normal print missing, option series normal table missing.
- Removed options: option databank logic, option interface table printcodes (renamed to 'operators'), option library file, option series array ignoremissing (renamed).

Gekko 2.4

Gekko 2.4 is a relatively small update on the top of Gekko 2.2, and 2.4 should be just as stable as 2.2. The main focus of development is the upcoming Gekko 3.0, but still 2.4 contains the following augmentations:

- PROCEDURE implemented. User-defined functions and procedures can be put in a general library file (lib.gcm), so they can be stored in a central place.
- IMPORT<collapse> for collapsing high-frequency data (for instance daily observations) into monthly, quarterly or annual Gekko-timeseries. The data must reside in an Excel spreadsheet, more formats will be supported later on.
- Array-series, with $-conditionals, summing etc, cf. under the SERIES statement. Array-series are further developed in the upcoming Gekko 3.0.
- Robust Newton (better handling of illegal starting values). This is managed by means of OPTION solve newton robust = yes|no, and with robust = yes (default), Gekko will handle illegal stating values (like the logarithm of a negative number) much better.
- Read/write of GAMS datafiles (gdx-files): IMPORT<gdx> and EXPORT<gdx>. See also the OPTIONs under OPTION gams ... regarding how gams files are handled.
- Reading of PC-Axis files: IMPORT<px>.
- Export of R-datasets: EXPORT<r>.
- New 'ser' (series) files format: IMPORT<ser>. This entails fast reading of flat SERIES-like lines like "x 2020 2023 100.0 210.0 150.5 200.7".
- Better engine regarding IMPORT and EXPORT of xlsx-files. The new system (default) does not depend upon Excel being installed on the pc, and should be more stable and leak less memory.
- MATRIX definition with row/colnames.
- Remote control of Gekko is made possible via using a remote.gcm Gekko program file, cf. OPTION interface remote = yes|no.
- EXPORT<cols> implemented for .csv and .prn files.
- Some new functions: avgt(), sumt(), time(). The two first handle sums and averages over time (for timeseries).
- Stand-alone html equation browser generator (DOC<browser>).
- OLS<dump> can dump results as FRML equation for use in models.

Gekko 2.2

Gekko 2.2 most notably adds a lot of new graphing capabilities (PLOT) to Gekko.

- PLOT statement completely overhauled, see demo plots here. Graphs can be controlled in a lot of new ways, either as options in the PLOT statement, or in a template file (.gpt), or both. Histograms/bars/boxes are supported, too. There is a special handy option to separate boxes and lines vertically: PLOT<separate>, and many other possibilities.
- OPEN<edit> should be used instead of OPEN<prim>, and LOCK/UNLOCK statements can lock/unlock already opened databanks. Opened databanks (OPEN without options) are now opened *last* in the list of databanks. Opened databanks are now protected (non-editable) per default.
- OLS statement improved, including linear restrictions on parameters.
- INTERPOLATE and REBASE statements implemented.
- XEDIT statement to open up a dedicated and in-built xml editor (for graph and table templates).
- MATRIX statement allows all kinds of indexers on left-hand side.
- SHEET<import matrix> imports a matrix from an Excel sheet.
- IMPORT accepts dates.
- PIPE improvements.
- Functions: random number functions, see rseed(), runif() and rnorm(). Functions pchy(), dify() and dlogy() to handle yearly differences. Functions movavg(), movsum() for moving averages/sums, and lag() for lags. Function chol() for Cholesky decomposition of matrices. See here.
- Some new table options: 'mdateformat', 'decimalseparator', 'thousandsseparator' and 'stamp'. See under OPTION, in the OPTION table ... section. With these options, a number like 12345.67 can be printed as 12,345.67 or 12.345,67, and this may be combined with negative decimal places (for instance "f9.-2", to produce 12,300 or 12.300). Monthly dates can be formatted as for instance 'Jan. 2020' instead of '2020m1'. Menu files accept links to .gcm files.
- The .gbk databank file format now uses a datafile called 'databank.data' internally (instead of 'databank.bin'). The old name caused problems when sending databank files over email. To produce a databank file suitable for Gekko 2.0 or 1.8, see the note in the WRITE help file.
- Options: see the end of the OPTION statement regarding new options in Gekko 2.2.

Regarding Gekko 2.0 and earlier, see this page.

1.2 Installation

Installation files

In order to install Gekko as a standalone package for economic analysis, go to www.t-t.dk/gekko, and choose 'Download' to choose between a stable version or a development version (direct link: stable versions or development versions). If you have problems installing, please consult the trouble shooting guide. After installation and starting up Gekko, it might be convenient to create a .bat file to start up the program in the future (see more in the 'Setup and environment' section below).

For ADAM users, please use the setup facilities supplied by Economic Modelling, Statistics Denmark, in order to install ADAM+Gekko (in order to uninstall ADAM+Gekko, use the uninstall facilities supplied by Economic Modelling, Statistics Denmark). Please note that Gekko is not tied to the ADAM model in any way, and is being used by many other users.

Uninstallation

Uninstalling Gekko as standalone package can be done from the Windows Control Panel. Close Gekko, start the Control Panel and choose Add/remove programs, then select Gekko and uninstall it.

Setup and environment

Gekko uses the concept of a working folder from which files are read and written. This may be chosen in two ways:

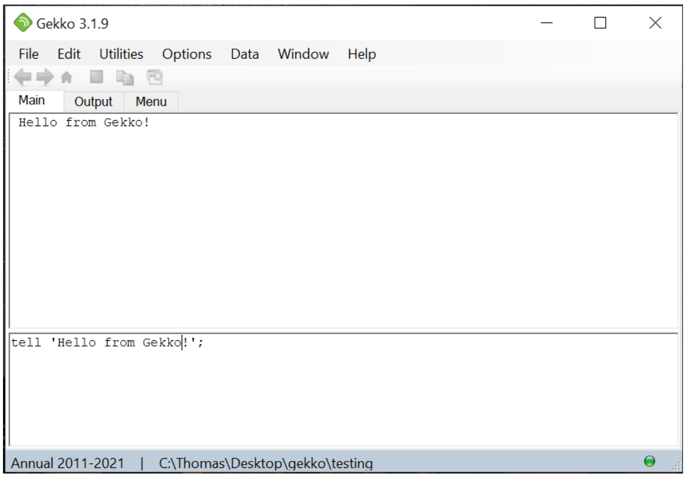
- If Gekko is started up from the 'Programs' menu in Windows, Gekko will open up the last-opened working folder. You may change this by means of 'File' --> 'Set Working Folder...' in the Gekko menu.
- If Gekko is started up from the Windows command prompt, Gekko will use that particular folder as its starting folder. Typing 'gekko' only works if a `gekko.bat` file is available, see 'Utilities' --> 'Make .bat file for easy Gekko startup...'. (This `gekko.bat` file should be put somewhere in your Windows path -- Gekko will try to put it into your Windows folder).

If a file with the name `gekko.ini` is present in the program folder (where `gekko.exe` is located) or in the working folder, this file will be executed at Gekko startup. Typically such a file contains OPTION, TIME, MODEL and READ statements setting up the environment for different kinds of analyses. The `gekko.ini` file will be rerun when issuing a RESTART statement (or an INI statement to just run `gekko.ini`), so this statement is in effect equivalent to closing and reopening Gekko (in contrast, the RESET statement omits loading the `gekko.ini` file).

For more advanced users, there is the possibility to indicate parameters when calling the `gekko.exe` file at Gekko startup. See the RUN help file for more on this.

1.3 User interface

The Gekko GUI (graphical user interface) looks as follows:



```
Hello from Gekko!
```

```
tell 'Hello from Gekko!';
```

Annual 2011-2021 | C:\Thomas\Desktop\gekko\testing

There are 3 tabs: "Main", "Output" and "Menu". The main tab is divided into two parts: the lower part is the command input window, and the upper part is the command output window. The output tab is sometimes used to show detailed lists etc, so that these do not clutter the main output window. The menu tab is for showing user-designed menus and tables. You may jump between the tabs by means of `Ctrl+M`, `Ctrl+O` and `Ctrl+U`.

Help on a particular statement is also available by means of typing `help;` or `help [statement name];` in the statement prompt (or you may press `F1`). The bottom bar shows the current global time period, loaded databanks, and current working folder. At the right-side of the bottom bar are red, yellow or green "traffic lights". Green means that Gekko ended the job successfully and awaits new input, red means that Gekko encountered an error and awaits new input, and yellow means that Gekko is working on a job. You can double click on the traffic light to see a list of currently executing jobs (practical for tracking the progress of long-running jobs).

You may use the `F2` key to see the list of open databanks.

List of function keys etc.

Function keys are used for quick access to specific Gekko statements. At the moment, only a few function keys are active.

F1	Opens up the help system. You can also type for instance `help;` or `help sim;` from the command line, in the latter case you will get help on that particular statement (SIM).

F2	Opens up the databanks window (close with Esc). Note that the `Ref`, `Local` and `Global` databanks do not show up in this window if they are empty. You may also use SERIES? to see what kinds of timeseries the databanks contain.
Tab	Starts variable autocompletion: first place the cursor at the (partial) variable name. Gekko will try to match the variable name, and will show a list of matching names to choose from. You may use autocomplete on a variable like `ab`, where Gekko will try to match the pattern `ab*` (Gekko implicitly adds the trailing `*`). You may also add a databank name, like `bank1:ab` to look for `ab*` in `bank1`. You may set `*` or `?` wildcards explicitly, but in that case Gekko will not add any trailing `*` on itself. Symbols `@` (Ref databank) or `!` (frequency) can be used, and array-series can be searched by using autocomplete on for instance the partial name `x[ab` to match all `x[ab*]`, or on `x[`, which will match all `x[**]` (returns all elements of the arrayseries, regardless of the number of dimensions). Autocomplete looks in open databanks and only works on timeseries names, not scalar or collection names. The list of matching names contain labels as popups. Note that `[Ctrl+Space]` can be used, too. Autocomplete uses the INDEX statement internally. Anything that can be used as input for the INDEX statement can be used for autocomplete, too.
Enter	If you hit `[Enter]` on a line without trailing `;`, Gekko will automatically add the `;` for you. If you hit `[Enter]` in the middle of a line not ending with `;`, Gekko will complain and not add the `;` automatically. Note that with `option interface edit style = ...` , you may swap the use of `[Enter]` vs. `[Ctrl+Enter]`.
Ctrl+Enter	New line in command prompt, without issuing the command line. Note that with `option interface edit style = ...` , you may swap the use of `[Enter]` vs. `[Ctrl+Enter]`.
Ctrl+Space	Starts variable autocomplete, cf. `[Tab]` above.
Mark+Ente r	You may mark several lines in the Main window and execute them as one block with `[Enter]`. This is functionally equivalent to putting the lines in a Gekko program file (.gcm) and executing them with RUN. Note that with `option interface edit style = ...` , you may swap the use of `[Enter]` versus `[Ctrl+Enter]`.

Ctrl+M	Jump to Main tab.
Ctrl+O	Jump to Output tab.
Ctrl+U	Jump to Menu tab.

As mentioned, you may double-click the 'traffic light' indicator in the lower right of the interface to open up the 'Run status' window.

The left- and right arrow buttons below the menu are for browsing back and forth when showing DISP (equation browser), or when showing tables by means of menus. The 'home' button navigates back to the start.

The 'Stop current job' red button tries to halt an executing job.

The 'Copy last ...' button/icon at the top of the main Gekko window copies the last PRT/MULPRT, table, matrix, etc. as spreadsheet cells on the clipboard, for subsequent pasting into a spreadsheet (similar to CLIP). This is convenient for copy-pasting to for instance Excel.

There is also a button to close all PLOT and DECOMP windows at once.

1.4 Notes and issues

Gekko 3.0 contains quite a lot of new features, and a cleaned up syntax. The syntax is hopefully more logical and consistent than version 2.0/2.2/2.4/2.5.x, and some of the most important changes regarding syntax are listed below.

At the end or this page, you will also find a list of issues with 3.0 and 3.1.x. These minor issues will be fixed in the form of patches to version 3.0 and 3.1.x.

Regarding lists of new statements, new built-in functions and new options, see the new features page, under Gekko 3.0. It is probably beneficial to read that section first, before reading the rest of the current page. See more on stability of the software below.

Stability

In general, Gekko 3.0 (release version) and 3.1.x (development version) are stable in the sense that smaller glitches may occur, while major bugs are not expected. There is currently a syntax freeeze, so the 3.1.x line of development versions will stay compatible with Gekko 3.0, and care is taken that the 3.1.x versions are as stable as the 3.0 version. Read more about the different Gekko versions here.

In general, it is **advised to use the latest version in the 3.1.x development line** rather than 3.0, and to migrate from Gekko 2.0/2.2/2.4/2.5.x if possible.

Besides the new components available in 3.1.x that are not present in 3.0, there is also a continued polishing going on in 3.1.x, which makes it a more pleasant experience than 3.0. Besides, the help system is much improved in 3.1.x relative to 3.0, and Gekko error messages will continually be made more understandable in 3.1.x (this is a project planned for the spring of 2021).

Gekko 3.0/3.1.x reads all databanks made by 2.x versions, and Gekko 2.5.x can read databanks from Gekko 3.0/3.1.x. So transferring databanks between Gekko 2.x and 3.x should not pose problems.

Beware

There is an automatic translator from Gekko 2.0 (or 2.2/2.4) to Gekko 3.0. See TRANSLATE, or more info here. The syntax changes compared to Gekko 2.0/2.2/2.4 programs are not too dramatic, the most important are the following:

- Since series statements are calculated in a vector-like fashion in Gekko 3.0/3.1.x (in contrast to the Gekko 2.x versions), lags no longer automatically accumulate period-for-period, if the left-hand side variable is present with lags on the right-hand side (lagged dependent/endogenous variable). See more on the help page on dynamic statements. In Gekko versions >= 3.1.7, a statement like for instance `x = x[-1] + 1` will need to be decorated with an indication regarding whether it should accumulate (`<dyn>`) or not (`<dyn = no>`), cf. examples on the linked help page. Note that for absolute accumulations like `x <dyn> = x[-1] + 1`, you may use `x <d>= 1`, `x ^= 1` or `dif(x) = 1` instead (and there are also variants regarding relative accumulations, see below). Note that setting `<dyn>` (or `block series dyn = yes`) unnecessarily entails a speed penalty and should therefore not be used unless needed.
- Symbols on scalars and collections must appear on the left-hand side too, for instance `val %v = 100;`, where `val v = 100;` is no longer legal. Note that assignment statements SERIES, VAL, DATE, STRING, LIST, MAP, MATRIX may be omitted, so you can use `%v = 100;` too.
- In general, when defining a list, the elements are enclosed in parentheses, but the 'naked' form `#m = a, b, c;` is allowed as short-hand for `#m = ('a', 'b', 'c');`. For lists of simple numbers, naked lists can be used, too, for instance `#m = 1, 2, 3;` or `y = 1, 2, 3;`.
- Beware that `for %i = #m;` is no longer legal, you must indicate type: `for string %i = #m;`.
- The NAME statement is deprecated, and in many places {}-curlies must now be used where they could be omitted in Gekko 2.0. For instance you must use `prt {#m};` or `prt {%s};` to print the variables corresponding to the the list of strings `#m` or the string `%s` (without the {}-curlies, the list elements or the string itself would be printed).
- Name compositions like `a{i}b` must now be `a{%i}b`: the `%`-symbol can no longer be omitted here (or anywhere else).
- Name concatenation like `a%i|b` is no longer endorsed, but will still work. It is generally better to use `a{%i}b`, for readability and consistency.
- `#m[0]` cannot be used to get the length of a list, use `#m.length()`.

- Using `#m[%s]` to check if the string `%s` is a member of the list of strings `#m` will be deprecated, and the expression `%s in #m` or `#m.contains(%s)` should be used instead.
- Series operators like `+`, `*`, `%`, etc. are now `+=`, `*=`, `%=`, etc., so assignments always contain the `=` symbol.
- List operators: `&+` is changed to `||`, `&*` is changed to `&&`, and `&-` is changed to `-`.
- Scalars inside quoted strings should use {}-curlies, for instance `%s = 'car'; tell 'The {%s} is red';`. Alternatively, you can use `tell 'The ' + %s + is red';`, which is harder to read. Beware that `tell 'The %s is red';` will no longer in-substitute `%s`. Inside a quoted string, any expression can be used inside {}-curlies, as long as it evaluates to a string or value type.
- IMPORT and EXPORT statements from Gekko 2.0 without time indication should be changed into IMPORT<all> and EXPORT<all>, respectively. In Gekko 3.0, IMPORT and EXPORT without time indication will use the global time period, potentially truncating the data.
- The following functions have been changed (see details here): avgt(), sumt(), piece(), search(), strip(), trim(), difference(), hpfilter(), fromseries().

Issues list

The following is a list of statements etc. that are known to be defunct in in Gekko 3.0:

- DECOMP. DECOMP works for expressions and equations. There are still issues regarding DECOMP of equations if the left-hand side is not equal to the right-hand side. Therefore: DECOMP of equations is ok if performed on simulated equations, but not on non-simulated equations.
- TIMEFILTER only works for annual frequency.
- User-defined procedures and functions have a problem with samples and composed series arguments, if these are later on lagged. For instance: `function series plus(series x1, series x2); return x1 + x2[-20]; end; time 2001 2003; y1 = 3; y2 <1981 1983> = 2; print y1, y2, plus(y1, y2+0);` This will print missings for the result of `plus(y1, y2+0)` whereas `plus(y1, y2)` will be fine. This is being looked into.

Part II

2 Search

To search the help system, you may try a site-restricted Google search via the `site:` parameter. For instance, if you search for `daily frequency site:http://t-t.dk/gekko/docs/user-manual` in Google, you will get results related to "daily frequency" from the newest help system.

- Search the help system ("daily frequency" example)

There is some inbuilt search capabilities in the non-pdf help system versions, but these are limited. A Google search like the example seach above will prioritize search results from the newest help system, but hits from older help systems may show up, too (if the search result contains for instance the text "user-manual-2.4", this indicates that it is from the Gekko 2.4 help system).

Notes

To seach the help system corresponding to Gekko 2.4 (and 2.5.x), you may use the parameter `site:http://t-t.dk/gekko/docs/user-manual-2.4`.

In the pdf version of the manual, you may use Ctrl+F to search the whole help system for words/phrases.

Using `site:http://t-t.dk/gekko/docs/user-manual` searches the full help system, not only the user manual part of it. For historical reasons, the sub-folder containing the help system is called `/user-manual`. This path will be changed into `/help-system` later on.

Part III

3 Gekko User Guide

The Gekko User Guide provides an overview of the program, with a lot of examples, and highlighting only the main points. The Gekko User Guide is aimed at beginners and new users, and does not presuppose anything, except some curiosity.

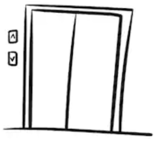
If you need more details or explanations, consider the Gekko User Manual, where you can find detailed descriptions of statements, functions, and other components.

There are presently the following parts of the user guide:

- Elevator pitch (why use Gekko?) Trying to explain the main points of Gekko.
- Data management guide. A guide regarding data management in Gekko.
- Modeling user guide. A guide regarding modeling in Gekko.

The data management and modeling guides are designed for beginners, being relatively easy to understand, with a lot of examples, and with less details than what is presented in the Gekko user manual. In the user manual, you can find conceptual descriptions of Gekko's syntax and capabilities, together with exhaustive descriptions of the different statements and functions that Gekko offers. If the user manual feels overwhelming and you would prefer something more destilled, the Gekko User Guide is the right place to start!

3.1 Elevator pitch (why use Gekko?)



Before delving into the data management and modeling user guides, first a so-called elevator pitch. An elevator pitch is a brief summary that it should be possible to deliver in the time span of an elevator ride. It is, of course, possible to skip directly to the Data Management User Guide or the Modeling User Guide. But please consider spending a modest amount of time to read or at least glance through the following brief listing of the main points of the Gekko software. And if this plea does not persuade you, please consider revisiting the current page at a later time!

Free and open-source programs like R or Python also manage data and can even run models, so what is so special about Gekko being timeseries-oriented? The following tries to explain this, highlighting some of the special concepts and syntax that Gekko uses in order to offer convenient timeseries management.

Timeseries-oriented

- Gekko operates with the concept of a global time period, so you normally do not need to state time, *t*, in timeseries expressions. Instead, you just use the timeseries names themselves, for instance `y = 2 * x;`.
- Gekko has a concept of global frequency, so you normally do not have to state the frequency of a timeseries variable, unless it differs from the global frequency. To denote a frequency explicitly, use the `!` symbol, for instance `x!q` for a quarterly timeseries.
- Lags, leads, time differences, percentage growth, etc. have easy syntax, for instance `x[-1]`, `x[+1]`, `dif(x)`, `pch(x)`, etc. You may also use `x.1` as short-hand for `x[-1]`.

Scalars and collections

- Timeseries variables are exempt from prefix symbols, whereas scalars use `%` prefix symbol (for instance `%x`), and collections use `#` prefix symol (for instance `#x`).
- All Gekko scalars have prefix symbol `%`, for instance `%x = 100;`. Scalars can be value, date, or string.
- All Gekko collections have prefix symbol `#`, for instance `#x = ((1, 2), 3);` for a nested list of values. Collections can be list, matrix, or map.

Databanks and comparisons

- Gekko operates with the concept of a hierarchical list of open databanks, in which the first-position databank is the so-called `Work` databank when Gekko starts up. A databank reference can be stated explicitly with colon (`:`), for instance `b1:x` to indicate the timeseries `x` from the databank `b1`. If a databank name is not stated, Gekko will look for `x` first in the first-position databank, next in the second-position databank, and so on (unless such databank searching is switched off).
- Besides the list of open databanks, Gekko operates with a so-called `Ref` databank. You may use for instance `prt <m> x;` to print absolute differences between the `x`'s in the two databanks (besides `<m>`, there are a lot of other comparison operators). The `Ref` databank is not part of the databank hierarchy.
- You may open an external Gekko databank file (.gbk file) as a 'named' databank, adding it to the end of the databank hierarchy. Alternatively, you may copy or

merge the contents of an external databank file into the first-position databank with READ or IMPORT.

- You may use the special `Global` databank to store global variables/settings, for instance `global:%tstart = 1980;`.

Variable names

- The name prefix symbols `%` and `#` (for scalars and collections, respectively), and the frequency symbol `!` are all to be interpreted as integrated parts of the variable name, alongside the more standard characters like `a`, `b`, ... `z`, etc. (+ digits and underscore).
- A variable can be uniquely identified by means of: `bank` + `:` + `symbol` + `variable` + `!` + `freq`, where `bank`/`variable` = a simple name, `symbol` = `%` or `#`, and `freq` = a frequency character. For instance, `b1:x1!q` looks in the databank `b1` for the variable `x1` with `q` (quarterly) frequency, or `b2:%y1` looks in the databank `b2` for the scalar variable `%y1`. Timeseries names have no prefix `symbol`. Often, bank and/or frequency may be omitted.
- The names of variables residing in a given databank can be indexed and searched in different ways, using wildcards, ranges, etc. In some statements, 'naked' wildcards can be stated as simple as `x*`, whereas in other statements `['x*']` or `{'x*'}` must be used (more about wildcard logic here).

Lists

- Lists can contain any Gekko variable, and a list of strings is often used to represent variable names (or name-parts). It is often more convenient to operate on lists of *names* of variables, rather than on lists of the variable objects themselves. There are two ways to define lists: **strict syntax**, or as **naked lists**:
- Strict syntax: `#m = ('a', 'b');` is a list of the two strings `'a'` and `'b'`. Note the use of parentheses and quoted strings.
- Naked list: `#m = a, b;`. Parentheses and quotes can omitted for such simple lists. The result is the same as above.
- Nested lists (lists of lists) can represent multidimensional data, for instance "cells" from spreadsheets etc.

Name-composition

- Gekko uses `{}`-parentheses to insert name-parts from a string or list of strings. For instance, `%i = 'a'; prt x{%i}y;` is the same as `prt xay;`. The `{}`-parentheses can contain any expression that evaluates to a string or a list of strings.

- You may interpret the `{}`-parentheses as "quote-removers", cf. the equivalence `x{'a'}y` = `xay`.

- Lists of strings can be used in name-composition, too. For instance `#m = a, b, c; prt x{#m}y;` is the same as `prt xay, xby, xcy;`.

- Instead of for instance `x{%i}y` or `x{#m}y`, you *may* use `x%i|y` or `x#m|y`, but this syntax is no longer officially endorsed.

- The `{}`-parentheses work inside strings too, for instance `%i = 'a'; %s = 'x{%i}y';` is the same as `%s = 'xay';`.

- Instead of name-composition, it is often convenient to use array-series, with syntax like `x[#m, y]`. More on this page.

Operators

- Operators are often used in PRT (print), PLOT or SHEET (spreadsheet) for viewing data.

- Time-differences use operators like `<d>` for absolute time-difference, or `<p>` for percentage growth

- Bank-differences use operators like `<m>` for absolute difference between the first-position and the `Ref` databanks, or `<q>` for percentage differences.

- The operators can also be used in timeseries statements, for instance `x <p>= 2;` or `x <q>= 2;`, the first making `x` grow 2% per period, and the last augmenting `x` with 2% (same period). Other operators are available too, for instance `x %= 2;` (grows 2% per period) or `x += 1;` (adds 1, same period).

- There is a special kind of operator, `@`, where `@x` is short-hand for `ref:x`, that is, fetching `x` from the `Ref` databank.

Statements, functions and procedures

- Statements can be stated with or without a `<>`-field to indicate 'local' options like the local time period. For instance, `PRT <2020 2030> x;` will use the period 2020-30 in the PRT statement, but not alter the global time period. Gekko functions and procedures also support an optional local time period.

- Gekko functions support so-called Unified Function Call Syntax. So for instance the function `y = f(x1, x2, x3)` can alternatively be called with `y = x1.f(x2, x3)`, moving the first argument out of the function and adding a dot (`.`). You may also use local time period in functions, like `y = x1.f(<2010 2020>, x2, x3)`.

Options

- The user can state general options with the OPTION statement, whereas local options can be stated inside the `<>`-field of individual statements. A local option inside `<...>` angle brackets is only in effect during the execution of that particular statement.

Trivia

- Gekko is in general case-insensitive. So `prt <p> x;` is the same as `PRT <P> X;`.
- Commas (`,`) are generally used to separate lists of elements (and function arguments).
- You must use semicolon (`;`) at the end of a statement. In the Gekko input window, `;` is set automatically when `Enter` is pressed.

More capabilities

Disclaimer: this section does not highlight the modeling capabilities of Gekko.

Gekko has a lot more capabilities than what has been listed above. What was listed above can be said to illustrate what is special about Gekko, and where its perceived advantages lie. To read about some more capabilities: +toggle.

- Gekko has its own open databank format (.gbk), but also interfaces with a lot of other databank formats (see IMPORT/EXPORT).
- Converting between frequencies, COLLAPSE and INTERPOLATE.
- Copy, rename, move, delete objects between databanks.
- Conditions (IF) and loops (FOR).
- Linear regression (OLS) with extra possibilities.

- Seasonal correction (X12A).
- Operate on timeseries metadata, for instance labels, units, etc.
- There are quite a lot of MATRIX possibilities (linear algebra), and conversions to/from timeseries is easy.
- The MAP variable type is essentially a mini-databank, handy for storing variables by name, or used to return multiple variables from a function call.
- There are a lot of possibilities and conveniences regarding array-series, and there is a tight correspondence to GAMS syntax for operating on such variables. So for instance, the array-subseries `pop[m, 40]` could represent the population of 40-year old males over time, instead of using a name convention like `popm40` or `pop_m_40` to store the same data in a normal timeseries. Array-series can be combined with sums, lists, and $-conditionals (known from GAMS).
- R- and Python code/programs can be run from within Gekko, and there is also the so-called Gekcel add-in for running Gekko from within Excel.

3.2 Data Management User Guide

At present, Gekko is used for many things, but the two foremost use cases are probably data management and modeling. Regarding the latter, there is a user guide available for modeling here. The present data management user guide does not cover modeling at all, except a bit on how to compare two databanks (also relevant for modeling).

The data management user guide contains the following sections:

- Download etc.
- Data management basics
- Databanks and variables
- Gekko programs
- Data reporting

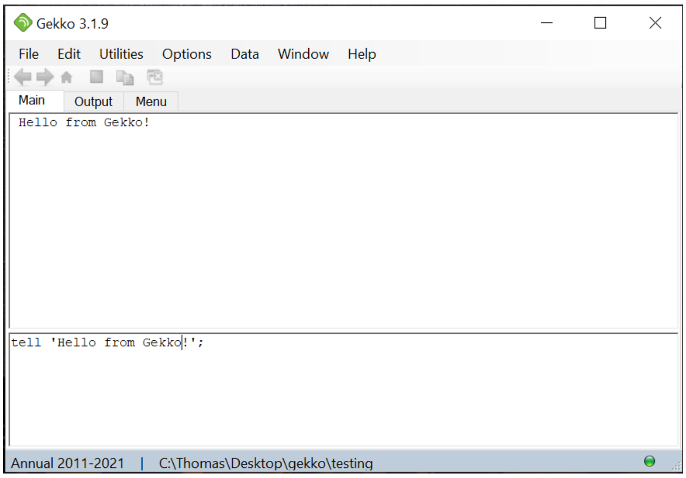
Note: Sections on "Data management" (strategies for managing data), and "Testing databanks" (consistency checks) are forthcoming.

3.2.1 Download etc.

This part of the user guide will focus on the data handling capabilities.

To download and install Gekko, see the general installation section.

You may consult the page on interface, function keys, etc. to read more about how the interface works. But for now, just note that you type statements in the lower part of the window, and results are shown in the upper part. You issue statements with `[Enter]`, and if you omit the ending `;` in a statement, the user interface will add it on its own.


Gekko 3.1.9
File Edit Utilities Options Data Window Help
Main Output Menu

```
Hello from Gekko!
```

```
tell 'Hello from Gekko!';
```

Annual 2011-2021 | C:\Thomas\Desktop\gekko\testing

3.2.2 Data managements basics

Before we start out the data management user guide, a few general words on what "data management" is. You may skip this part of the user guide if you prefer, and move on to the section on databanks and variables.

So what is "data management"? Essentially, we may think of data as arriving in more or less raw (or at least less refined) form. Then the is processed, checked, adjusted, aggregated, and so on, refining it in the process. This can be done in many ways, among the most simple is to fire up a spreadsheet and manage the data there, and among the most complicated is performing big data analytics on massive amounts of data.

Raw data will ofte be represented as "tables", where each row of the table represents an "observation", and where each column represents some characteristic (dimension) of the data. We might for instance envision columns representing sex, age, year and value regarding population data, where one row might represent for instance, sex = male, age = 40, year = 2020, value = 60.000, stating that in 2020, there were 60.000 40-year old males.

This "observation" could be represented as a variable `pop`, where we could assume that `pop[male, 40, 2020] = 60.000`. What Gekko does about data management is to assume that all (or at least most) variables are in the time dimension, so instead of `pop[male, 40, 2020]`, Gekko would talk about the timeseries `pop[male, 40]`, which would be defined over some period, for instance 1980-2020. Such a series could be represented as a co-called array-timeseries in Gekko. But in practice such timeseries are often represented via naming conventions, providing simple names: for instance representing the timeseries as `popmale40` or `pop_male_40`.

So what Gekko data management is about is essentially the "wrangling" of timeseries like `popmale40` or `pop_male_40`, adjusting them, filling out holes, printing them, looking at graphs, etc. Gekko is timeseries-oriented, which among other things entails that timeseries are "first-class citizens" of Gekko, and a lot of functionality is built around that concept. For instance, you can typically omit stating the time dimension explicitly, and concepts like percentage growth rates, lags and leads, frequencies, etc., are an integral part of the Gekko language.

But to return to the concept of data "wrangling", Gekko data management is often about data wrangling of timeseries variables. The purpose of Gekko is that such timeseries wrangling should be relatively easy to do, without very complicated syntax or very complicated concepts. Granted, some parts of the Gekko data management capabilities *are* rather complicated, but a lot of it does not entail a very steep learning curve.

The following sections is an attempt to describe these not-too-complicated parts in a hopefully understandable manner.

3.2.3 Databanks and variables

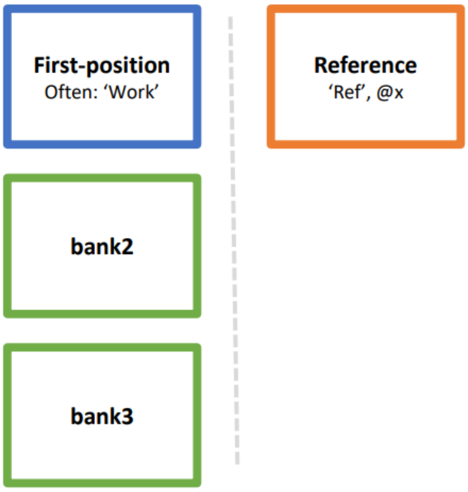
This section describes how Gekko can handle and manage data, using databanks to store different types of Gekko variables. The section covers the databank concept, and how frequencies, time periods, and timeseries work in Gekko. There are also sections on copying and importing/exporting data. In addition, it describes how Gekko can interact with a particular online database API to fetch timeseries data online.

Read more about databanks and variables in the sub-sections.

3.2.3.1 Databanks

The data management capabilities of Gekko has a lot to do with databanks. Databanks store Gekko variables, for instance timeseries, lists or scalar variables. The databanks can be read and written as external .gbk databank files, which is Gekko's own open databank format. When read into Gekko, databanks are in-memory, bounded in size only by available RAM. Databanks can contain many variables, so beware that the keys `[Tab]` or `[Ctrl+Space]` offer autocompletion on timeseries names (cf. here).

Hierarchical databanks list



As seen on the left side of the illustration above, Gekko operates with the concept of a hierarchical list of open databanks. When Gekko starts up, only an empty `Work` databank is present in the first position in the databank hierarchy, providing a kind of working area. Besides `Work`, you may add other databanks to the databank hierarchy with the OPEN statement. The databanks are hierarchical in the sense that when issuing a statement like `prt x;`, Gekko will first look for the variable `x` in the first-position databank, then in the second-position databank (`bank2` above), then in `bank3`, and so on.

If you need to, you may specify a databank name explicitly with colon `:`, like for instance `prt bank2:x;`. In that case, `x` is searched for only in `bank2`. Left-hand side variables like `y` in `y = 2 * x;` are always taken from the first-position databank only, unless a databank name is provided explicitly (like for instance `bank2:y = 2 * x;`). There are options to control whether Gekko searches the databank hierarchy for a variable like `x` or not.

Gekko also operates with the concept of a reference databank (called `Ref`). This databank is often used for comparisons, for instance in modeling, but in data management programs the existence of `Ref` can be safely ignored. And rest assured, in a statement like `prt x;` or `y = 2 * x;`, Gekko will never search the `Ref` databank for the `x` variable. To access a variable from `Ref` directly, you may use for instance `prt ref:x;` or `prt @x;`, where the `@` operator is short-hand for `ref:`.

As mentioned, when Gekko starts up, the `Work` databank can be thought of as the 'working area' in which timeseries and other variable types are stored per default. The `Work` databank can be filled with data via for instance the READ statement (reading an external Gekko databank .gbk file), or alternatively with IMPORT (obtaining data from different supported formats, for instance .csv, .xslx., etc.). If you subsequently want to save the variables of the `Work` databank, you must either save it as a Gekko .gbk databank file with WRITE, or export all the variables to a format that the EXPORT statement supports.

As an alternative to writing a databank with WRITE, you may use OPEN<edit> to open a databank .gbk file as editable in the first position in the databank hierarchy.

After this, you may define and alter variables in that databank, and when you CLOSE the databank again, all changes to it are automatically written back to the same .gbk file.

Whether you use READ and WRITE, or alternatively OPEN and CLOSE to change the contents of a Gekko databank file is basically a matter of taste: data management programs tend to use OPEN/CLOSE more, whereas modeling programs tend to use READ/WRITE more. One thing to note about READ is that it first removes any variables already existing in the first-position databank, including helper variables you may have set previously, for instance time periods, file paths, etc. So when using READ, such variables could instead be put into the so-called Global databank, more on this at the end of this page.

Some relevant databank statements:

OPEN	Opens a .gbk databank as read-only, in the last position in the list of open databanks.
OPEN <edit>	Opens a .gbk databank as editable in the first position in the list of open databanks. If the corresponding .gbk file does not exist, it will be created.
LOCK/UNLOCK	Can be used to set a databanks as read-only (LOCK) or editable (UNLOCK). Often OPEN <edit> is sufficient, and note that a normal OPEN statement without <edit> option always opens a databank as read-only.
CLEAR	CLEAR clears a particular databank (deletes the contents). If no bankname is given, the first-position databank is cleared.
CLOSE *	Closes all open databanks (except `Work`) in the databank hierarchy, and saves all 'named' databanks to files, if these have been changed. Databanks can also be closed individually. The `Work` databank cannot be closed because it serves as a kind of 'working environment'.
RESET/RESTART	RESET implicitly performs a `close *;` (cf. above), and subsequently deletes and restores everything in Gekko, as if the Gekko main window had been closed and opened up again. With RESTART, user-defined options etc. can be reloaded.

Gekko's own file format .gbk is the default format and used for READ/WRITE/OPEN/CLOSE, but as mentioned, Gekko can handle a number of other file types, which is described in the user guide section on import and export.

We will start out creating two databanks from scratch for later use. You do not need to understand the following code, just run it to create a couple of Gekko databanks for later use.

```
reset;
```

```
time 2010 2020;
open <edit> bank1; clear bank1;
x1 = 100; x2 = 200;
close bank1;
open <edit> bank2; clear bank2;
x1 = 101; x2 = 201;
close bank2;
```

The following example shows how to open up databanks:

```
reset;  //resets everything
open <edit> bank; clear bank;
open bank1 as b1;
open bank2 as b2;
```

When Gekko starts up, or when reset or restarted, there is only one databank in the databank hierarchy: the empty `Work` databank. The second statement above opens up the file `bank.gbk` in the first position in the databank hierarchy (note that `bank` is also cleared, to erase any data that might be already stored, if `bank.gbk` already exists). Opening `bank` in the first position demotes `Work` to the second place in the hierarchy. Subsequently, two more databanks are opened with the normal OPEN statement, which puts databanks at the end of the databank hierarchy (as read-only). In the third statement above, `bank1.gbk` is put into the third position, with the alias name `b1`. Finally, `bank2.gbk` is put into the fourth position, with the alias name `b2`. In the Gekko window, you can hit the `F2` button to see the databank hierarchy, which will show the following:

Priority	Name	Filename
1.	bank	bank.gbk
2.	Work	
3.	b1	bank1.gbk
4.	b2	bank2.gbk

As mentioned above, you can always refer to a variable from a specific databank using a colon `:` like this:

```
open bank1 as b1;
open bank2 as b2;
prt <2020 2024 n> b2:x1, b1:x1, b2:x1-b1:x1;
```

In this example, the values of the timeseries `x1` in each of the two databanks, together with the difference between the values are printed. The `<n>` option ensures that only levels are printed (no percentage growth rates).

So to sum up: An `open<edit>` will open that databank at the *top* of the databank hierarchy list (and in this case demote `Work` to second position), whereas a normal OPEN opens the databank at the *bottom* of the databank hierarchy.

The HDG statement allows you to include meta-information about a databank which will be shown when it is read or opened later on. For instance:

```
open <edit> bank1;
hdg 'Historical databank, period = 2010-2020';
close *;
// -----------------------
open bank1;  //test that it has a new heading
```

This yields the following when the bank is read or opened:

```
--------------------------------------------------------------------------------
| DATABANK bank1                                                               |
--------------------------------------------------------------------------------
| Info      : Historical databank, period = 2010-2020                          |
| Date      : 08-02-2021 09:40:28                                              |
| File      : c:\gekko\testing\bank1.gbk (vers: 1.2)                           |
| Period    : The file contains data from 2010-2020                            |
| Size      : Read 2 variables from file into cleared Work databank (0.46 sec) |
| Note      : Press F2 for info on databanks                                   |
--------------------------------------------------------------------------------
```

The contents of a databank can be erased with the CLEAR statement:

```
clear bank1;
```

If you need to clear the contents of the first-position databank, you can instead just use this:

```
clear;
```

You may also close any open databanks with `close *;`. The following code cleans up databanks:

```
close *; clear;  //clear here will clear the Work databank
```

Here, it should be noted that the `Work` databank can never be opened or closed, since it is to be understood as a kind of 'working area' always present. The two statements above will in general provide a 'cleaned up' working environment, closing all banks besides `Work`, and also deleting all variables already in `Work`. But some things live on in Gekko after a `close *; clear;`, for instance options, loaded models, functions and procedures, and other things. To also clear these things, you must use RESET or RESTART (or close and reopen Gekko).

The Global databank

As mentioned above, READ and WRITE are often used in modeling scenarios, whereas OPEN and CLOSE are often used for data management purposes. Still, READ and WRITE can advantageously be used for data management too, and may sometimes be more practical to use than a hierarchy of open databanks. As mentioned, this is a matter of taste, but if you plan to use READ and WRITE, beware that READ reads an external .gbk databank file into the first-position databank (typically the `Work` databank), after first *clearing its contents*.

For instance:

```
reset;
%ti = 2000;  //resides in the Work databank
read bank1;
fyi <1980 2020> = fy/fy[%ti]*100;  //will fail, cannot find %t
```

Here, we start out defining an index year `%ti`, which we plan to user subsequently (`%ti` is a scalar, more on scalars later on). However, any variables you may have already defined in the first-position databank will be deleted after issuing a READ, and therefore the third statement fails, because it cannot find `%ti`. You may use READ<merge> to avoid `%ti` being deleted, but then the first-position databank is 'polluted' with `%ti`, which you may not want to be included in the .gbk databank file, when you WRITE later on in your program.

Instead of this, you can put such helper variables into the so-called `Global` databank:

```
reset;
global:%ti = 2000;  //resides in the Global databank
read bank1;
fyi <1980 2020> = fy/fy[%ti]*100;  //now it finds %ti in the Global
databank
write bank1;  //will include the series fyi, but not the scalar %ti
```

Like the `Work` databank, the `Global` databank always starts out empty when starting up Gekko, and can not be opened or closed. When looking for a variable, `Global` will be searched *last* for this variable, after all other databanks in the databank hierarchy have been searched. The variables in the `Global` databank both survive a `READ` and a `close *; clear;`, but `Global` is cleared after a RESET/RESTART.

An alternative to the above methodology is to use OPEN<edit> and CLOSE, for instance:

```
reset;
%ti = 2000;  //resides in the Work databank
open <edit> bank1;  //Work will now be in 2. position in databank
hierarchy
```

```
fyi <1980 2020> = fy/fy[%ti]*100;  //finds %ti in Work
close bank1;  //writes the changes to bank1.gbk, %ti is not
included
```

With OPEN and CLOSE, it is easier to avoid that the databank gets 'polluted' with irrelevant helper variables, and this is perhaps the reason why OPEN and CLOSE is used a bit more than READ and WRITE for data management programs.

3.2.3.2 Timeseries

Note: There is an additional kind of timeseries available in Gekko, namely so-called array-timeseries. These are explained in a later section.

Timeseries variables can be defined and transformed easily, and variables from any open databank can be used in a series expression. The syntax is as follows:

```
var1 <per1 per2> = value  |  list of values  |  expression;
```

The variable name, `var1`, is here the name of the timeseries that is created or assigned with values. You may prepend SERIES to the statement to indicate the variable type, but this is not necessary (see more on this: +toggle)

In Gekko versions prior to 3.0, you had to start the statement with the SERIES keyword, for instance

```
series <2010 2020> y = 2 * x;
```

This is not necessary in Gekko versions >= 3.0, where you can just write

```
y <2010 2020> = 2 * x;
```

Note the placement of the local `<>` option field between the left-hand side variable and the `=` symbol. The SERIES keyword is not necessary, because Gekko knows from the variable name `y` that it must be a timeseries (names of other types of variables like scalars and collections must start with `%` or `#` symbol).

If only one value is specified, this value will be applied to the entire update period. If several values are specified, the number of values must match the number of periods. The following examples set values for the timeseries `var1` over the period `per1` to `per2`:

```
var1 <per1  per2> = value;
var1 <per1  per2> = value_1, ... , value_n;
```

The var1-var4 examples below shows the use of a value, a list of values, or an expression:

```
time 2021 2023;
%v = 120;

var1 = 100;
var2 = 110, 120, 115;
var3 = var1 + var2;  //210, 220, 215
var4 = (%v-10, %v, 100+15); //note parentheses

prt var1, var2, var3, var4;
```

Note that var4 is not simple numbers like var2, and therefore the right-hand side has to defined with parentheses (list syntax). It is possible to define the variable over a specific period of time. For example, if you want to form a Laspeyres chain price index, with 2010 equal to 1, on the basis of two quantities in current prices, q1 and q2, as well as their prices p1 and p2:

```
reset;
time 2010 2015;
p1 = 1, 1.02, 1.04, 1.04, 1.06, 1.05; //price
p2 = 1, 0.99, 1.03, 1.06, 1.05, 1.08; //price
q1 = 10, 12, 9, 14, 13, 15;           //quantity
q2 = 20, 19, 22, 23, 22, 25;          //quantity

// --- compute laspeyres chain indexes ---
p <2010 2010> = 1;      //aggregate price index
p <2011 2020 dyn> = p.1 * (p1*q1 + p2*q2)/(p1.1*q1 + p2.1*q2);
q = (p1*q1 + p2*q2)/p;  //aggregate volume index
prt p, q;

// --- use in-built function ---
#p = p1, p2;  //prices
#q = q1, q2;  //quantities
#m = laspchain(#p, #q, 2010);  //#m is a map
prt #m.p, #m.q;
```

Result:

```
                   p        %            q        %
  2010        1.0000        M      30.0000        M
  2011        1.0016     0.16      31.0000     3.33
  2012        1.0359     3.42      30.9101    -0.29
  2013        1.0546     1.80      36.9242    19.46
  2014        1.0557     0.11      34.9329    -5.39
  2015        1.0708     1.42      39.9247    14.29
```

```
                #m.p         %          #m.q         %
2010          1.0000         M       30.0000         M
2011          1.0016      0.16       31.0000      3.33
2012          1.0359      3.42       30.9101     -0.29
2013          1.0546      1.80       36.9242     19.46
2014          1.0557      0.11       34.9329     -5.39
2015          1.0708      1.42       39.9247     14.29
```

First of all, note that in general, you can state a timeseries lag more compactly as `x.1` instead of `x[-1]`. In the formula, `(p1*q1 + p2*q2)/(p1.1*q1 + p2.1*q2)` means taking the total expenses in the numerator, and the total expenses in the previous year's prices in the denominator. This fraction is used to construct the price index `p` successively year for year. Note also the `<dyn>` tag in the formula. Without it, Gekko will complain that because of the accumulating lag `p.1`, you need to decorate the statement with either `<dyn>` or `<dyn = no>` (the latter would not make sense here). Read more about `<dyn>` and the reason why Gekko does not just run with implicit `<dyn>` tags added in all cases like this: +toggle.

As mentioned, Gekko will detect if you omit a `<dyn>` tag in a statement like `x = x[-1] + 1;`, where the dependent variable `x` appears with a lag on the right-hand side. Such a statement must be decorated with `<dyn>` or `<dyn = no>`, otherwise Gekko fails with an error. What `<dyn>` does is in reality quite simple. In a timeseries statement like `x <dyn> = x[-1] + 1;`, instead of running the statement in one go over the whole period (let us assume that the period is 2020-2023), it will run the statement period-by-period in four steps: in 2020, in 2021, in 2022, and in 2023. In that way, lags will accumulate, and `x` will augment with 1 for each period. If the statement is not run in four steps, it does not accumulate, because timeseries are treated like vectors in Gekko, cf. this explanation page with illustrations etc.

But why doesn't Gekko run such statements stepwise period-by-period on itself? Among other things, the reason is that some kinds of statements cannot be run interchangeably with `<dyn>` or `<dyn = no>`, like for instance the following: `x = x[-1] + (1, 2, 3, 4);`. To a human, it may seem obvious that this means augmenting `x` with 1 in 2020, 2 in 2021, 3 in 2022, and 4 in 2023. But the statement cannot just be run for one period at a time: there is only 1 period and 4 values, and Gekko will abort with an error because the number of observations and the number of values do not match. Work-arounds for such cases could perhaps be invented, but simplicity is preferred. The simplicity lies in the fact that with `<dyn = no>` tag, a series statement like `x = x[-1] + 1;` is run vector-like in one go (without accumulation), whereas with `<dyn>` tag, the statement is run period by period, accumulating.

A radical solution to this problem would be to decide that *all* series statements should be run period-by-period (that is, with an implicit `<dyn>` tag). This would create other problems, however. Perhaps the worst of these problems is that it would make all series statements run much slower, because of the period-by-period looping. Normally, timeseries calculations in Gekko operate like vector arithmetics, for instance adding or multiplying vectors (arrays) on a low machine level, whereas period-by-period looping operates at a much higher and slower machine level. Therefore, unnecessarily decorating a series statement with `<dyn>` as in `y <dyn> = 2`

`* x;` basically makes the statement run *n* times slower, where *n* is the number of time periods. The statement still yields the same results as without `<dyn>`, but for longer time periods and many series statements, the slowdown would be really bad.

Instead of constructing the Laspeyres chain index by hand, Gekko has an in-built function to compute these values. The function `laspchain()` returns a map, containing the two timeseries (aggregated price and quantity).

The following is an example of changing the base year of a variable:

```
reset;
time 2010 2015;
p1 = 1, 1.02, 1.04, 1.04, 1.06, 1.05;
p1a = p1 / p1[2012]; //p1a is 1 in 2012
prt p1, p1a;
```

The growth rates of `p1` and `p1a` are identical, but `p1a` has value 1 in 2012 rather than in 2010.

In the following example, the series `x1`, `x2`, `x3`, and `x4` are added together into a sum series called `xsum`.

```
reset;
time 2010 2015;
x1 = 1; x2 = 2; x3 = 3; x4 = 4;
xsum = x1 + x2 + x3 + x4;
prt xsum;  //value: 10
```

Instead of this, you can define a list of series names and use the `sum()` function to sum over this list:

```
//...continued
#xlist = x1, x2, x3, x4;  //naked list, list of strings
xsum = sum({#xlist});     //you can also use sum(x1, x2, x3, x4);
prt xsum;  //still 10
```

Note the use of `{}`-parentheses. First, it should be noted that we are using a so-called 'naked' list definition, being short hand for the more strict `#xlist = ('x1', 'x2', 'x3', 'x4');`, more about this in the user guide section on lists. So the elements of `#xlist` are the four strings `'x1'`, `'x2'`, `'x3'`, `'x4'`, and not a list of four timeseries objects. Using `sum(#xlist)` would amount to `sum('x1', 'x2', 'x3', 'x4')`, and in that case Gekko will complain that it cannot extract a value from a string type. In a sense, the `{}`-parentheses 'eat' quotes (`'`), transforming the sum into `sum(x1, x2, x3, x4)`.

If confused about the use of `{}`-parentheses, try to read the elevator pitch.

The user may aks why Gekko does not just 'know' that `sum('x1', 'x2', 'x3', 'x4')` is to be understood as the sum of the variables that these string names refer to? But such contextual interpretation is bound to create other problems. For instance, when printing out `#xlist`, how should Gekko 'know' whether the user wants to have the four raw strings printed, or the four timeseries that the strings refer to? The PRT statement here has the exact same `{}`-logic:

```
//...continued
prt #xlist;    //prints the strings 'x1', 'x2', 'x3', 'x4'
prt {#xlist};  //prints the series x1, x2, x3, x4
```

It is often convenient to use name-parts ('codes') instead of full names, and use lists to compose the names from these name parts. For instance, you may use a name-composition with the name `x{#xcodes}` in the following way: `#xcodes = ('1', '2', '3', '4'); prt x{#xcodes};`. This also prints `x1`, `x2`, `x3`, and `x4`. Note that `#xcodes` does not use the syntax `#xcodes = 1, 2, 3, 4;`, because in that case, the four elements become values, and Gekko will complain that it cannot convert values directly into strings (but you could use `#xcodes = #xcodes.strings()` to convert the list elements into strings).

In addition to `sum()`, there is a number of other functions that can replace the corresponding expressions on the right side of the equals sign.

Transformations and operators

Gekko also allows you to use a number of operators for easy transformations. Some of the most used are the following:

`^=` absolute time-change:

```
x ^= 15;               //absolute time-change, same as x <d>= 15;
x <dyn> = x[-1] + 15;  //corresponds to this
```

`%=` relative time-change:

```
x %= 15;                //relative time-change, same as x <p>= 15;
x <dyn> = x[-1] * 1.15; //corresponds to this
```

`+=` absolute change (same period)

```
x += 15;     //absolute change (same period), same as x <m>= 15;
x = x + 15;  //corresponds to this
```

`*=` relative change (same period)

```
x *= 1.15;     //absolute change (same period), same as x <q>= 15;
x = x * 1.15;  //corresponds to this
```

`#=` change in growth rate

```
x #= 15;                         //absolute change (same period),
same as x <mp>= 15;
gx = x/x[-1];                    //corresponds to ...
x <dyn> = x[-1] * (gx + 0.15);   //... these two lines
```

Instead of `^=`, `%=`, `+=`, `*=` and `#=`, you may use the equivalent `<d>`, `<p>`, `<m>`, `<q>`, or `<mp>` operators, for instance `x <d>= 15`; instead of `x ^= 15;` These `<>` operators correspond exactly with the corresponding operators used in PRT, PLOT, SHEET, etc. There is also a third possibility, namely using left-hand side functions, for instance `dif(x) = 15;` which is equal to both `x ^= 15;` and `x <d>= 15;`

The choice of operator type is mostly a question of taste, and perhaps habit.

If `<keep=p>` is used as option, Gekko will keep the growth rate of the left-hand series intact after the period over which the series is updated. For instance, `x <2020 2025 m keep=p> x = 100;` will add 100 to `x` over the period 2020-25, since `<m>` is the operator for absolute change (same period). The `keep=p` setting makes sure that the growth rate of `x` regarding 2026 and on is the same as before the update.

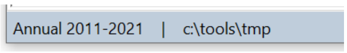
Data tracing

Gekko timeseries variables support data tracing, that is, the ability for a given timeseries to "remember" how it has been constructed. Consider the statements `x1 = 1; x2 = 2; x = x1 + x2;`. Data tracing implies that `x` remembers that it has been produced according to the expression `x1 + x2`, and `x` will also remember how both `x1` and `x2` were constructed (as sub-traces: even if `x1` and `x2` are subsequently deleted). Data tracing also works through Gekko databanks (.gbk files). Think "Trace Precedents" from Excel, just more powerful. See more here.

3.2.3.3 Frequency and time period

When working with timeseries, setting the global frequency and the global time period is usually the first thing to do. When Gekko starts up, the default frequency is annual (`a`), and the time period is set from `t`-10 to `t`, where `t` is the current year.

The frequency and time period settings control a wide range of statements, for instance creation of new timeseries, databank search, transformations, estimation, printing, etc. It is recommended to keep an eye on how the frequency and time period is set, and to aid this, the current frequency and time period are displayed in the lower left corner of the main Gekko screen:


Annual 2011-2021 | c:\tools\tmp

You may change the global time period with TIME:

```
reset;
time 2010 2020;
%t = 2021;
time %t-10 %t;  //using a scalar date %t
```

The time period includes both the starting and ending period. You may use scalars like `%t` instead of fixed dates, to control time settings more dynamically.

With `option freq ... ;`, the global frequency can be set:

```
reset;
option freq q;
time 2020q1 2020q4; //4 quarters
option freq d;
time 2020m1d1 2020m12d31; //366 days from Jan. 1, 2020 to Dec. 31,
2020.
```

This shows the use of quarterly (`q`) or daily (`d`) frequency. In both cases, you may alternatively use `time 2020 2020;`, and Gekko will transform this into the same time periods with 4 and 366 observations, respectively. (In general, Gekko tries it best to translate 'foreign' frequency dates into current frequencies).

There are the following frequencies available in Gekko:

Name	Frequency	Example	Periods
a	Annual	`time 2020 2022;`	3
q	Quarterly	`time 2020q1 2022q4;`	12
m	Monthly	`time 2020m1 2022m12;`	36
w	Weekly	`time 2020w1 2022w52;`	157
d	Daily	`time 2020m1d1 2022m12d31;`	1096
u	Undated	`time 0 100;`	101

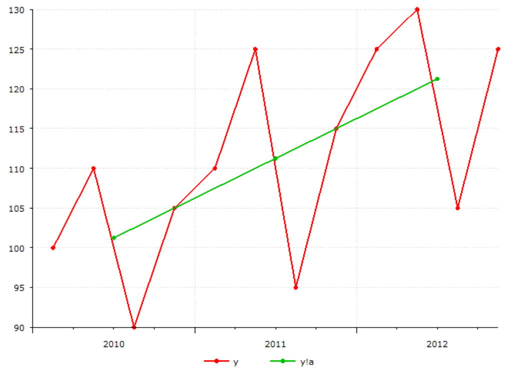
Undated frequency can be used to create, for instance, a timeseries defined over different ages, so that `x[30]` corresponds to 30-year olds, etc. It could also represent hours measured from some starting point, or any other repetitive data. Note that week numbering follows the ISO 8601 standard. This means that weeks around New Year are basically "strange" in the sense that the last days of particular year may belong to the first week of the subsequent year, and the first days of a particular year may belong to the last week of the preceding year (week number 52 or 53: the total number of weeks varies from year to year).

The COLLAPSE statement is used to convert a higher frequency time series into a lower frequency timeseries. If, for example, you only have quarterly data for a series `y`, it can be converted into a corresponding annual series by taking the average over quarters:

```
reset;
option freq q;
time 2010q1 2012q4;
y!q = 100, 110, 90, 105, 110, 125, 95, 115, 125, 130, 105, 125;
collapse y!a = y!q avg;
prt y!q, y!a;
plot y!q, y!a;

//Note: above, instead of y!q, you could just use y, because the
frequency is
//       set to quarterly at the top. The !q is added here for
clarity.
```

Result:

```
                  y         %            y!a         %
2010
q1         100.0000         M
q2         110.0000     10.00
q3          90.0000    -18.18
q4         105.0000     16.67
a                                   101.2500         M

2011
q1         110.0000      4.76
q2         125.0000     13.64
q3          95.0000    -24.00
q4         115.0000     21.05
a                                   111.2500      9.88

2012
q1         125.0000      8.70
q2         130.0000      4.00
q3         105.0000    -19.23
q4         125.0000     19.05
a                                   121.2500      8.99
```



Here, we note the frequency symbol `!`, which is used to distinguish the annual series `y!a` from the corresponding quarterly series `y!q`. In general, if the global frequency is for instance quarterly, you do not need to decorate/suffix every quarterly timeseries `y!q` with `!q` , but can just use `y`, where Gekko implicitly adds the `!q` behind the scenes. The frequency indicators are mostly used when mixing frequencies.

As arguments to the COLLAPSE statement, you can choose between `total`, `avg`, `first`, or `last`. When using `total`, you get the sum of the observations, with `avg` the average, while `first` and `last` use the first and last observation, respectively. Gekko also supports auto-collapse when printing, for instance `prt <collapse=avg> y!q;` (this will print yearly averages).

If the global period is not undated, and if you use an integer sufficiently small, Gekko adds 1900 to the year. This means, for example, that 95 becomes 1995, and 125 becomes 2025 (the latter variant is less used, but `time 80 2020;` instead of `time 1980 2020;` can be practical enough).

Date scalars

In many contexts it is practical to use pre-defined date variables to set the different time periods. Scalar variables are often used for this purpose, that is, values, dates and strings (cf. also the user guide section on scalars). Such variables are often used for settings, for instance path strings that determine the location of different databanks, or date scalars that set different time periods. This is especially practical in large Gekko program systems, where the statements, for example calculations and updates etc. are unchanged, but where the desired period over which the calculations are to be made changes.

Gekko scalars (strings, values, and dates) must start with the symbol `%`, for instance:

```
reset;
//general settings
%per0 = 2012;
%per1 = 2015;
%per2 = 2020;
%path = 'e:\forecasts\scenario1\';
%bank = 'long.gbk';
```

```
%factor = 1000;
// --------------------------------
time %per0-1 %per2;
open {%path}{%bank} as b;
prt <%per1 %per2> b:gdp/%factor 'GDP';
```

Such settings variables may occur in many places in subsequent Gekko program files in a larger data management system. When new data arrives, these settings may change, for instance changing the three date variables to 2013, 2016, and 2021. The location of the databanks may also change, for instance during testing. And a variable like `%factor` might be set to `1e6` instead, so that printing is done per million rather per thousand. To inspect such scalar variables, the MEM statement is practical, because it shows all scalar variables in all open databanks.

As mentioned in the user guide section on databanks, it can be convenient to place such settings variables in the `Global` databank, because this databank survives both READ, CLOSE *, and CLEAR statements. The `Global` databank is always open, so you can just prefix `global:` on the settings, for instance `global:%per0 = 2012;` (cf. also the GLOBAL statement).

It should be noted that for instance `%per0` in the example above is not really a date, but an (integer) value. In practice, this is not an issue, because `%per0` can be used as a date in all relevant statements and functions. If you really wanted a date, you could indicate the type with `date %per0 = 2012;`, or use `%per0 = 2012a;` but there is not really any advantage of doing this. Dates can be stated in any frequency, for instance `%per0 = 2012q2;` or `%per0 = 2012m4d1;`.

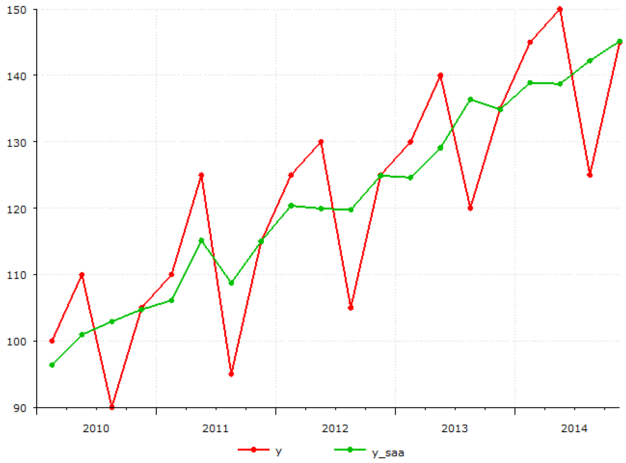
Seasonal adjustment

When working with monthly or quarterly data, there are often systematic fluctuations, recurring depending on the month or quarter. It is often necessary to correct for these seasonal fluctuations. In Gekko, the X12A statement is used for seasonal correction.

If you have a quarterly series `y` that needs to be seasonally adjusted, you can write:

```
reset;
option freq q;
time 2010q1 2014q4;
y = 100, 110, 90, 105, 110, 125, 95, 115, 125, 130, 105, 125, 130,
140, 120, 135, 145, 150, 125, 145;
x12a y; //Returns the variable y_saa as seasonally adjusted
plot y, y_saa;
```

Result:



You may fine-tune the X12A statement with special parameters, cf. the statement description.

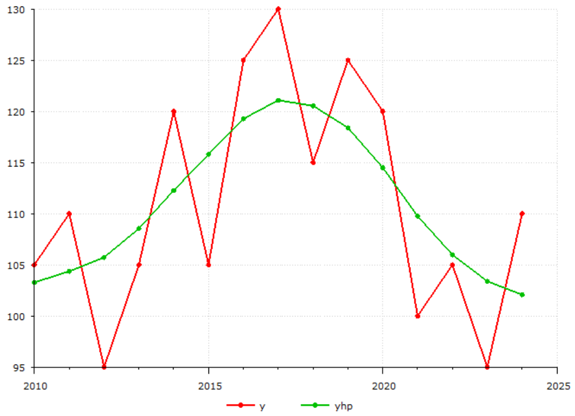
Structural levels

When working with monthly, quarterly or annual data, one is sometimes interested in finding the structural levels. One way to find the structural level of a series is by using a Hodrick–Prescott filter (HP filter), which inputs an economical time series `y[t]`, it adjusts a trend `g[t]` by solving a minimization problem.

The syntax for using a HP filter in Gekko is for instance:

```
reset;
time 2010 2024;
y = 105, 110, 95, 105, 120, 105, 125, 130, 115, 125, 120, 100, 105,
95, 110;
%lambda = 6.25;
y_hp = hpfilter(y, %lambda);  //or: y.hpfilter(%lambda);
plot y, y_hp;
```

Result:



The statement returns a trend-adjusted variable `yhp` of the variable `y`. Lambda is usually set to 6.25 for annual series, 1600 for quarterly series, and 129600 for monthly series. Usually one finds the structural level of seasonally adjusted variables, so in our case the following: `yhp = hpfilter(y, 1600);`

Note that in general, you may move the first argument of a function out of the function like this: `y.hpfilter(1600)`, instead of `hpfilter(y, 1600)`. This often allows a more "fluent" syntax, and allows chaining like for instance `y.hpfilter(1600).avgt()` for taking the average, instead of the more backwards `avgt(hpfilter(y, 1600))`.

3.2.3.4 Copying etc.

As mentioned in the user guide section on databanks, Gekko databanks can contain different types of variables. Such variables may be copied, rename, moved, etc. between databanks, cf. the following statements:

CLONE	Makes the `Ref` (reference) databank an exact copy of the first-position databank. The statement is often used for modeling, but can also be used for data management, when comparing two databanks.
COPY	Copies variables inside a databank or between databanks. COPY copies all observations of timeseries, unless COPY<respect> is used. Wildcards and ranges can be used.
DELETE	Deletes variables in a databank (by name or wildcards/ranges)
INDEX	Searches a databank for variables (with wildcards or ranges). Can also search inside array-series.
RENAME	Renames variables in a databank (by name or wildcards/ranges). Can also be used to move variables between databanks (without copying).

In general, regarding statements like the above, it should be noted that a seemingly all-encompassing wildcard like `*` will only match timeseries of the current frequency in the first-position databank. If you need, for instance, to match all timeseries (of the current frequency) in all databanks, use `*:*`, and to match all timeseries of all frequencies (in the first-position databank), use `*!*`. There are also special rules regarding matching of scalars and maps, cf. this page on wildcards.

Some examples:

```
copy x1 to x2;                //make x2 an exact copy of x1
copy <respect> x1 to x2;      //only for the global time period
copy #m1 to #m2;              //copies the *list* #m1 to the *list*
#m2
copy {#m1} to {#m2};          //copies *elements* of #m1 to the
*elements* of #m2
copy b1:x1 to b2:x2;          //copies between databanks
copy b1:x* to b2:*;           //using wildcards
copy b1:x*;                   //copies into the first-position
databank
```

As with other statements, it should be remembered that `copy #m1 to #m2;` will copy the object `#m1` itself (`#m1` may or may not be a list) into the object `#m2`, whereas `copy {#m1} to {#m2};` will copy the variables corresponding to the string names inside `#m1` (`#m1` must be a list) into the variables corresponding to the string names inside `#m2`.

About wildcards, there are some special rules regarding these in relation to the symbols `%`, `#`, and `!`, cf. the user guide page on lists (or a more detailed explanation of wildcard logic here).
The DELETE statement just deletes named variables in a databank, for instance:

```
delete x1, x2, x3;
delete b1:x1, b2:x2;
delete b2:x*;
```

RENAME has the same syntax as COPY, and the only difference between the two is that COPY clones objects/variables, whereas RENAME renames/moves them. For instance:

```
rename x1, x2 as y1, y2;    //renaming two variables
rename x* as y*;            //rename using wildcards
rename b1:x1 as b2:x1;      //moving the variable between banks
```

The INDEX statement searches for variables and returns the results as a list of string names. See more about INDEX on the user guide page on lists. Beware that the keys `[Tab]` or `[Ctrl+Space]` offer autocompletion on timeseries names (cf. here).

3.2.3.5 Import and export of data

There is a difference between reading a .gbk Gekko databank and importing a data file in a 'foreign' format. A similar distincion can be made between writing a .gbk databank and exporting data to en external foreign format. Import and export of data is characterized by the fact that it the files are in non-gbk formats like csv, xls(x), prn, tsd, etc.

The Gekko databank files (gbk) support all Gekko variable types, both series, scalars and collections. Foreign formats do not support all Gekko data types, but a number of formats can be used for transportation of timeseries data. Examples could be the csv, prn, and tsd formats. These are characterized by having a well-described and open format. In addition, many software packages will be able to import and export timeseries to these formats.

Note that Gekko can also exchange data with Excel and gnuplot. In both cases, Gekko can do more than just send data to and from formats supported by Excel and gnuplot and in both cases perform some of the tasks that user would otherwise have to do after exporting the data. The use of Excel or gnuplot as a backend for Gekko is dealt with in a separate section.

Supported transport formats:

Program/format	**File type**	**Import**	**Export**	**Comment**
AREMOS	.dmp	x		AREMOS also supports the tsd format, but .dmp (dump) files have a lot better precision (significant digits).
Apache	.arrow		x	Supported by many programs, including R and Python. Import will come soon, too.

Arrow				
csv	.csv	x	x	An open text format. Many programs read this format, including Excel.
Excel	.xls(x)	x	x	Regarding Excel, there is also the so-called "Gekcel" project, where Gekko statements can be issued from within Excel, without opening up Gekko at all. This project will soon be released.
flat	.flat	x	x	A Gekko-specific text format that resembles Gekko series statements, but is more simple and very fast.
gcm	.gcm	x	x	When exporting, the exported timeseries are exported in the form of .gcm statements that can be used directly in Gekko. Not very fast, cf. the 'flat' format.
GAMS	.gdx	x	x	The gdx format is supported for both import and export. Is often used for array-series, and GAMS sets are interchanged, too.
gnuplot	.dat		x	Gekko can export data in a form suitable for gnuplot.
PCIM	.bnk	x		PCIM has previously been used for modeling. Gekko can import its binary databank format.
pr	.prn	x	x	A csv-like open text format.

n				
PC-Axis	.px	x		Gekko can import PC-Axis px files (these are text-based). See also the DOWNLOAD statement.
Python	.py		x	Gekko can export data as a Python script. Alternatively, use the Apache Arrow format.
R	.r		x	Gekko can export data as a R script. Alternatively, use the Apache Arrow format.
tsd	.tsd	x	x	An open text-based format used by AREMOS and EViews.
TSP	.tsp	x	x	Can import and export TSP records.

Import of data

IMPORT and READ are similar, but READ is meant for Gekko's own .gbk databank format, for instance loading the file `data.gbk` like this:

```
read data; //Gekko appends .gbk automatically
```

Regarding timeseries, READ gets all observations regardless of the global time period, and READ also first clears the first-position databank, before data is loaded.

In contrast, the IMPORT statement restricts import to the global time period (unless otherwise stated), and merges data with existing data in the first-position databank. Therefore note that IMPORT may not import all observations from the data file, and existing values outside of the global time period will remain unchanged. Use IMPORT<all> to import all observations and replace existing timeseries completely. If you need to IMPORT into an empty databank, you can use CLEAR first.

Example:

```
import <tsd> data.tsd;      //will only import for the global time
period
import <tsd all> data.tsd;  //imports all data for all observations
in the data file
```

When reading csv and xls(x) files, one must be aware that Gekko per default reads the time series in rows, and expects the first column to contain series names, and the first row to contain dates. If the time series are in columns (transposed), this must be specified with the `<cols>` option in IMPORT:

```
import <csv cols> data.csv;   //series running downwards in columns
import <xlsx cols> data.xlsx; //series running downwards in columns
import <prn> data.prn;
```

Regarding prn, this format is similar to csv, only it uses space as delimiter, rather than semicolon, and the first "cell" (corresponding to A1 in a spreadsheet) inside the prn file contains info on whether the timeseries run outwards in rows, or downwards in cols.

Export of data

EXPORT and WRITE are similar, but WRITE is meant for Gekko's own .gbk databank format, for instance writing the file `data.gbk` like this:

```
write data; //Gekko appends .gbk automatically
```

Regarding timeseries, WRITE writes all observations regardless of the global time period. In contrast, the EXPORT statement restricts export to the global time period (unless otherwise stated). Therefore note that EXPORT may not export all observations from the first-position databank. Use EXPORT<all> to export all observations. It is also possible to store one or more selected variables in the .gbk databank file:

```
write x1, x2, x3 file = data; //Three series, Gekko appends .gbk
automatically
```

The EXPORT statement is very similar, for instance (here, the csv format is used: the file is indicated with extension .csv for clarity, even though Gekko would add the extension automatically):

```
export <csv> data.csv; //All series, global period
export <csv all> data.csv; //All series, all observations
export <csv> x1, x2, x3 file = data.csv; //Three series, global
```

```
period
export <csv all> x1, x2, x3 file = data.csv; //Three series, all
observations
```

The EXPORT statement supports Excel xls(x) files, but export of data to Excel can also be done using the SHEET statement, where you can control the sheet name, starting cell, and the orientation (columns or rows) of the timeseries. Regarding Excel, the difference between EXPORT and SHEET can summed up like this:

- EXPORT<xlsx> exports timeseries to a matrix of cells starting in cell A1, where the first row is dates, and the first column is series names. Only raw values are exported, no transformation is possible, and EXPORT<xlsx> is very similar to EXPORT<csv>.
- SHEET is essentially a print or plot statement in disguise, allowing all the same transformations and expressions used in prints or plots, for instance the use of operators, mathematical expressions, functions, etc. SHEET also allows choosing the sheet name, starting cell name, and it can suppress period and/or name labels to only show raw data, etc.

The SHEET statement is therefore a flexible way of transferring data to Excel. Excel must be installed for SHEET to show the results, and the statement does not work with e.g. LibreOffice Calc or Google Sheets. For this, it is recommended to use EXPORT<csv>, since the csv format is supported by most spreadsheet programs. Alternatively, it is possible to use the CLIP statement, which is equivalent to SHEET, but places the result on the clipboard for later pasting with Ctrl-V.

As a minimum, the SHEET statement must specify which variables are to be transferred to Excel, e.g.:

```
sheet x1, x2, x3;
```

The above statement puts the timeseries `x1`, `x2`, and `x3` into Excel (in rows). No file is saved, and the Excel sheet containing the data is automatically opened by Excel and shown. As mentioned, SHEET is a PRT/PLOT statement in disguise, so it allows all kinds of transformation, for instance:

```
sheet x1/(x1+x2), x3 <p>;
sheet log(x1/x2) 'lx1x2' file = logdata.xlsx;  //changing the label
```

In the first statement, `x3` is shown as percentage growth due to the operator `<p>`. In the second statement, the result is not shown in Excel, but instead the file `logdata.xlsx` is written to disk. The label `'lx1x2'` is set, because a label (name) like `'log(x1/x2)'` is not useable if the file is being re-imported into Gekko from this spreadsheet (with IMPORT<xlsx>).

In the following example, several series are exported from the bank `hist1115.gbk` to Excel:

```
read hist1115;
time 2010 2020;
sheet fY, fM, fCp, fCo, fI, fE;
```

After issuing a SHEET statement, Excel should start up, showing the data.

Piping

A special kind of export is using a pipe. The PIPE statement is generally used to store Gekko output in a file by directing (piping) Gekko's output into the file, instead of on screen. In the following example, the output is saved in a the text file `data.prn`:

```
pipe prt.txt;
prt <n> x1, x2, x3;
pipe <stop>;
```

The `<n>` operator suppresses percentage growth, which is normally shown in PRT, together with levels. After `pipe<stop>;` the file `prt.txt` contains the print, and no output is shown in the Gekko output window. You can use TELL to add text to such a pipe file. Instead of text format, the PIPE statement can also generate output in html format, and with the use of PIPE<html> in combination with PLOT and TABLE, quite advanced "reports" can be produced.

3.2.3.6 Download API

Gekko has an interface to the online API of Statistics Denmark (statbank.dk). This is perhaps not very interesting for non-Danish users, but other online databanks could be supported in Gekko, too.

You can use the DOWNLOAD statement to download timeseries originating from data tables on statbank.dk, without having to go through a manual process. The syntax for accessing the API is:

```
download 'http://api.statbank.dk/v1/data' statbank.json;
```

With the above statement line, Gekko accesses the online API via the provided link and a .json file describing which data to download. The file indicates which table and which variables from that table are to be retrieved. The json file is based on the .json format, which can be thought of as a kind of simplified and more readable xml. For example:

```
{
  "table": "PRIS6",
  "format": "px",
  "variables": [
    {"code": "VAREGR", "values": ["011200", "011100"]},
    {"code": "UNIT", "values": ["100"]},
    {"code": "TIME", "values": ["*"]}
  ]
}
```

With the above specification, Gekko will use the API to fetch the data table `PRIS6` (`"table": "PRIS6"`), in PC-Axis format (`"format": "px"`, which is currently the only format Gekko can use regarding statbank.dk), and the price index (`"code": "UNIT", "values": ["100"]`) retrieve the product groups with the codes `011200` and `011100` (`"code": "PRODUCT", "values": ["011200", "011100"]`) for the entire period (`"code": "TIME", "values": ["*"]`).

The variables are placed in Gekko's work bank and the variables are named according to the following convention:

```
 <table
name>_<dimension1>_<code1>_<dimension2>_<code2>_<dimension3>_<code3> ...
```

When the above DOWNLOAD statement is run, the following output is shown:

```
 -> Download start ...
 PRIS6_VAREGR_011200_ENHED_100, with freq M, 2000m1-2015m12
 PRIS6_VAREGR_011100_ENHED_100, with freq M, 2000m1-2015m12
 -> Downloaded 2 timeseries in total
```

And the two time series `PRIS6_VAREGR_011200_ENHED_100` and `PRIS6_VAREGR_011100_ENHED_100` are located in Gekko's first-position databanke, where the name composition corresponds to this:

`PRICE6`	`<table name>`
`_VAREGR_011200`	`_<dimension1>_<code1>`
`_ENHED_100`	`_<dimension2>_<code2>`

In addition, Gekko shows info about frequency and time period, respectively. For instance, "with freq M, 2000m1-2015m12" means that the frequency is monthly (M), and the dates mean that the data runs from January 2000 to December 2015. If it had instead been 2000m2-2015m11, the data would have run from February 2000 to November 2015.

If you want to retrieve all item groups and unit types from a table, you may use `"*"` inside the square bracket that follows the table group code (`PRODUCT`, `UNIT` and `TIME` in the above). This is e.g. done for the time period in the json file above.

If you wanted to e.g. to retrieve the entire table, `PRICE6`, with price indices for all product groups, the json file should look like this:

```
{
  "table": "PRIS6",
  "format": "px",
  "variables": [
    {"code": "VAREGR", "values": ["*"]},
    {"code": "UNIT", "values": ["*"]},
    {"code": "TIME", "values": ["*"]}
  ]
}
```

The API is a handy tool for downloading multiple statbank data at once. As mentioned, the DOWNLOAD interface could be adapted for other online databanks, too.

3.2.4 Gekko programs

When using Gekko in daily work, you will usually use Gekko program files, if you have a program that takes up more than a few statements, and this statement sequence will be run several times. A Gekko program file is an alternative to typing each statement interactively in the Gekko main window. An additional advantage of using Gekko programs is that the documentation of the performed calculations is done automatically -- it is simply the Gekko program(s).

When using Gekko programs, it is thus possible to review at any time which statements have previously been run in order to produces some results. Gekko program files also provide some protection against errors, since programming with the help of Gekko programs ensures a certain order of the program structure.

A Gekko program is characterized by two properties:

- It contains only Gekko statements.
- It is a system file with file extension: .gcm

To run a Gekko program file with the name `data.gcm`, type:

```
run data;  //Gekko will add extension .gcm
```

All the statements in the file will then be executed exactly as if they were issued interactively from the statement window.
Below is an example of a Gekko program as it will look in Notepad (read more about editors: +toggle).

An editor like Window's in-built Notepad text editor can be used for Gekko programs, but this editor is not particularly advanced. It is advised to use a more capable text edit, because you may need to operate on blocks of code, search and replace in a more advanced fashion, etc. Additionally, but not least, more advanced editors allow code coloring (highlighting). This is the same kind of coloring seen in the above box illustrating the RUN statement: note that the semicolon is blue, and the comments are green. A particular handy possibility is the possibility to feed the editor with a list of Gekko statement names, so that these get their own color.

With a more advanced text editor, such coloring is possible, coloring for instance all the special symbols in one particular color, numbers in another color, text strings in a third color, etc. Some editors can even color different levels of nested parentheses in different colors.

- The so-called Kedit text editor has historically been used a lot together with Gekko, among other things because it can be fed with a syntax coloring definition file. However, Kedit is a more or less discontinued product, and is not free.
- Else, there is Sublime Text, for which there is a free open-source Gekko module available that handles text coloring, but also other things like remote-controlling Gekko. Sublime Text is free for evaluation purposes.
- The possibilities regarding text editors are practically infinite, it basically comes down to personal preference.

```
// ----- file start -----
read hist1115;
ul = ua - q;
prt ul;
// ----- file end -------
```

The above file first loads the databank `hist1115.gbk`. Next, the number of unemployed (`ul`) for the current period is calculated and printed.

It is advised to include comments in Gekko programs. This increases readability both for the user him/herself, but especially for other users who need to understand the contents of the file. Comments can be included in the file with either:

- `// some text`
- `/* some text */`

The former (`// ...`) only out-comments the rest of the current line, wheras the latter (`/* ... */`) can span several lines and is therefore practical for multi-line commentaries, or for out-commenting larger blocks of code (read about other ways of skipping code: +toggle).

You may out-comment code like this:

```
x1 = 1;
/*
x2 = 2;
x3 = 3;
*/
x4 = 4;
```

In that case, only x1 and x4 are calculated (`//` could have been used as well, on the two lines). An alternative is this:

```
x1 = 1;
if(1 == 0);
  x2 = 2;
  x3 = 3;
end;
x4 = 4;
```

Here, activating the two lines is a bit easier, just change into for instance `if(1 == 1);`. Or use a 'control' variable, like `if(%cond == 1);`, where the code is run only if the settings variable `%cond` is set to 1.

To stop execution of Gekko at a particular location, it is advised to insert a STOP statement, for instance

```
x1 = 1;
stop;
x2 = 2;
x3 = 3;
x4 = 4;
```

Now only the first line of the Gekko program file is run, after which Gekko stops, and the user can inspect variables or try to sort out bugs. Gekko cannot resume after a stop, but you may move the STOP statement and run the file again.

Besides gcm program files, Gekko also recognizes files named `gekko.ini` (henceforeward called the 'ini file'). The file usually contains a number of custom options that the user wants Gekko to set at startup. The options can, for example, set the folder structure for which Gekko should look for model files, bank files, and Gekko programs, along with other more general settings like setting the global time period, etc. The ini file is run automatically, if Gekko is started from a folder in which such a file exists (otherwise, the INI statement can execute the ini file on demand, and the RESTART statement resets Gekko and subsequenctly runs the ini file). An example of the contents of an ini file could be:

```
option folder bank = C:\Datop\bank\;
option folder command = C:\Datop\;
run period.gcm;
read hist1115;
time 2010 2020;
```

The first two lines define the folders in which Gekko searches for data banks (`option folder bank`) or Gekko program files (`option folder command`). When bank and Gekko programs paths are defined as above, in subsequent statements it is not necessary to use full path indication if you open up a data bank or run a Gekko program. The third statement loads date variables from an external `period.gcm` file. Finally, a databank is read, and the global time is set.

It is a good idea to start your Gekko program with the following:

```
restart;
```

which closes all data banks, clears everything (including options, etc.), and re-runs gekko.ini. A RESTART is essentially equivalent to closing the Gekko main window, and starting Gekko up again. Related to RESTART is the RESET statement, which also closes all data banks but fails to run any ini file.

```
reset;
```

With RESET, a `gekko.ini` file is not searched for and run. There is the following equivalence: `RESTART` = `RESET; INI;`. Since both RESTART and INI do not abort with an error if no ini file is found, if you need to be absolutely certain that a gekko.ini file is found and run (and no-one accidentally deleted the file), instead of `RESTART;` at the top of your Gekko program, you can use `RESET; RUN gekko.ini;`. In that way, if the ini file is not found, Gekko will stop with an error.

If you need to clear the output window, you can use CLS (clear screen):

```
cls;
```

Neither RESET nor RESTART clears the output window, so `cls; RESTART;` is often seen in combination.

Read more about Gekko programs in the sub-sections.

3.2.4.1 Module organization of command files

Gekko program files can be nested, that is, a Gekko program file calling another program file, which in turn calls a third program file, etc. It can be an advantage to separate large automated runs in several program files, where the individual steps are separated into modules. For example, `program.gcm` can contain several separate steps:

Program.gcm:

```
run step1.gcm; // Reads input data
run step2.gcm; // Converts to aggregated groups
run step3.gcm; // Calculates derived variables
run step4.gcm; // Writes calculated variables to databank
```

There are several advantages to modularity. For example, it makes it possible to reuse a group of statements. It can also provide a clearer structure in more complex tasks, which makes the programs easier to write, troubleshoot and reuse.

However, complications can occur, especially when calling Gekko program files from other program files. Circular calls should obviously be avoided, for instance where `file1.gcm` calls `file2.gcm`, which in turn calls `file1.gcm`.

The following statements are particularly useful in modular/nested program files:

ACCEPT	Can be used for interactive user input. Example: `accept string %yesno 'Do you want to continue? (y/n)';`
PAUSE	Stops the execution of statements, writes the text to the output window, and waits for the user to press [Enter]. Example: `pause 'Click [Enter] to continue';`.
TELL	Example: `tell 'The calculations are now finished';`. Writes text to the output window.
RETURN	Returns from the currently executing program file, and returns to its 'parent' (calling) program file. RETURN is also used for functions and procedures.
STOP	Stops the run and ignores the rest of the cmd file. Also interrupts runs of nested files, in contrast to RETURN.

Instead of using (nested) program files, it is also possible to use user-defined functions or procedures (more on this in a later section). These are defined first, and can be called subsequently. They can be called with arguments, providing ease of use and encapsulation.

When a larger Gekko session is running (for instance running a number of program files), you may use the menu item 'Utilities' --> 'Run status...' to see how Gekko is progressing (or double-click the yellow 'traffic light' at the bottom left of Gekko main window). In addition, you may track which data files are read and written by activating the menu item 'Data' --> 'Track data'. When active, a Gekko session will conclude by showing which data files were read and written. This info can be practical for larger structures of Gekko program files.

3.2.4.2 Scalars

As explained in the section on frequency and time period, scalar variables can practical for storing general settings in the beginning of a system of Gekko program files. Scalars are useful in longer program files, where you easily lose track of for instance time settings, file paths etc. Scalars can be of type value, date or string, and scalars always start with the symbol `%`.

Note: Gekko variables can also be of collection type, for instance list type. Lists can also be practical to define at the beginning of a system of Gekko program files (lists are explained in the following user guide section on lists).

Gekko has three types of scalars:

value	Numeric values, with decimals (also used for integer values). For instance `%v = 25`, `%v = -1.2345`, or `%v = 2.57e9`.
date	Date variable with time indications. For instance `%d = 2012a`, `%d = 2012q4` or `%d = 2012m12`. If you write `%d = 2012`, `%d` will technically be value, but can still be used just like an annual date, no worries. See also "date combining functions" under functions.
string	For example text strings or path names. For instance `%s = 'Price index, 100 = 1'` or `%s = 'm:\data\revision2\'`. See also "string combining functions" under functions.

Notes:

- Values: a particular observation from a timeseries can easily be fetched as a value: `%v = x[2020];`, where `x` is a timeseries. In Gekko, there is no special integer type: for integers, you just use values. Values often represent annual dates, too.
- Dates: you may add or deduct integers from dates, for instance the quarterly `%d1 = 2012q4; %d2 = %d1 + 10;` will add 10 periods/quarters to `%d1`, with result `%d2 = 2015q2` (fourth quarter of 2012 + 10 periods = second quarter of 2015). You may also use date differences: `%d2 - %d1 = 10`.
- Strings: can be concatenated with `+`, for instance `%s = 'rise'; prt 'sun' + %s;`, which prints `'sunrise'`. Alternatively, and often more readable, Gekko also supports string interpolation, where Gekko replaces any `{...}` curlies inside a string with the result of evaluating the curlies. For instance, `%s = 'rise'; prt 'sun{%s}';` will also print `'sunrise'`. String interpolation corresponds tightly with the equivalent concept of name-composition in Gekko, where `%s = 'rise'; prt sun{%s};` will try to find a timeseries with name `sunrise` (note the absense of quotes in this PRT variant).
- Types: you may prepend explicit scalar type, for instance `val %v = 100;`, but normally this is not necessary, because the scalar type is normally given from the right-hand side of the assignment. Beware that, in general, if you add a value and a string, the value will first be promoted to a string, and therefore `prt '2' + 3;`, which print `'23'`, not `5`. Therefore, you may sometimes need to convert, which is done with the functions `val()`, `date()` or `string()`. These functions try to convert to the desired type, for instance `prt val('2') + 3;` will print `5`.

Example:

```
%bank = 'hist1115';
read %bank;
fy <2010 2012> += 1000;
write %bank;
```

The series `fy` from the bank `hist1115.gbk` is augmented by 1000 in 2010-12. Here, we do not need to use `{}`-curlies around `%bank`, (but still this is perfectly legal), because the READ and WRITE statements (and IMPORT and EXPORT, too) support both of the two variants `read hist1115;` or `read 'hist1115';`.

In connection with data revision programs, it may be a good idea to incorporate the names of input banks, output banks and time period settings centrally at the top of a program file, so that it is easy to adjust.

```
// --- Update banks and dates if necessary ---
cls; reset;
%in = 'in.gbk'; // input bank
%out = 'out.gbk'; // output bank
%per1 = 2012;
%per2 = 2014;

// ---- Does not normally need to change ---
read %in;
time %per1 %per2;
var1 %= 2;  //2 percent growth rate
var3 = var1 + var2;
write <%per1 %per2> var3 file=%out;
reset;
```

In the above example, an input databank `%in` is read, time is set to the chosen period, the growth rate of the series `var1` is set to 2% annually, and `var3` is calculated as the sum of `var1` and `var2`. Finally, a databank is written, with the given `%out` name.

If this needs to be performed on two other databanks, or on a different time period, this can easily be adjusted in the top part of the program.

3.2.4.3 Lists

A Gekko list is an ordered sequence of Gekko variables. A list name must begin with the `#` symbol like the other Gekko collection types (matrix and map). Lists are often used to store strings, particularly strings that represent variable names. But lists are practical for many things. For instance, you may store 2-dimensional Excel spreadsheet cells inside a nested list (a list of lists), see example later on in this section.

A simple list example with a list of strings:

```
#m1 = ('x1', 'x2', 'x3');  //strict syntax
#m2 = x1, x2, x3;          //equivalent 'naked list' simple syntax
prt #m1;                   //prints the strings 'x1', 'x2', 'x3'
prt {#m1};                 //prints the series x1, x2, x3
```

Two important things must be noted about the above example. First, the strict syntax for creating a three-element list of strings uses parentheses `()` and quotes `'`. Using `()`-parentheses is the the most general/strict way of defining lists, but in the particular case of lists of strings, a simpler 'naked list' syntax is legal, too (naked due to the omitted parentheses and quotes). For a naked list, strings are not enclosed in quotes, either, cf. the second statement above. It should be remembered, however, that even though the second statement above may look like it inserts three timeseries objects `x1`, `x2`, and `x3`, this is not the case. The naked list definition `#m2` is just a convenient syntactical short-hand for the strict list definition `#m1` above, and inside the resulting list there are in both cases three strings `'x1'`, `'x2'`, and `'x3'`, not three timeseries objects `x1`, `x2`, and `x3`.

Regarding the print-statements, with `prt #m;`, you are printing out the list itself -- that is, raw strings. If you need to print out the variables that the strings *refer to*, you must enclose the list in `{}`-curlies, as shown in the last print statement. Lists are also often used in connection with loops, where the same calculations are performed for each of the list elements individually (cf. the loops user guide section).

You may pick out an individual list element with an `[]`-index, for instance `#m[2]` to pick out the second element of the list `#m`. To see all defined lists, use the following statement:

```
list?;
```

Naked list syntax

If you are combining strings and lists of strings, naked list syntax offers the following easy way of combining them:

```
//naked list combinations
#m1 = b, c;               // ('b', 'c')
#m2 = e,;                 // ('e',) --> note trailing comma!
#m3 = a, {#m1}, d, {#m2}; // ('a', 'b', 'c', 'd', 'e')
```

Note the `{}`-curlies, and note the trailing comma in the `#m2` definition (a naked list of only one element, cf. the section on 'singletons' later on). The use of `{}`-curlies in naked lists is similar to printing lists of variables with `prt a, {#m1}, d, {#m2};`, which would be equivalent to `prt a, b, c, d, e;`. You may use the `rep` keyword to repeat element, for instance:

```
#letters = a rep 2, b, c rep 3;  // ('a', 'a', 'b', 'c', 'c', 'c')
```

The comma `,` is useful for concatenation, and for lists of strings, the naked list syntax is often convenient. Naked lists can also be used to define lists of values or lists of 'codes', for instance:

```
#m1 = 2, 3, 4;         // (2, 3, 4)         values
#m2 = 1, -2.3, 3.4;    // (1, -2.3, 3.4)    values
#m3 = 1, a, 2;         // ('1', 'a', '2')   strings
#m4 = 2, 03, 4;        // ('2', '03', '4')  strings
```

The first two lists become lists of values (value scalars), whereas the last two lists become lists of strings. The main rule is that if all elements are normal values, the elements of the resulting list will be values, too. In all other cases, all the elements will become string types. There are some special rules regarding such naked lists, see more on the naked list page. To convert the elements of a list to a particular type, you may use the `values()`, `dates()`, or `strings()` functions.

Note: When using naked lists, a lot of single quotes can be avoided for long lists of strings. But this convenience may introduce some confusion. On the right-hand side of LIST or FOR statements, you should be aware that there are three different kinds of possible expressions:

```
#m = something;                         //normal assignment
#m = something, something, ... ;        //naked list definition
#m = (something, something, ... );      //strict list definition
```

In the first, there is no comma and no soft parentheses. In the second, there are one or more commas, but no soft parentheses. In the third, there are both commas and parentheses. Now, consider this:

```
#m = a;                 //normal assignment
#m = a, b;              //naked list definition
#m = (a, b);            //strict list definition
```

In the first, you are trying to assign the timeseries `a` to `#m` (this will fail with a type error). In the second, the list will contain the two strings `'a'` and `'b'`. And in the last, the list will contain the two series objects `a` and `b`.

Therefore, in LIST or FOR statements, always take a look at the right-hand side to see if there are commas or enclosing parentheses. On the right-hand side of a naked list definition, other lists must be enclosed in `{}`-curlies, for instance `#m = {#m1}, {#m2};` to append the two lists `#m1` and `#m2`. So why is `#m = #m1;` legal, omitting the `{}`-curlies? That is because there is no comma, and the statement is therefore a normal assignment (the equivalent naked list definition would be `#m = {#m1},;` with trailing comma).

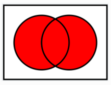
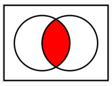
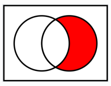
Other operators

You may also combine lists with the `+` or `||` operators:

```
#m1 = a, b, c, d;  // ('a', 'b', 'c', 'd')
#m2 = c, d, e, f;  // ('c', 'd', 'e', 'f')
#m3 = #m1 + #m2;   // ('a', 'b', 'c', 'd', 'c', 'd', 'e', 'f')
#m4 = #m1 || #m2;  // ('a', 'b', 'c', 'd', 'e', 'f')
```

The `#m3` list contains duplicates, because the `+` operator for lists is a simple concatenation of the two lists. In `#m4`, however, duplicates are removed, because the `||` is the union operator. Some commonly used operators and functions:

Some useful operators and functions for two lists

`#m1 || #m2`	Union 	The union of the two lists, no dublets introduced.
`#m1 && #m2`	Intersection 	The intersection of the two lists.
`#m1 - #m2`	Difference 	The difference between the two lists.
`#m1 + #m2`	Concatenation	This is like a 'naive' union where dublets may be introduced if `#m2` contains some of the same elements as `#m1`.

`unique()`
`sort()`
	Functions	There are a large number of list functions available, see more under LIST or under functions. But for instance `unique()` will remove dublets, and `sort()` will sort alphabetically. Such functions may be nested, for instance `#m = #m1.unique().sort();`.

The `+=` and `-=` operators can also be practical when appending to or removing elements from naked lists. To append an element to a list, the `append()` function can also be used, and `prepend()` to prepend. To remove an element, `remove()` can be used. Examples:

```
#m = a, b, c, d;                   //Result: ('a', 'b', 'c', 'd')
#m += e, f;                        //Append using a naked list and +=
operator
#m = #m.append('g').append('h');  //Append using append() function
#m -= g, h;                        //Remove using a naked list and -=
operator
#m = #m.remove('e').remove('f');  //Result: ('a', 'b', 'c', 'd')
#m = #m.prepend('x');              //Result: ('x', 'a', 'b', 'c',
'd')
```

Beware that you cannot use for instance `#m + 'a'` or `'a' + #m` to append/preprend the string `'a'` to the list `#m`, whereas you *can* use `#m - 'a'` instead of `#m.remove('a')` to remove `'a'` occurrences from the list.

To add or remove a string `%s` from a list, the following can be convenient. Note the comma (singleton comma, cf. below). More elements can be put after the comma.

```
#m += {%s},;  //add string %s to the list
#m -= {%s},;  //remove string %s from the list
```

Strict list syntax

If you need fine-grained control over the types of list elements, you can use strict syntax, that is, lists defined with `()`-parentheses. For instance:

```
#m = (1, 'a', 2020q1);   // (1, 'a', 2020q1)
```

In a strict-syntax list definition, `()`-parentheses cannot be omitted, and all string elements must be quoted. Above, the `#m` list contains a value, a string, and a date, so the list elements are all of different type. Other examples where strict `()`-syntax must be used:

```
#m1 = (x1, x2);                         //list of series objects
#m2 = (2001q1, 2002m12);                //list of dates
#m3 = (#m1, #m2);                       //list of lists (nested list)
#m4 = ((x1, x2), (2001q1, 2002m12));   //same as above
```

Instead of the first list (a list of series objects), it is often simpler and more manageable to store the *names* (as strings) of the timeseries (the two strings `('x1', 'x2')`), and then use `{}`-parentheses afterwards when referring to the series objects.

Wildcards

With the INDEX statement, you can use wildcards and ranges to search/index variable names in databanks. For instance:

```
index fx* to #m1;        //wildcard
index fxa..fxn to #m2;   //range
```

The first statement looks in the first-position databank for variable names that match `fx*`, that is, variables beginning with the string `'fx'` (this search is case-insensitive). The result is put into the list `#m1`. If the `to #m1` part of the statement is omitted, Gekko will just print the matching names on the screen, providing an overview of how many variables in the databank start with `'fx'`. Ranges are supported, too: `fxa..fxn` finds names in the alphabetical range `fxa` to `fxn`.

You may choose whether you want INDEX to return "full" variable names with bankname and frequency (like `b1:x1!q`) or not (like just `x1`). Gekko has a lot of functions for dealing with for instance banknames or frequency parts in lists of variable names, for instance `removebank()` or `removefreq()` to remove banknames or frequencies, but also functions for setting or adding those. See more under "Bank/name/frequency/index manipulations" on the functions page.

You may equivalently search/index in the following way:

```
#m1 = ['fx*'];         //wildcard
#m2 = ['fxa..fxn'];    //range
prt ['fx*'];           //prints the matching timeseries names (as
strings)
prt {'fx*'};           //prints the matching timeseries themselves
```

As it is seen from the examples, a wildcard can either be 'naked' like `fx*`, used in for instance the INDEX or COPY statements, or enclosed inside quotes and brackets/parentheses, like `['fx*']` or `{'fx*'}`. The former is used for lists, and the latter is used in for instance PRT, PLOT, etc. The user might ask herself the following question: if `index x*y;` is legal syntax for finding timeseries starting with `'x'` and ending with `'y'`, why is it not possible to use `prt x*y;` to print those timeseries? The reason is that `x*y` can also be understood as `x` multiplied by `y`, so in statements like PRT or PLOT, where mathematical expressions are allowed, you cannot use the 'naked' wildcard `x*y`, but must instead use the more cumbersome `prt {'x*y'};`.

Regarding wildcard searching/indexing of databanks, there are some special rules regarding the special name-characters `%`, `#`, and `!`. Read more about these rules in this +toggle.

About the wildcard syntax, there are some special rules when using wildcards to search/index databanks. Variable names may start with the symbol `%` or `#`, or end with a frequency indicator like `!q` or `!m` (quarterly and monthly). When searching/indexing databanks, a general wildcard like `*` will not match `%` or `#`, so if the first-position databank contains the variables `xy1` (series) and `%xy1` (scalar), the wildcard `*y1` only matches the series `xy1`, not the scalar `%xy1`. Similarly, if the global frequency is set to annual and the first-position databank contains the variables `xy1!a` and `xy1!q` (an annual and quarterly timeseries), the wildcard `xy*` will only match the annual `xy1!a`, not the quarterly `xy1!q`. Therefore, a general wildcard like `*` only matches timeseries, and only timeseries of the current frequency. If you need to match for instance scalars, you can use `%*`, and if you need to match a 'foreign' monthly frequency, you can use for instance `*!m`. Another thing to note is that a wildcard like `fx*` only searches the first-position databank. To search all open databanks for `fx*`, use the wildcard `*:fx*`, or use `b1:fx*` to search inside a specific databank. Read more about these wildcard rules here.

The INDEX statement has options to control whether the resulting list of names includes banknames and frequency indicators. It should be mentioned that indexing works differently when a list rather than a databank is indexed, so for instance `#m['*y1']` will match all elements of the list `#m` ending with `'y1'`, following normal matching rules, with no special treatment of `%`, `#`, and `!`.

Listfiles

You may create and manipulate lists of strings via list files. List files are just text files, where each line contains a string (without quotes), for instance:

vars.lst

```
a
b
c
d
```

If you put the file `vars.lst` into the working folder, you may subsequently use this listfile as if it was a normal list, using the syntax `#(listfile vars)`. The listfile elements may contain characters like `%`, `#` and `!`, but you may also prepend a `-`. The latter can be useful for certain kinds of operations where you for instance want to subtract a variable from a sum. If needed, each line of a list file may contain several elements, delimited with the `;` symbol. In that case, the list will become a nested list, kind of like a mini-spreadsheet where all the cells are strings (such a list will resemble the csv file syntax).

Gekko databank files (.gbk files) can store any kinds of (nested) lists, but sometimes when operating on lists of strings, it is just easier to be able to operate directly on a list in the form of a text file. The following shows how to create and use a listfile:

```
#(listfile vars) = a, b, c, d;        //creates file vars.lst
prt #(listfile vars);                 //'a', 'b', 'c', 'd'
#m = #(listfile vars) - 'c';          //remove 'c'
prt #m;                               //'a', 'b', 'd'
```

Instead of a list file, you may alternatively import a list or a nested list from a spreadsheet (xls(x) or csv file), read more in this +toggle.

Instead of using a list file you may alternatively import cells directly from a spreadsheet (xls(x) or csv file) into a nested list with SHEET<import list> like this:

```
sheet <import list> #m file=data.xlsx;
```

The user manual regarding the SHEET statement contains some examples of advanced "data wrangling" of a nested list of spreadsheet cells ('nested' understood as rows of columns of cells). Such cells need not be only strings: they can be values and dates, too.

One-element lists and trailing commas

The last thing to note about lists is that one-element lists (also called singletons lists) have to be treated in a special way, adding a 'superfluous' trailing comma:

```
//singleton list
#m1 = x1,;       //naked list with comma: result = ('x1',)
#m2 = ('x1',);   //strict syntax with comma: result = ('x1',)
```

The problem with one-element lists is that in general mathematics, `(x) = x`, and circumventing this general rule would introduce all sorts of other problems in Gekko. Therefore, as it is also done in e.g. Python, a trailing 'superfluous' comma is used to indicate that we are defining a one-element list. Without the extra comma, in the first statement above, Gekko would fail because it cannot convert a series `x` into a list, and in the second statement, Gekko would fail because it cannot convert a string `'x1'` into a list.

If you do not like to use trailing commas, you can alternatively use the list() function:

```
#m1 = list('x1');    //one-element list, note: no comma
#m2 = list();        //empty list
```

In the first statement, quotes must be used for the string. The `list()` function without arguments is useful if you need to construct an empty list.

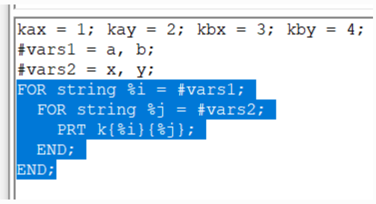
More

You can do many more things with lists and nested lists. Some of the more relevant capabilities are listed below. For an exhaustive explanation of all capabilities, see the LIST statement.

- On the present page, we have mostly dealt with lists of strings. But lists of values or dates are also useful, as is lists of lists (nested lists). Nested lists are practical for representing table-like structures of "cells", for instance the contents of spreadsheets. In contrast to the matrix variable type, with a nested list, you can store dates and strings, too (and not only values). Additionally, nested lists can be of any dimensionality. See more on this page.
- Regarding nested lists, you may for instance use `#m[2][3]` to pick out the second element of `#m`, which is itself a list. From this element (that is, from `#m[2]`), the third element is selected. In such cases, you may alternatively use the syntax `#m[2, 3]`, which would do the same. But beware that there is a difference regarding ranges, where for instance `#m[2..4][3]` is not the same as `#m[2..4, 3]`. More on this here.
- Lists are used a lot for array-series. If you define a list of strings like `#s = a, b;`, and define array-series `x[a]` and `x[b]`, you may use `x[#s]` to refer to these array-series, or `sum(#s, x[#])` to sum them up, etc. See more under SERIES.
- Lists have a close cousin: maps. Maps allow referring to an element with a *name* (rather than picking out with an element number), and can be thought of as a kind of mini-databanks. So instead of picking out an element of a list with an index number (for instance `#m[2]` to pick out element number 2), maps assign a name to each element, so you can instead pick out an element with for instance `#m['%dw']`. This picks out a value corresponding to a Durbin-Watson test (the name `%dw` is easier to remember and understand, than the index number 2...). Instead of the strict `#m['%dw']`, you may also use the equivalent `#m[%dw]` or `#m.%dw`. Note how the last variant resembles `m:%dw`, picking out the variable `%dw` from the databank `m`.
- The INDEX statement can also search inside array-series, cf. INDEX.

3.2.4.4 Loops

Note: If you are running one of the examples interactively in the Gekko input window, you should mark the code from `for` to the corresponding `end`, followed by [Enter]. Executing a FOR loop line by line in the Gekko input window will not work.



```
kax = 1; kay = 2; kbx = 3; kby = 4;
#vars1 = a, b;
#vars2 = x, y;
FOR string %i = #vars1;
  FOR string %j = #vars2;
    PRT k{%i}{%j};
  END;
END;
```

Also, beware that you may often use implicit looping over lists, as shown in the next section.

Loops are used when the same statements or calculations reused over several variables. Loops can simplify and reduce the length of program files considerably. In Gekko, the FOR statement is used for looping, and lists are often used in combination with loops.

Example with lists:

```
kax = 1; kay = 2; kbx = 3; kby = 4;
#i = a, b;
#j = x, y;
for string %i = #i;
  for string %j = #j;
    prt k{%i}{%j};
  end;
end;
```

This will print the timeseries `kax`, `kay`, `kbx`, `kby`, one by one. In this particular case, the user could just use `prt x{#i}{#j};`, and Gekko would auto-unfold and print the same timeseries (cf. the section on array-series), but for many other purposes, fine-grained looping can be practical. You may also obtain roughly the same output with `prt {'k*'};`, but the difference is that the wildcard version includes all variables starting with `'k'`, possibly including irrelevant variables.

When using a FOR loop, the type of the loop variable must always be stated (the same is the case for function and procedure definitions), and a FOR loop must always end with a matching END statement.

FOR loops can also be used for date and value variables. Regarding these types, intervals are often used (and the `by` keyword can be used for step length), for instance:

```
%v1 = 1;
%v2 = 99;
%sum = 0;
for val %v = %v1 to %v2 step 2;
  %sum += %v;
end;
//%sum = 2500 (sum of uneven numbers)

// -----------------------------------------
option freq q;
time 2020q1 2021q4;
y = 1, 2, 3, 4, 5, 6, 7, 8;
x = 100;
%d1 = 2020q1;
%d2 = 2021q4;
for date %d = %d1 to %d2 by 2;
  y[%d] = x[%d];
end;
// y = 100, 2, 100, 4, 100, 6, 100, 8;
// every second quarter skipped
```

Lists may also contain values or dates, and such lists may be looped in the same way as the first example (looping over a list of strings). Synchronized/parallel loops are possible, too, for instance:

```
#i = a, b;
#j = x, y;
for string %i = #i  string %j = #j;
  tell 'Combination {%i}, {%j}';
end;
//-------------------------------------------
//output:
//
// Combination a, x
// Combination b, y
```

Forward Laspeyres chain aggregation can, for example, be made with a loop (this is just an illustration: there is an inbuilt function `laspchain()` that does this automatically):

```
%t1 = 2010;
%t2 = 2020;
#s = 1, 2, 3;           //sectors
#s = #s.strings();      //else they are of value type
py <%t1-1 %t1-1> = 1;   // base year
for date %t = %t1 to %t2;
  y  <%t %t> = sum(#s, p{#s} * fx{#s});               //p1*fx1 +
...
  fy <%t %t> = sum(#s, p{#s}[-1] * fx{#s}) / py[-1];  //p1[-1]*fx1
+ ...
  py <%t %t> = y / fy;
end;

TODO: check the above!
```

The sum() function over lists is practical for repetitive sums like `p1*fx1 + p1*fx2 + p3*fx3` and similar. Such sums are often used for array-series,

3.2.4.5 Array-timeseries and implicit loops

Array-timeseries are a bit more advanced than normal timeseries. The current section also explains implicit looping over lists, which can also be used for normal timeseries.

An array-series `q` may for instance represent production quantities in different sectors `#i` (where `#i` is a list of sector names/strings), and another array-series `p` may represent price levels in the same sectors. Using array-series, you may use expressions like `v[#i] = p[#i]*q[#i]` to calculate the nominal production values in the different `#i` sectors, or `v = sum(#i, p[#i]*q[#i])` to calculate the sum of these values. Such syntax should not be too difficult to understand, since it resembles normal mathematical expressions like the following:

$$v_i = p_i\, q_i \qquad v = \sum\nolimits_i p_i\, q_i$$

Using array-series, it is typically possible to avoid explicit looping over indexes, and perform the looping implicitly in one statement. The Gekko `sum()` syntax also resembles GAMS syntax for summation, which is no coincidence.

But before we move on to array-series, it must be emphasized that much of the same loop-avoidance can be achieved with normal timeseries, too. An example will illustrate this.

Implicit loops with name-composition

As mentioned in previous sections (for instance the elevator pitch), you may use `{}`-name-substitution in series names. Inside the `{}`-curlies, it is allowed to place a list of strings, like for instance:

```
reset; time 2010 2012;
#i = a, b, c; //list of three strings 'a', 'b', 'c'
pa = 1.1; pb = 1.2; pc = 1.3;
qa = 200; qb = 300; qc = 400;
v{#i} = p{#i}*q{#i};
vtot = sum(#i, p{#i}*q{#i});
prt <n> v{#i}, vtot;  // <n> omits percentage growth

//                    va              vb              vc
vtot
//  2010        220.0000        360.0000        520.0000
1100.0000
//  2011        220.0000        360.0000        520.0000
1100.0000
//  2012        220.0000        360.0000        520.0000
1100.0000
```

In the statement `v{#i} = p{#i}*q{#i}` above, implicit looping over the elements of `#i` is performed, so the statement amounts to `va = pa*qa; vb = pb*qb; vc = pc*qc;`. Similarly, `v = sum(#i, p{#i}*q{#i})` amounts to `v = pa*qa + pb*qb + pc*qc`.

A naming scheme like above can work pretty well, using normal timeseries (and it works in multiple dimensions, too). But one of the problems is that for instance the three series `va`, `vb`, `vc` are not coupled in any way, except for their names. For instance, if you need to delete all these nominal production values from the databank, how can you be sure that you really delete all of them? There might be stray `vd` and `ve` series lying around, but do these represent sectors `d` or `e` of the same kind of variables, or are their names just coincidences? So there is a looseness about using naming conventions that the concept of array-series tries to resolve.

Implicit loops with array-timeseries

Array-timeseries provide a tighter coupling of dimensional timeseries data, and also provide other benefits and conveniences. Let us try to reconstruct the above example with array-series instead:

```
reset; time 2010 2012;
#i = a, b, c; //list of three strings 'a', 'b', 'c'
v = series(1); p = series(1); q = series(1);  //1 for 1-dimensional
p[a] = 1.1; p[b] = 1.2; p[c] = 1.3;  //sub-series
q[a] = 200; q[b] = 300; q[c] = 400;  //sub-series
v[#i] = p[#i]*q[#i];
vtot = sum(#i, p[#i]*q[#i]);
prt <n> v[#i], vtot;  // <n> omits percentage growth
index v[**]; //see all sub-series inside v

//                  v[a]           v[b]           v[c]
vtot
// 2010       220.0000       360.0000       520.0000
1100.0000
// 2011       220.0000       360.0000       520.0000
1100.0000
// 2012       220.0000       360.0000       520.0000
1100.0000
//
// v[a], v[b], v[c]
//
// Found 3 matching items
```

This replicates the previous example, where normal timeseries and name-composition were used. So instead of for instance the normal timeseries `va`, `vb`, `vc`, we now have an array-series `v` with sub-series `v[a]`, `v[b]`, `v[c]`. Instead of `index v[**]` here, `v[*]` could have been used (since `v` is 1-dimensional), cf. the INDEX statement.

What is the big deal about that, except for the use of `[]`-brackets to access the sub-series? The following examples try to illustrate the point of using array-series:

```
// ...continued

prt <n> v;  //prints out all of the sub-series

//                   v[a]           v[b]           v[c]
//   2010       220.0000       360.0000       520.0000
//   2011       220.0000       360.0000       520.0000
//   2012       220.0000       360.0000       520.0000

disp v;  //get info on dimensionality etc.

//
===================================================================
=======================
//   SERIES Work: v
//   Annual series has 3 elements in 1 dimensions (period 2010 -
2012)
```

```
//  Dimension 1 (3 elements): a, b, c
//  First/last elements (alphabetically): v[a] ... v[c]
//
===================================================================
=======================

prt p*q/1000;  //some algebra is legal directly on array-series --
               //this presupposes that the p and q sub-series
               //are compatible (contain the same elements)

//         p*q/1000 [a]   p*q/1000 [b]   p*q/1000 [c]
//  2010         0.2420         0.4320         0.6760
//  2011         0.2420         0.4320         0.6760
//  2012         0.2420         0.4320         0.6760

v = p*q;    //this also works, if the sub-series are compatible.
delete v;   //deletes v and all its sub-series in one go.
```

As it can be seen from the above examples, an array-series like `v` can be used more like a matrix or array, for instance when printing it (which prints all its sub-series), DISP-ing it (showing dimensionality, size and element information), or deleting it.

Simple algebra on the array-series themselves is also supported, allowing index-free syntax (like `p*q/1000` instead of `p[#i]*q[#i]/1000`). This can be thought of as being a bit similar to matrix algebra, performing bulk operations on the sub-elements.

In addition, there are other capabilities:

- When you are looping implicitly or summing up over dimensions, you may omit/skip list elements via the `$` conditional operator (similar to GAMS).
- You may assign domains to the dimensions, so that you do not accidentally use an illegal element name (for instance mistyping a sector name).
- Multiple dimensions works too, and it is quite easy to aggregate dimensions (roll-up), or pick an element from a dimension (slicing), etc. You just use commas to separate the dimensions, like `x[a, k]` or `x[#i, #j]`. You may for instance eliminate (aggregate) the second dimension from the array-series `x` like this: `y[#i] = sum(#j, x[#i, #j])`.
- You may assign "default sets", so that `prt v;` only prints out some of the (most important) elements.
- You can use special rules regarding the printing of and calculation with array-series, for instance if there are "holes"/missings in these. You can use the options `option series array print missing = ... ;` and `option series array calc missing = ... ;` to control this.
- There is no risk of name-collisions with array-series. With name-composition there is always a risk of this. For instance if `Kb` means buildings capital, and `q` is a sector name, `Kbq` could mean buildings in the `q` sector. But what if there was another sector with the name `bq`? Couldn't `Kbq` then mean total capital `K` in the `bq` sector? This is often "solved" with underscores like `Kb_q` vs. `K_bq`, but array-series are impervious to this problem, because `Kb[q]` and `K[bq]` are necessarily different.
- Array-series can be sparse. If the first dimension of the array-series `x` has 100 possible elements (for instance 100 sectors), and the second dimension has 100

possible elements (for instance export categories), there can be up to 10.000 different kinds of sub-series inside x. But if there are fewer sub-series than that, corresponding to data being sparse, these "holes" (zero values) will not take up any space/RAM.

- Array-series containing ages (like population data) can be "rotated", so that for instance age profiles can be plotted, cf. the rotate() function, more under PLOT (age profiles section).

More on array-series under the SERIES statement.

Array-series do not yet work in model files (.frm), but this is on the to-do list.

3.2.4.6 Conditional statements

Note: If you are running one of the examples interactively in the Gekko input window, you should mark the code from if to the corresponding end, followed by [Enter]. Executing an IF statement line by line in the Gekko input window will not work.

The IF statement is used to execute statements if a certain condition is met (and an optional ELSE statement will execute other statements if not). The syntax is pretty standard for programming languages, for instance:

```
%x1 = 2; %x2 = 3;
if (%x1 < %x2);
  tell 'true';
else;
  tell 'false';
end;
```

If the condition is true, the following statements are executed, and if the condition is false, ELSE can be used to execute alternate statements. You normally compare scalar expressions via one of the following comparison operators: <, <=, ==, >=, > or <>, and these operators can also be combined with the logical keywords and, or, and not.

You may also compare two whole timeseries, but beware that there are some special rules in play in that case, and there are also special rules when one or both of the values contain missing values (more here or here).

Several terms can be combined with and, or, and not, like in the following example:

```
if (not(x[%t] > 100 or %x == 'yes') and (%x1 > %x2));
  tell 'true';
end;
```

Below is an example of a part of a program where the program file interrupts, if the variable `fvar` has a zero value in the period `%t`.

```
if (fvar[%t] == 0);
  pause 'A price cannot be calculated';
  stop;
else
  //continue normally
end;
```

You can also combine ACCEPT and IF to control the flow of a program interactively:

```
accept string %yesno 'Do you want to multiply x by 1000?';
if (%yesno == 'yes');
  x *= 1000;
else;
  if (%yesno <> 'no');
    tell 'You must answer "yes" or "no"';
    stop;
  end;
end;
```

When comparing, you may need to convert between variable types. In the following example, `%s` is a text string, `%d` is a date, and `%v` is a value.

```
%s = '2012';
%d = date(%s);
%v = val(%d);
mem;  //show these scalars
```

When comparing those, you will need to convert `%s` and `%d` to values, for instance:

```
if (val(%s) == val(%d) and val(%s) == %v and val(%d) == %v);
  tell 'true';
else
  tell 'false';
end;
```

It should be mentioned that Gekko supports `$`-conditionals, similar to how these work in GAMS. These are a kind of inline if-conditions, an example:

```
time 2010 2013;
x1 = 100;
x2 = 90, 110, 80, 120;
y = x2 $ (x2 > x1);  //0, 110, 0, 120
```

In the last statement, Gekko checks the condition for each period, and if it is true, `x2` is used, otherwise the value is 0. Also see the iif() function with similar capabilities.

3.2.4.7 Functions and procedures

> Note: If you are running one of the examples interactively in the Gekko input window, you should mark the code from `function` or `procedure` to the corresponding `end`, followed by `[Enter]`. Executing an `function` or `procedure` statements line by line in the Gekko input window will not work.

It is possible for Gekko users to compose user-defined functions and procedures. This is described in depth on the help pages corresponding to FUNCTION and PROCEDURE. Note that user-defined functions and procedures do not live in Gekko databanks. Instead, it is advised to store them (as Gekko code) in external .gcm program files. Instead of using external .gcm files for functions/procedures, you may alternatively pack these into a .zip file and use this package file as a *library* (cf. the LIBRARY statement).

An example of a user-defined function that multiplies two values:

```
function val mul(val %x1, val %x2);
  return %x1 * %x2;
end;
// --------------
%z = mul(3, 5);
prt %z; //15
```

An example of a corresponding procedure performing the same multiplication and printing it out.

```
procedure mul val %x1, val %x2;
  tell '{%x1}*{%x2} = {%x1*%x2}';
end;
// --------------
mul 3 5;  //3*5 = 15
```

The TELL inside the procedure uses string interpolation, calculating the inside of the `{}`-curlies, and inserting the result into the resulting string.

A procedure is fundamentally the same as a user-defined function without return value. But procedures do not use parentheses or commas when called. This may pose challenges if you are inputting negative values in a procedure, like `mul 3 -5;`, expecting it to print out -15 as the result. In that case, Gekko will complain that it cannot find the procedure `mul` with only 1 argument, because it interprets `mul 3 -5;` as being the same as `mul -2;` , subtracting the two values. The solution to this could be to either use a function without return value instead (calling it with `mul(3, -5);`), enclosing the negative number in parentheses: `mul 3 (-5);`., or subtract it from 0: `mul 3 0-5;`

Another thing to note is that user-defined functions and procedures support a special variable type: *name*. On the inside of a function or procedure, a name type variable behaves just as a string, so the only difference between name and string is that, when called from the outside, a name argument does not need quotes, whereas a string argument does. For example:

```
procedure ratio name %n1, name %n2;
  prt {%n1}/{%n2};
end;
// --------------
x1 = 3; x2 = 5;
ratio x1 x2;  //prints 0.60
```

If, instead, we had used string arguments, the procedure would have looked like this:

```
procedure ratio string %n1, string %n2;
  prt {%n1}/{%n2};
end;
// --------------
x1 = 3; x2 = 5;
ratio 'x1' 'x2';  //prints 0.60, calling without quotes will fail
```

So nothing is changed in the inside of the function: the only difference is that the user now has to quote the arguments, which can be tedious for functions or procedures where some of the arguments are names of variables.

You may also use series type is argument in functions and procedures, but in many cases name type works better and provides equivalent syntax for calling the function/procedure.

3.2.4.8 Options

Four types of options

Gekko operates with four types of options: global options, general options, block-options and local options. **Global options** are read when Gekko launches and cannot be altered later on (they have the form `option global ...`). **General options** are stated with the OPTION command and apply until changed. **Block options** are stated with the BLOCK command and apply for a block of Gekko statements. **Local options** are stated inside the `<>` local option field and apply to that single statement only. See more here.

Global options

Global options have the form `option global ...` and can only be set programatically in the `gekko.ini` file next to the `gekko.exe` file (cf. INI). These options are primarily for advanced users.

General options

If you need to use an option for the whole (remainder of the) Gekko program, you just use a general option. This may be an option like setting the frequency.

```
option freq = a;
x1 = 1;
option freq = q;
x1 = 2;
option freq = a;
x2 = 3;
```

This will produce the series `x1!a` = 1, `x1!q` = 2 and `x2!a` = 3. (Instead of changing the frequency you could in this case have used `x1!q = 2;`, but the example is for illustrative purposes).

Block options

Switching general options back and forth can be error-prone and tedious, so therefore Gekko offers to use a BLOCK statement:

```
option freq = a;
x1 = 1;
block freq = q; //or: block option freq = q; --> both variants are
legal syntax
  x1 = 2;
end;
x2 = 3;
```

When BLOCK is encountered, it works like setting `option freq = q;`, but when the END is encountered, `option freq = ...` is restored to what it was before the BLOCK statement. BLOCK is particularly practical when the option applies to several Gekko statements.

Local options

Local options are the last type, applying to exactly one Gekko statement. The option is put inside the `<>` local option field, and many statements have taylor-made local options to control the behavior. Examples:

```
read <csv> data1;            //<csv> means csv file type
read <ref csv> data1;        //<ref> indicates reference databank
prt <d dec=0> x;             //<p dec=0> means absolute time-change
```

```
with no decimals
plot <linecolor='blue'> x; //blue line
plot <2020 2025> x;         //use this period
```

In addition to such taylor-made local options, you may also use general options and put them into the `<>` field. Example:

```
option freq = a;
x1 = 1;
x1 <option freq = q> = 2;
x2 = 3;
```

Here, the general option `option freq = q` is used, but it applies only to that particular statement. If you need to mix taylor-made local options with general options, put the general options *last*, after a semicolon (`;`). For instance:

```
option freq = q;
time 2020q1 2021q4;
y = 1, 2, m(), 4, 11, 12, 13, 14;
prt <2020q1 2021q2 p; option print freq = simple; option series
data missing = zero> y;
```

The green `2020q1 2021q2 p` part are normal taylor-made print options: setting the time period and choosing operator `p` (percentage growth). The blue `; option print freq = simple; option series data missing = zero` part contains general options from the OPTION statement.

Sometimes, a general option already exists as a taylor-made local option for the command. For instance, continuing from the previous example, these two statements are equivalent:

```
prt <2020q1 2021q2 collapse = total> y;
prt <2020q1 2021q2; option print collapse = total> y;
```

When an equivalent taylor-made option exists, it is recommended to just use that option.

Functions

Local options do not work for functions (yet). Syntax could be something like this:

```
y = f(<2020q1 2021q2; option series data missing = zero>, x1 + x2);
```

3.2.5 Data reporting

In order to view, inspect and error-check data, there are different kinds of reporting facilities in Gekko. In the following sub-sections, we will look at simple printing, plotting/graphs, "printing" to Excel, and producing tables.

- Printing
- Plotting
- Excel printing
- Tables

You may also use other programs for the reporting of Gekko data. To see how to produce R or Python graphs from within Gekko, see the examples/interfaces here or here. In the Excel printing sub-section, there is an example regarding how to produce an Excel graph from Gekko data.

3.2.5.1 Printing

To see the contents of Gekko-timeseries (and other variables), the PRT statement is often used.

> Note: PRT can print mixed frequencies in the same print.

The statement prints output on the upper part of the Gekko main window, for instance:

```
reset;
time 2010 2014;
x1 = 15, 12, 14, 16, 12;
x2 = 13, 14, 16, 15, 16;
prt x1, x2;  //local period: prt <2011 2013> x1, x2;
```

Result:

```
                   x1           %            x2           %
  2010        15.0000           M       13.0000           M
  2011        12.0000      -20.00       14.0000        7.69
  2012        14.0000       16.67       16.0000       14.29
  2013        16.0000       14.29       15.0000       -6.25
  2014        12.0000      -25.00       16.0000        6.67
```

PRT prints the levels (4 decimals), and in addition, percentage growth rate (2 decimals). The number of decimals etc. can be controlled via options (see `option print ...` under options). (Note that after a print has been shown in Gekko, you may click the "Copy" button in the user interface and paste the printed "cells" into for instance Excel).

The PRT statement can also be used to print mathematical expressions. The following example prints the ratio between the two series x1 and x2:

```
//continued
prt <n> x1/x2;
```

Result:

```
                     x1/x2
  2010             1.1538
  2011             0.8571
  2012             0.8750
  2013             1.0667
  2014             0.7500
```

Here, percentage growth is suppressed with option <n> (for levels). There are a number of operators like <n> available, which can be seen on the PRT page, for instance <p> for percentage growth, etc.

Ref databank comparison

PRT can also be used to compare values in the first-position databank with values in the Ref (reference) databank, for instance like this:

```
reset;
time 2010 2014;
x1 = 15, 12, 14, 16, 12;
clone;  //makes Ref an exact copy
x1 = 13, 14, 16, 15, 16;  //changes x1
prt <n r m q> x1;
```

Result:

```
                        x1            <r>            <m>        <q>
  2010            13.0000        15.0000        -2.0000     -13.33
  2011            14.0000        12.0000         2.0000      16.67
  2012            16.0000        14.0000         2.0000      14.29
  2013            15.0000        16.0000        -1.0000      -6.25
  2014            16.0000        12.0000         4.0000      33.33
```

This PRT statement prints levels in first-position databank (<n>), levels in Ref databank (<r>), absolute difference (<m>), and relative difference (<q>). The n, r, m and q letters are so-called print operators (there is a separate <i> option to print as index = 100).

Lists

The PRT statement has a lot of functionality ready regarding lists of variables, or variable names composed of lists of "codes". A few examples:

```
reset;
time 2010 2012;
xa = 1; xb = 2; xc = 3;
#m = a, b, c;   //"codes"
prt <n> x{#m}, sum(#m, x{#m});
```

Result:

```
                                                                 s
                   xa             xb             xc  um(#m, x{#m})
  2010         1.0000         2.0000         3.0000         6.0000
  2011         1.0000         2.0000         3.0000         6.0000
  2012         1.0000         2.0000         3.0000         6.0000
```

Array-series

Instead of composing names like `xa`, `xb`, `xc`, such a naming pattern can also be implemented via array-series., which were mentioned earlier on in the user guide. You may try the following nearly identical example:

```
reset;
time 2010 2012;
x = series(1);  //define 1-dimensional array-series
x[a] = 1; x[b] = 2; x[c] = 3;
#m = a, b, c;
prt <n> x[#m], sum(#m, x[#m]);
prt <n> x;
disp x;  //shows meta-information on the array-series
```

Result:

```
                                                                 s
                 x[a]           x[b]           x[c]  um(#m, x[#m])
  2010         1.0000         2.0000         3.0000         6.0000
  2011         1.0000         2.0000         3.0000         6.0000
  2012         1.0000         2.0000         3.0000         6.0000

                 x[a]           x[b]           x[c]
  2010         1.0000         2.0000         3.0000
  2011         1.0000         2.0000         3.0000
  2012         1.0000         2.0000         3.0000

 ========================================================================
==================
 SERIES Work: x
 Annual series has 3 elements in 1 dimensions (period 2010 - 2012)
 Dimension 1 (3 elements): a, b, c
 First/last elements (alphabetically): x[a] ... x[c]
 ========================================================================
==================
```

As it is seen from the last PRT statement, with an array-series, you can use the convenient `prt x;` to print all the sub-series inside `x`, and DISP will print general info on the dimensionality of `x` etc. In general, array-series can be handy for variables with well-defined dimensions like sectors, countries, age groups, etc.

Wildcards

You can use wildcards to print out timeseries. In that case, for PRT you have to use the notation `{'...'}`, for instance like this:

```
reset;
time 2010 2012;
x1a = 1; x2a = 2; x1b = 3; x2b = 4;
prt <n> {'x*a'};
prt <n> {'x1*'};
```

Result:

```

                  x1a          x2a
  2010         1.0000       2.0000
  2011         1.0000       2.0000
  2012         1.0000       2.0000

                  x1a          x1b
  2010         1.0000       3.0000
  2011         1.0000       3.0000
  2012         1.0000       3.0000
```

Without the `{'...'}` notation, `prt <n> x*a;` will fail with an error, complaining that the variables `x` and `b` can not be found (it expects to multiply them). Statements like INDEX, COPY and others do not need that notation, because they do not accept mathematical expressions, and therefore the following:

```
//continued
index x*a;
```

Produces this:

```
 x1a, x2a

 Found 2 matching items
```

The PRT statement can also print other variable type like scalars or lists. Note that MEM is handy for printing out all scalar variables from all databanks.

DISP

Another way of printing timeseries is using the DISP statement. This statement does not accept mathematical expressions, and only prints one timeseries at a time.

```
reset;
time 2010 2014;
x1 = 15, 12, 14, 16, 12;
disp x1;
```

Result:

```
 ========================================================================
==================
 SERIES Work: x1
 Annual data from 2010 to 2014 (updated: 05-02-2021)
 Source: 2010-2014: x1 = 15, 12, 14, 16, 12
 ------------------------------------------------------------------------
------------------
 Period        value        %
 2010        15.0000   ******
 2011        12.0000   -20.00
 2012        14.0000    16.67
 ------------------------------------------------------------------------
------------------
 2 periods hidden (show)
 ========================================================================
==================
```

The DISP statement show more information on the timeseries, for instance frequency, data period, last updated time, and the source of the variable. It may also show other forms of meta information, like explanation, units, etc. When a model is loaded, DISP can be used to start equation browsing. Wildcards may be used, like `disp x*;`.

3.2.5.2 Plotting

Gekko does not have its own in-built graphics engine, but you can use the PLOT statement to plot via gnuplot as engine, or alternatively transfer the variables to for instance Excel, R, Python, or others, for plotting. In this section, we will focus on the PLOT statement, which is quite central when working with timeseries.

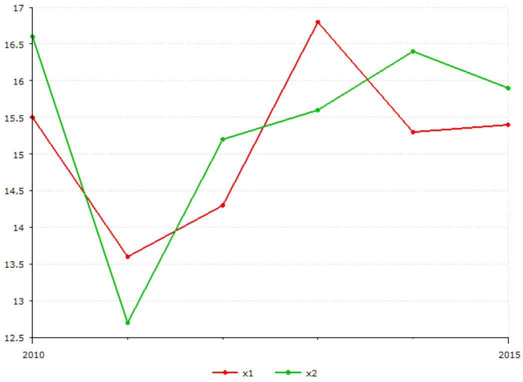
> To see what PLOT is capable of regarding styling etc., see the Gekko demo plots.
> Note: PLOT can plot mixed frequencies in the same plot.

First of all, it pays off to understand that the statements PRT, PLOT and SHEET are close cousins. This means that any expression, with any operator, used in one of the statements, can be used in the same way in one of the other statements. For instance, you may take a PRT statement and just replace PRT with PLOT, and the timeseries will be plotted instead.

Therefore, the examples from the preceding section on printing can be used for plotting, too.

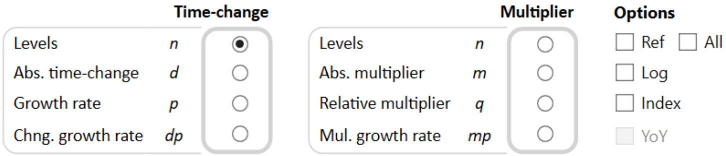
```
reset;
time 2010 2014;
x1 = 15.5, 13.6, 14.3, 16.8, 15.3, 15.4;
x2 = 16.6, 12.7, 15.2, 15.6, 16.4, 15.9;
plot x1, x2;  //local period: plot <2011 2013> x1, x2;
```

Result:



There are a lot of options to control and style plots, see the possibilities under the PLOT statement.

The PLOT window has some clickable controls, similar to the DECOMP controls:



Time-change

Levels *n* ●
Abs. time-change *d* ○
Growth rate *p* ○
Chng. growth rate *dp* ○

Multiplier

Levels *n* ○
Abs. multiplier *m* ○
Relative multiplier *q* ○
Mul. growth rate *mp* ○

Options

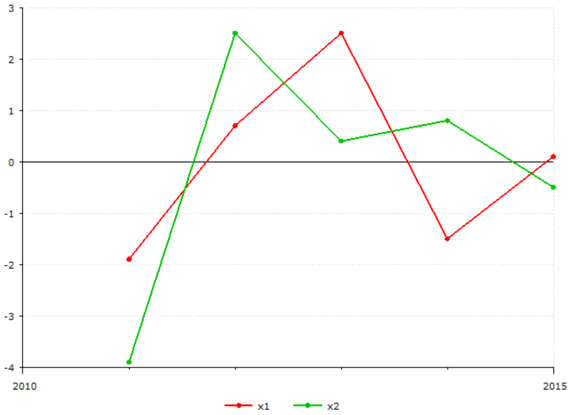
□ Ref □ All
□ Log
□ Index
□ YoY

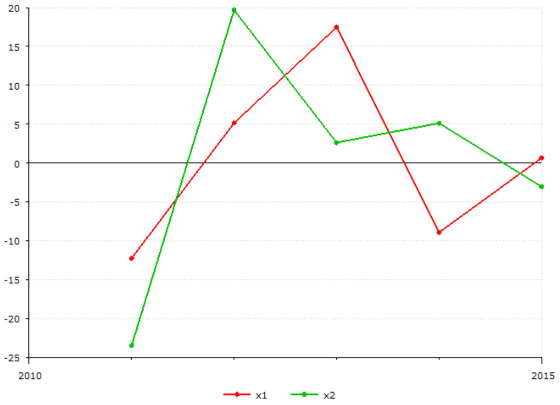
You may click on "Levels" to get levels (corresponding to `<n>` operator), or on "Growth rate" to get period for period relative change (operator or `<p>`). If you click "Relative multiplier", Gekko will show relative differences relative to the `Ref` databank (operator `<q>`), and so on. To show values from the `Ref` databank, use the "Ref" checkbox (or that "All" checkbox), and logarithms, indexing, and year-over-year transformations are also possible.

You may try this:

```
//continued
plot <d> x1, x2;  //absolute time difference
plot <p> x1, x2;  //percentage growth
```

Operator <d>, absolute change **Operator <p>, growth rate (%)**





If you use the 'Copy link' button, the plot is copied as a .svg file, for pasting in, e.g., Microsoft Excel of similar. Alternatively, you may save these .svg files to disk with the other buttons. Note that PLOT supports other output file formats, too, like for instance .png, .pdf, .emf, etc. The .svg format is practical for use with html pages. This is how the demo plots are made.

Like for PRT, you may plot any mathematical expressions like `plot x1/x2;` etc.

See the page on printing regarding plotting with databank comparisons, lists, array-series, and wildcards -- the syntax is the same as for PRT.

You can use PLOT silently to produce graph files, for instance:

```
//continued
plot <d> x1, x2 file = plot.svg;
```

This will produce the file plot.svg in the working folder, without opening up a PLOT window.

The PLOT statement may produce a large number of plot windows. These can be automatically closed via the Gekko menu in the user interface: Window --> Close --> Close all PLOT. Alternatively, the "Close all PLOT and DECOMP windows" button can be used, and there is also a CUT statement that does the same.

3.2.5.3 Excel printing

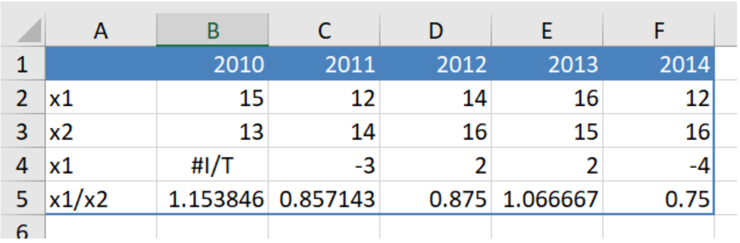
You may use the SHEET statement to "print" directly to an Microsoft Excel worksheet. (For non-Excel users, you may try CLIP instead of SHEET (to copy-paste via the clipboard)).

As mentioned in the section on plotting, the statements PRT, PLOT and SHEET (and CLIP) are close cousins. This means that any expression, with any operator, used in one of the statements, can be used in the same way in one of the other statements. For instance, you may take a PRT statement and just replace PRT with SHEET, and the timeseries will be transferred to Excel instead.

A simple example could be the following:

```
reset;
time 2010 2014;
x1 = 15, 12, 14, 16, 12;
x2 = 13, 14, 16, 15, 16;
sheet x1, x2, x1<d>, x1/x2;  //local period: use sheet <2011 2013>
... ;
```

The following Excel worksheet is shown:

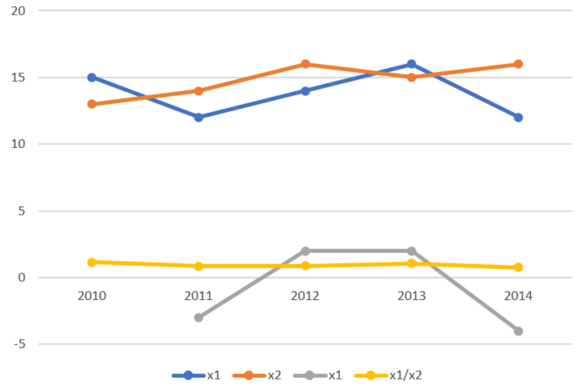


	A	B	C	D	E	F
1		2010	2011	2012	2013	2014
2	x1	15	12	14	16	12
3	x2	13	14	16	15	16
4	x1	#I/T	-3	2	2	-4
5	x1/x2	1.153846	0.857143	0.875	1.066667	0.75
6						

As with PRT or PLOT, you can use operators and mathematical expressions. The third argument `x1<d>` shows absolute time-difference in `x1`, because of the operator `<d>`.

Transforming the worksheet into an Excel graph is quite easy. Just click on the A1 cell, and then choose "Insert" and choose the chart type that you prefer (here: line with markers):

Excel graph



The SHEET statement needs Excel installed to work (else use CLIP, mentioned below). Per default, SHEET shows timeseries row-wise (use `sheet<cols>` to "print" in columns). In this worksheet, a missing value is shown as `#I/T` (because row 4 shows absolute time-change in `x1`), So operators like absolute or percentage time-change are easy to use with Excel, and expressions, too. This may in some cases be more convenient than using Excel formulas to do the same.

All in all, the SHEET statement is practical for showing things in Excel, rather than in the Gekko output window (with PRT). There are, however, other ways of interacting with Excel:

- Gekcel. This is an add-in for Excel that makes it possible to issue Gekko statements from within Excel (without starting up the Gekko graphical user interface).
- CLIP. This statement produces a semicolon-separated string that can be pasted to Excel (or other spreadsheets). Compared to SHEET, if using CLIP to paste into Excel, some precision and formatting is lost.
- IMPORT<xlsx> and EXPORT<xlsx>. These statements operate on Excel sheets formatted with variables as rows and dates as columns. Similar to csv import/export, just for .xlsx.
- SHEET<import>. This is a special kind of SHEET statement, where data is imported from Excel. In contrast to IMPORT<xlsx>, SHEET<import> is more flexible and can operate more freely on individual cells. SHEET<import> can also import a Gekko matrix from Excel cells.

3.2.5.4 Tables

The TABLE statement is used to call tables (.gtb files) designed in a special XML format. Tables are intended for the tabelling of timeseries, with quite a lot of special functionality regarding this. Tables can be produced in either text or html format, and can be shown in the "Menu" tab of the Gekko graphical user interface. (Note that after a table has been shown in Gekko, you may use the 'Copy' button to copy the cells to the clipboard, for successive pasting into e.g. a spreadsheet like Excel).

Simple example

We will try to construct a simple table with a few timeseries. First, we construct the table file (.gtb). You can use Notepad to do this, try using the `edit demo.gtb;`, after which an empty file shows up (click 'yes' to create a new file). Next, put the following into demo.gtb and save the file.

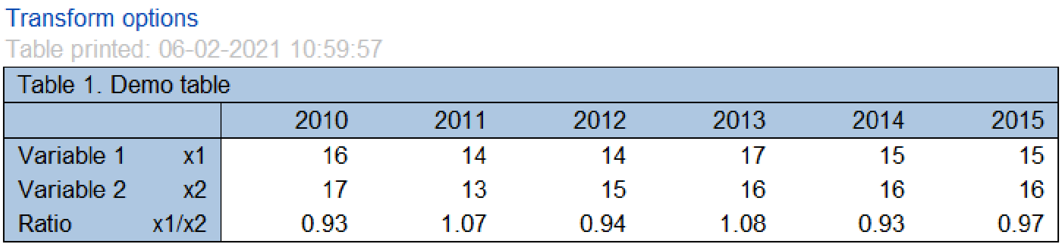
Gekko table file: demo.gtb

```
<?xml version="1.0" encoding="Windows-1252"?>
<gekkotable>
  <tableversion>1.0</tableversion>
  <table varformat="f9.0">
    <cols>
      <colborder/>
      <col txtalign="left"/>
      <col txtalign="right"/>
      <colborder/>
      <col type="expand" txtalign="center"/>
      <colborder/>
    </cols>
    <rows>
      <rowborder/>
      <row>
        <txt colspan="3">Table 1. Demo table</txt>
      </row>
      <rowborder/>
      <row>
        <txt/>
        <txt/>
        <date/>
      </row>
      <rowborder/>
      <row>
        <txt>Variable 1</txt>
        <txt>$</txt>
        <var>x1</var>
      </row>
      <row>
        <txt>Variable 2</txt>
        <txt>$</txt>
        <var>x2</var>
      </row>
      <row varformat="f9.2">
        <txt>Ratio</txt>
        <txt>$</txt>
        <var>x1/x2</var>
      </row>
      <rowborder/>
    </rows>
  </table>
</gekkotable>
```

Now, we can try showing the table:

```
reset;
time 2010 2017;
x1 = 15.5, 13.6, 14.3, 16.8, 15.3, 15.4;
x2 = 16.6, 12.7, 15.2, 15.6, 16.4, 15.9;
table demo;  //adds .gtb on its own
```

Result (shown in the "Menu" tab):



Transform options

Table printed: 06-02-2021 10:59:57

Table 1. Demo table							
		2010	2011	2012	2013	2014	2015
Variable 1	x1	16	14	14	17	15	15
Variable 2	x2	17	13	15	16	16	16
Ratio	x1/x2	0.93	1.07	0.94	1.08	0.93	0.97

Here, you may click the link 'Transform options' to transform the data (for instance the `p` button for percentage growth).

First of all, it is seen that the decimals are removed for `x1` and `x2`, because of the tag `varformat="f9.0"`. Here, `f9.0` means floating point, 9 characters wide, 0 decimals. This `varformat` is set for the whole table at the top of the .gtb file, but is "overruled" in the `x1/x2` row, where it is set to `f9.2` (two decimals).

The table is designed by first stating `<cols>`, and then `<rows>`. Regarding the columns, we do not need to state a column for each year/period, but just a column that is "expandable".

- **Col 1**: `<col txtalign="left">`. The first column is for showing the labels (for instance "Variable 1").
- **Col 2**: `<col txtalign="right"/>`. The next column is for shown the variables/formulas (like "x1").
- **Col 3**: `<col type="expand">`. The third column is an expandable column for the six years 2010-2015.

There are also some `<colborder>` tags to insert vertical borders/lines. So technically, you should think of the table as having three columns, where the last column is "expandable".

Next, we define the `<rows>`.

- **Row 1**: The first row looks like this: `<txt colspan="3">Table 1. Demo table</txt>`. The `colspan="3"` tag means that the cell spans three columns, in effect merging these cells into one "super cell". Here, it should be noted that Gekko visualizes the table as having three columns in all (where the last column is expandable), so `colspan="3"` effectively also merges with all the date columns (2010-2015), so that the first row spans all columns of the table.

- **Row 2**: The next row has these tags: `<txt/><txt/><date/>`, which means empty text for the first two columns, and then `<date>` tag to ask Gekko to show the dates (in this case: years) corresponding to the global data period 2010-2015.
- **Row 3**: There are these tags: `<txt>Variable 1</txt> <txt>$</txt> <var>x1</var>`, so in the first column, the text `Variable 1` is inserted, whereas the special `$` in column two means that it should take the text from the next column, that is, insert the text `x1`. This is just to avoid repetitive typing. The last column uses a `<var>` (variable) tag to state the variable or expression for the expandable column.
- **Row 4**: Just like row 3.
- **Row 5**: Also just like row 3, but overruling the formatting, to show two decimals (`f9.2`).

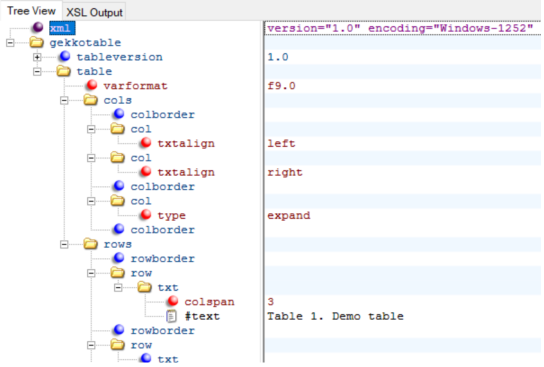
In addition, there are some `<rowborder>` tags to insert horizontal borders/lines.

Note that you may use strings in `{}`-curlies inside the xml elements, for instance instance `<txt>Table regarding the year {%year}</txt>`.

More details

Tables can also be called from menus (see MENU). Menus are used to store and organize a collection of tables in tree-like structure for better overview (if there are many tables).

Instead of using Notepad or a similar text editor, you may alternatively use the in-built XML Notepad editor to edit the .gtb file, try `xedit demo.gtb;`, and see XEDIT. Use 'View' --> 'Expand All' to unfold all XML nodes for better overview, and use Ctrl+D to duplicate an XML node (including its children nodes).



```
xml                          version="1.0" encoding="Windows-1252"
gekkotable
  tableversion               1.0
  table
    varformat                f9.0
    cols
      colborder
      col
        txtalign             left
      col
        txtalign             right
      colborder
      col
        type                 expand
      colborder
    rows
      rowborder
      row
        txt
          colspan            3
          #text              Table 1. Demo table
      rowborder
      row
        txt
```

For more advanced users, you may ask Gekko to output the raw HTML code corresponding to the table, and together with PIPE<html>, and graphs stored in the .svg format (to produce these, use the syntax `PLOT x1, x2 file = plot.svg;`), you may construct html pages mixing text/explanations, tables, and plots. The Gekko demo plots are actually an example of an auto-constructed html page (this page does not, however, contain html tables). Html pages can be converted to pdf for easier distribution (for instance via mail).

3.3 Modeling User Guide

This guide showcases the modeling capabilities of Gekko. The guide is centered around a concrete practical simulation example (for which data can be downloaded).

The modeling user guide contains the following sections:

- Download etc.
- Modeling basics
- Historical simulation
- Multiplier analysis (shocks)
- Add-factors etc.
- Goal-search etc.
- Forward-looking

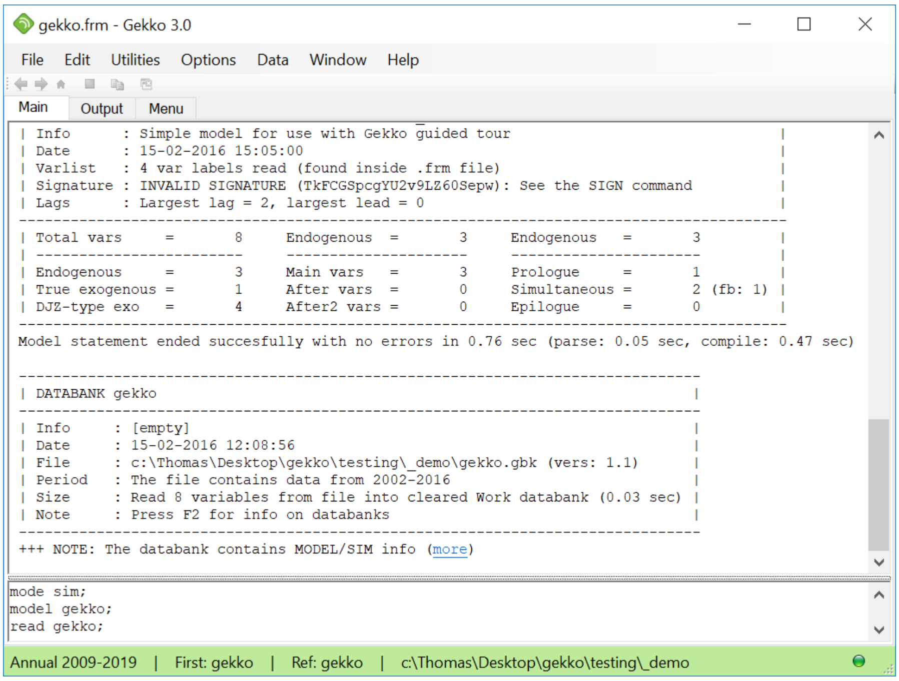
3.3.1 Download etc.

This part of the user guide will focus on the simulation capabilities. We will be using sim-mode (in contrast to data-mode) in this part of the Gekko User Guide.

> **Sim-mode**. This mode can be used for modeling, and entails two main differences compared to data-mode (which is default). First, in a statement like `y = 2 * x;`, in sim-mode Gekko will never look for `x` in a databank that was opened with the OPEN (or the identical READ ... AS) statement. And secondly, in a statement like `y = 2 * x;`, Gekko will fail with an error if the timeseries `y` does not exist already (else you must use CREATE to create `y` first). These restrictions alleviate some common errors when modeling. But note that it is entirely possible to do modeling in data- or mixed mode, and some users even prefer that. It is basically a matter of taste.

To download and install Gekko, see the general installation section.

You may consult the page on interface, function keys, etc. to read more about how the interface works. But for now, just note that you type statements in the lower part of the window, and results are shown in the upper part. You issue statements with `[Enter]`, and if you omit the ending `;` in a statement, the user interface will add it on its own.



```
| Info      : Simple model for use with Gekko guided tour                               |
| Date      : 15-02-2016 15:05:00                                                       |
| Varlist   : 4 var labels read (found inside .frm file)                                |
| Signature : INVALID SIGNATURE (TkFCGSpcgYU2v9LZ60Sepw): See the SIGN command          |
| Lags      : Largest lag = 2, largest lead = 0                                         |
-----------------------------------------------------------------------------------------
| Total vars     =       8     Endogenous  =       3     Endogenous   =       3         |
| ------------------------     ---------------------     ----------------------         |
| Endogenous     =       3     Main vars   =       3     Prologue     =       1         |
| True exogenous =       1     After vars  =       0     Simultaneous =       2 (fb: 1) |
| DJZ-type exo   =       4     After2 vars =       0     Epilogue     =       0         |
-----------------------------------------------------------------------------------------
Model statement ended succesfully with no errors in 0.76 sec (parse: 0.05 sec, compile: 0.47 sec)

-------------------------------------------------------------------------------
| DATABANK gekko                                                              |
-------------------------------------------------------------------------------
| Info     : [empty]                                                          |
| Date     : 15-02-2016 12:08:56                                              |
| File     : c:\Thomas\Desktop\gekko\testing\_demo\gekko.gbk (vers: 1.1)      |
| Period   : The file contains data from 2002-2016                            |
| Size     : Read 8 variables from file into cleared Work databank (0.03 sec) |
| Note     : Press F2 for info on databanks                                   |
-------------------------------------------------------------------------------
+++ NOTE: The databank contains MODEL/SIM info (more)
```

```
mode sim;
model gekko;
read gekko;
```

Annual 2009-2019 | First: gekko | Ref: gekko | c:\Thomas\Desktop\gekko\testing_demo

When Gekko starts up, it will start up in a 'working folder'. To run the modeling guide, you will need some external files that can be downloaded from the following link (these files should be downloaded to the newly created working folder. Right-click the following zip-file, and choose "Save as..." or "Save link as..."):

- Example model and data files, demo.zip

The zip file contains the files `gekko.frm` (the model) and `gekko.gbk` (the databank). You can typically just double-click the zip-file to open it, and then copy-paste the two files into your working folder.

3.3.2 Modeling basics

Before we start out the modeling user guide, an explanation of some basic concepts that are relevant when modeling in Gekko. You may skip this part if you prefer and move directly to the next section with a more practical simulation example. (On the contrary, to read even more about databanks, timeseries, lists, and other things, you may consult the Data Management User Guide which has much more info and examples regarding this).

Databanks

Databanks are containers of variables, for instance timeseries. Gekko always starts out with two empty databanks (in memory): `Work` and `Ref` (reference). The `Work` databank is where data is normally changed, unless otherwise stated. For instance, simulations are always performed on `Work` databank data. The `Ref` databank can be thought of as a background databank, being particularly handy when comparing two scenarios. Try pressing the `F2` key to see the open databanks (note: if the `Ref` databank is empty, it does not show up in the `F2` list). Since a READ statement wipes out the contents of the `Work` and `Ref` databanks, it may be practical to put settings etc. (for instance paths, time periods, etc.) into the so-called `Global` databank. Databanks can contain many variables, so beware that the keys `[Tab]` or `[Ctrl+Space]` offer autocompletion on timeseries names (cf. here).

Timeseries

Timeseries reside in a databanks. Timeseries may have frequency annual, quarterly, monthly, weekly, daily or undated. If data has been read for timeseries `y` regarding the period 2015-2020, printing out `y` for the period 2021 will show a missing value ('M').

When doing modeling (model simulations) with Gekko, timeseries are often called "variables". Model files (.frm) files contain equations that describe relationships between these variables (timeseries), and therefore timeseries are fundamental when doing modeling.

Note that statements involving timeseries can have a local time period indicated, for instance printing with `prt <2020 2030> var1, var2;`. Global time can be altered with the TIME statement.

Ref databank and comparisons

When a databank is read (the READ statement), the Work databank is cleared, and all the variables from the file are put into the `Work` databank (it is possible to merge databanks if this behavior is preferred). After this, the `Ref` databank is also cleared, and all variables are copied from `Work` to `Ref`. So after reading a databank file with READ, the `Work` and `Ref` databanks are always identical (there are other ways of opening databanks, but for now we focus on READ).

The `Ref` databank is typically used for multiplier analysis (i.e., experiments). Say you read a databank and then perform some experiment. This experiment will only alter variables in the `Work` databank, so after the experiment is finished, you can compare the variables (timeseries) in the `Work` and `Ref` databanks (Gekko has a lot of statements to do such comparisons, for instance MULPRT, DECOMP etc.).

The so-called operators are used for comparisons. For instance, `m` means absolute multiplier, whereas `d` means absolute differences (or `q` and `p` in their relative versions). You may consult the PRT (print) statement regarding this, but suffices to say that you may write for instance `prt <m> var1;`, `plot <d> var1;`, `sheet <q> var1;`, etc. There are also some more mnemotechnic ('long') operators availiable, for instance `abs` or `pch` (try for instance `prt<abs>` or `prt<pch>`).

If, at some point, you wish to make sure that the `Work` and `Ref` databanks are identical (for instance after a simulation), you can use the CLONE statement. This statement clears the `Ref` databank, and copies the `Work` databank into it (in memory). CLONE is typically used just after simulating (SIM) a baseline/reference scenario.

Cleanup and restart

There is a cleanup-statement: RESTART. This statement clears the `Work` and `Ref` databanks, in addition to clearing models, lists, options and other things. This provides a clean state of Gekko, as if it had just been closed and reopened. If there is a file with the name `gekko.ini` present in the working folder, the Gekko-statements in this file will be run by RESTART, so `gekko.ini` can be used to contain options and other statements (for instance a MODEL statement) that the user wishes to "survive" a RESTART statement.). You may also use CLS ("clear screen") to clear the output window.

In general, when doing simulations (in so-called sim-mode) and want to define a new timeseries variable not already present in the model or databank, you will have to CREATE it first (unless the timeseries starts with the letters `xx`). However, it should be noted that when a databank is read, any model variables not present in the databank will be auto-created as timeseries (with all observations set to missing values). Because of this, it is often most convenient to put MODEL statements before READ statements. Preferably use this order in Gekko program files: first the RESTART statement, then the MODEL statement, and then the READ statement (or in the `gekko.ini` file, put the MODEL statement before the READ statement).

Lists, filenames, etc

Regarding models, it should also be noted that the list of endogenous variables in a model is simply the set of all the variables at the left-hand side of the equations. This may be changed afterwards by means of the ENDO and EXO statements. Regarding equation syntax, you may consult the latter part of the MODEL help file, if you need more information on this.

The hash sign (`#`) is used for collections (lists, maps and matrices), and the percent sign (`%`) is used for scalar variables (value, date and string). So in general you refer to these with `#x` or `%x`, but note that in name composition using string/name scalars, it is advised to use {}-curlies, for instance `fK{%type}{%sector}`, where `%type` and `%sector` could be type and sector names (stored in strings).

Generally, list items are separated by commas, e.g. `#mylist = var1, var2, var3;` (this stores the three strings `'var1'`, `'var2'` and `'var3'` in the list). This is also the case when the list of items contains expressions: `prt x/y, w/z;`. Lags and fixed dates are put inside brackets, for instance: `var1[-1]` or `var1[2020]`. You may use `x.1` or `x.2` as short-hand for `x[-1]` or `x[-2]` and so on. Square brackets are also used for wildcard-lists, so instead of a standard list (`#mylist`), you may use for instance `['fX*']` to obtain a list (of strings) of all variables in the Work databank beginning with `fX`.

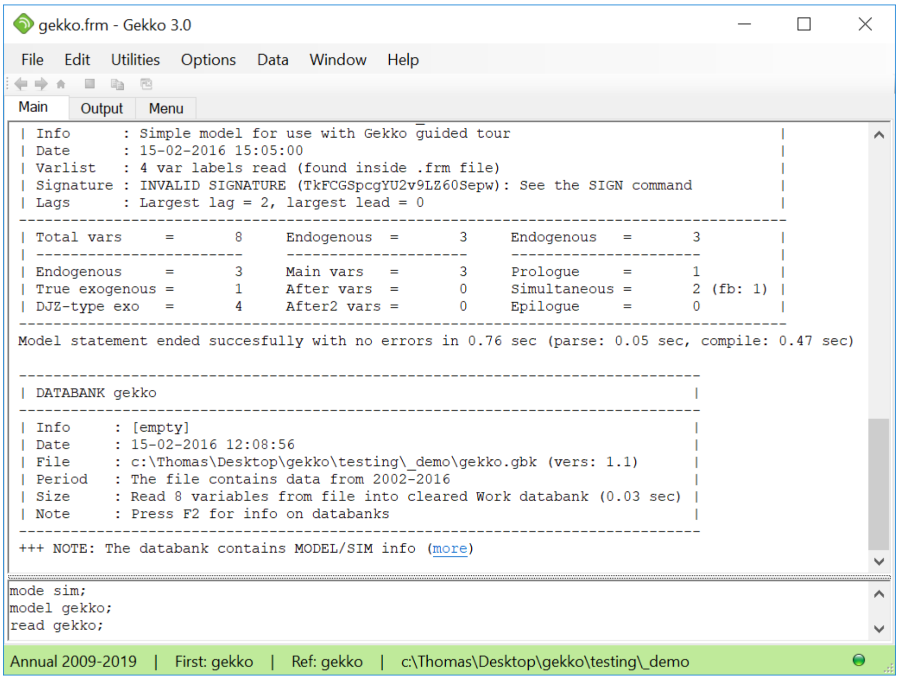
Regarding file names, you may use relative paths like `\subfolder\filename.txt`. Using relative paths makes it easier to move a system of program files (using sub-folders) to another location/computer if needed. Special user-paths can also be designated by means of the `option folder …` settings.

3.3.3 Historical simulation

This section describes how to perform a historical simulation on a (historical) databank. In order to follow the examples, you must first download the model and databank (click demo.zip, and copy the two files gekko.frm and gekko.gbk into your Gekko working folder). The example uses annual data, but other frequencies would run very similarly.

(See the bottom of this page for the full code).

Start up Gekko in the working folder.



```
| Info      : Simple model for use with Gekko guided tour                 |
| Date      : 15-02-2016 15:05:00                                         |
| Varlist   : 4 var labels read (found inside .frm file)                  |
| Signature : INVALID SIGNATURE (TkFCGSpcgYU2v9LZ60Sepw): See the SIGN command |
| Lags      : Largest lag = 2, largest lead = 0                           |
-------------------------------------------------------------------------------
| Total vars     =       8     Endogenous  =       3     Endogenous   =       3         |
| ------------------------     ---------------------     ----------------------         |
| Endogenous     =       3     Main vars   =       3     Prologue     =       1         |
| True exogenous =       1     After vars  =       0     Simultaneous =       2 (fb: 1) |
| DJZ-type exo   =       4     After2 vars =       0     Epilogue     =       0         |
-------------------------------------------------------------------------------
Model statement ended succesfully with no errors in 0.76 sec (parse: 0.05 sec, compile: 0.47 sec)

----------------------------------------------------------------------
| DATABANK gekko                                                      |
----------------------------------------------------------------------
| Info     : [empty]                                                  |
| Date     : 15-02-2016 12:08:56                                      |
| File     : c:\Thomas\Desktop\gekko\testing\_demo\gekko.gbk (vers: 1.1) |
| Period   : The file contains data from 2002-2016                    |
| Size     : Read 8 variables from file into cleared Work databank (0.03 sec) |
| Note     : Press F2 for info on databanks                           |
----------------------------------------------------------------------
+++ NOTE: The databank contains MODEL/SIM info (more)
```

```
mode sim;
model gekko;
read gekko;
```

Annual 2009-2019 | First: gekko | Ref: gekko | c:\Thomas\Desktop\gekko\testing_demo

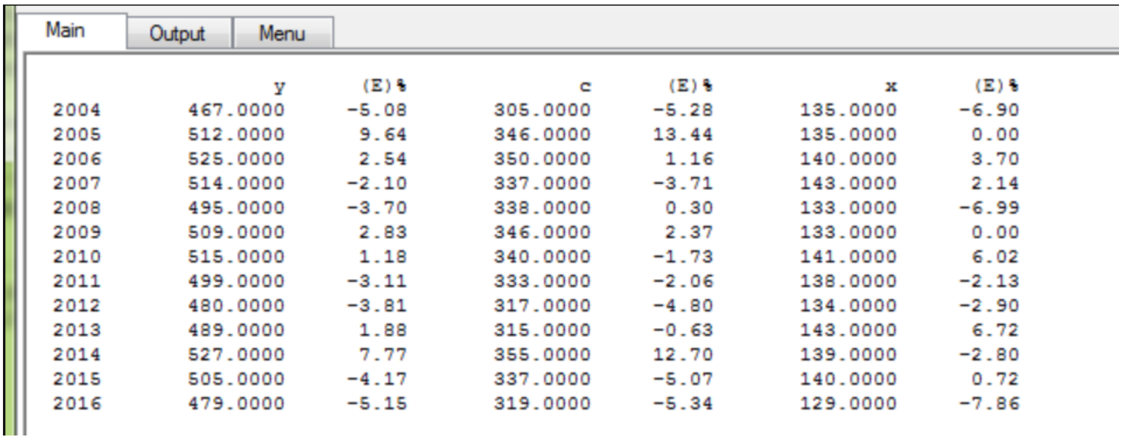
Then type this in the statement prompt (at the bottom):

```
restart;
mode sim;
time 2004 2016;
```

If you copy-paste these statements to Gekko, you may either execute them individually one by one by pressing `[Enter]`, or first mark them as a block and then press `[Enter]` to execute them at once.

The statements clear up the workspace, and set the global time period (for which we will later simulate the model). At the bottom of the Gekko window, you can see that this time period has been set ("Annual 2004 2016"). You may try to load the model, read the data variables, and print them out:

```
model gekko;
read gekko;
prt <2004 2016> y, c, x, g;
```



```
                   y     (E)%              c     (E)%              x     (E)%
 2004       467.0000    -5.08       305.0000    -5.28       135.0000    -6.90
 2005       512.0000     9.64       346.0000    13.44       135.0000     0.00
 2006       525.0000     2.54       350.0000     1.16       140.0000     3.70
 2007       514.0000    -2.10       337.0000    -3.71       143.0000     2.14
 2008       495.0000    -3.70       338.0000     0.30       133.0000    -6.99
 2009       509.0000     2.83       346.0000     2.37       133.0000     0.00
 2010       515.0000     1.18       340.0000    -1.73       141.0000     6.02
 2011       499.0000    -3.11       333.0000    -2.06       138.0000    -2.13
 2012       480.0000    -3.81       317.0000    -4.80       134.0000    -2.90
 2013       489.0000     1.88       315.0000    -0.63       143.0000     6.72
 2014       527.0000     7.77       355.0000    12.70       139.0000    -2.80
 2015       505.0000    -4.17       337.0000    -5.07       140.0000     0.72
 2016       479.0000    -5.15       319.0000    -5.34       129.0000    -7.86
```

The first statements load gekko.frm and read the gekko.gbk databank file. Note the use of the < and > brackets denoting local options. In all relevant statements, you can state a time period inside such brackets, and the given time period will only be used in that statement. The PRT statement prints both levels and annual growth for the variables. (Note the codes '(E)' for endogenous, and '(X)' for exogenous).

The data is artificial, and contains no growth. So the model can be envisioned as either depicting a stationary-state model (with dynamics), or depicting growth-corrected variables. Avoiding growth in the data is just to keep the model simple.

Lists may be used to avoid repetitive typing:

```
#vars =  y, c, x, g;
```

Try printing out the current lists:

```
list ?;
```

This shows the newly created list `#vars`. There are also some model lists. For instance, the list `#all` contains all model variables, `#endo` contains all endogenous variables (i.e., variables at the left-side of an equation), and `#exo` contains all exogenous variables (to see the contents of the `#exo` list, use `PRT #exo;`). Now you may print the variables using the list (instead of `prt y, c, x, g;`):

```
prt {#vars};
```

Note that `prt #vars;` would print out the list itself (that is, four strings). Instead of lists, wild-cards can be used, for instance:

```
prt {'*'};
```

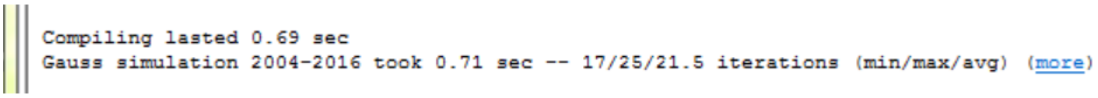
This will print out all timeseries in the Work databank. The alternative `prt ['*'];` would print out the variables as strings (names). Note that there are some model-created variables (dummies, add-factors, with missing values ('M')) — these will be described later on. Wildcards follow the standard: `*` for any match, and `?` for single character match.

We will now try to simulate the model. It should be noted, that any READ statement works like this: First the Work databank is cleared, and the databank file is read. Next the Ref databank is cleared too, and all variables in Work are copied into Ref. Hence, after any READ statement, the two databanks Work and Ref are always identical. (You may use READ<merge> to merge data, or READ<ref> to load data into the Base databank separately). To verify this, try to print a multiplier difference:

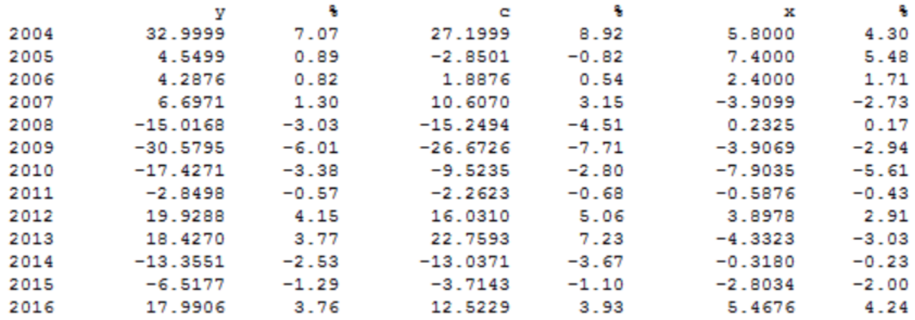
```
mulprt {#vars};
```

As you can see, there is no difference. This will change after we simulate the model:

```
sim;
mulprt {#vars};
```



```
Compiling lasted 0.69 sec
Gauss simulation 2004-2016 took 0.71 sec -- 17/25/21.5 iterations (min/max/avg) (more)
```



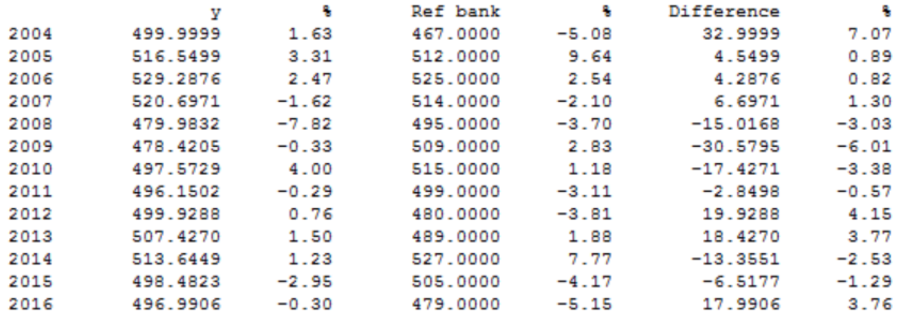
```
                 y        %            c        %            x        %
2004       32.9999     7.07      27.1999     8.92       5.8000     4.30
2005        4.5499     0.89      -2.8501    -0.82       7.4000     5.48
2006        4.2876     0.82       1.8876     0.54       2.4000     1.71
2007        6.6971     1.30      10.6070     3.15      -3.9099    -2.73
2008      -15.0168    -3.03     -15.2494    -4.51       0.2325     0.17
2009      -30.5795    -6.01     -26.6726    -7.71      -3.9069    -2.94
2010      -17.4271    -3.38      -9.5235    -2.80      -7.9035    -5.61
2011       -2.8498    -0.57      -2.2623    -0.68      -0.5876    -0.43
2012       19.9288     4.15      16.0310     5.06       3.8978     2.91
2013       18.4270     3.77      22.7593     7.23      -4.3323    -3.03
2014      -13.3551    -2.53     -13.0371    -3.67      -0.3180    -0.23
2015       -6.5177    -1.29      -3.7143    -1.10      -2.8034    -2.00
2016       17.9906     3.76      12.5229     3.93       5.4676     4.24
```

The SIM and MULPRT statements are executed over the period 2004-2016 since this is previously set as the global time period. As expected, for `g` there is no difference (it is exogenous), whereas there are differences for the other variables (the simulation is dynamic: a static simulation using historical values for lagged endogenous variables can be done with SIM<static>). To see the data more clearly, you may try:

```
prt @y, y;
```

The symbol `@` indicates that the value is taken from the Ref databank. Alternatively, try:

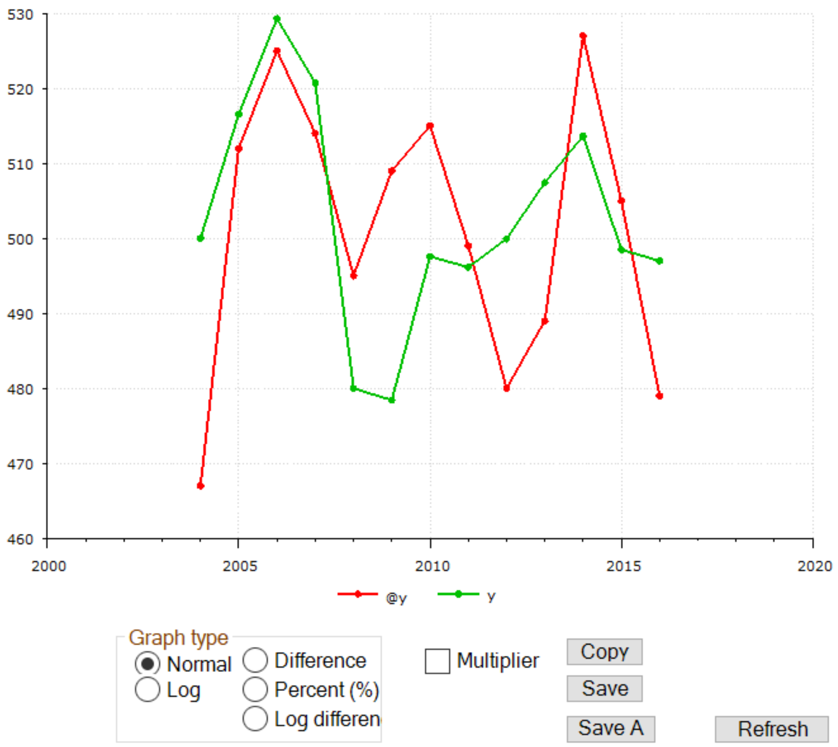
```
mulprt <v> y;
```



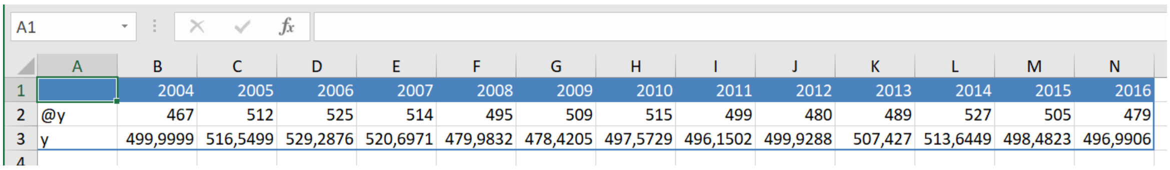
```
                    y          %       Ref bank          %     Difference          %
2004         499.9999       1.63       467.0000      -5.08        32.9999       7.07
2005         516.5499       3.31       512.0000       9.64         4.5499       0.89
2006         529.2876       2.47       525.0000       2.54         4.2876       0.82
2007         520.6971      -1.62       514.0000      -2.10         6.6971       1.30
2008         479.9832      -7.82       495.0000      -3.70       -15.0168      -3.03
2009         478.4205      -0.33       509.0000       2.83       -30.5795      -6.01
2010         497.5729       4.00       515.0000       1.18       -17.4271      -3.38
2011         496.1502      -0.29       499.0000      -3.11        -2.8498      -0.57
2012         499.9288       0.76       480.0000      -3.81        19.9288       4.15
2013         507.4270       1.50       489.0000       1.88        18.4270       3.77
2014         513.6449       1.23       527.0000       7.77       -13.3551      -2.53
2015         498.4823      -2.95       505.0000      -4.17        -6.5177      -1.29
2016         496.9906      -0.30       479.0000      -5.15        17.9906       3.76
```

which prints out both Work and Ref values, and multiplier differences (`v` means 'verbose'). Instead of printing, you may create a graph with PLOT:

```
plot @y, y;
```



In general, you may substitute PLOT for PRT, since the syntax is identical. In the window, you may for instance try to click the "Percent (%)" radio button, and you may use `[Esc]` to close a graph window quickly. Instead of PLOT, you may also try SHEET, which shows the data as an Excel table (if Excel is available on the computer, else use CLIP).



	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1		2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	2015	2016
2	@y	467	512	525	514	495	509	515	499	480	489	527	505	479
3	y	499,9999	516,5499	529,2876	520,6971	479,9832	478,4205	497,5729	496,1502	499,9288	507,427	513,6449	498,4823	496,9906
4														

In general, for statements like PRT, PLOT, SHEET and CLIP, you may use so-called "operators". For instance, `p` means annual growth rate, `m` means absolute multiplier differences, and `q` means relative multiplier differences. So this statement:

```
prt <p> {#vars};
```

will print percentage annual growth rates for the variables, and

```
plot <p> {#vars};
```

plots them. You may also use a `r` to indicate Ref databank, for instance:

```
prt <rp> {#vars};
prt <rn> {#vars};
```

This prints percentage change and levels for the Ref databank.

If you wish to store the result, you may write it to a databank (.gbk file):

```
write simple;
```

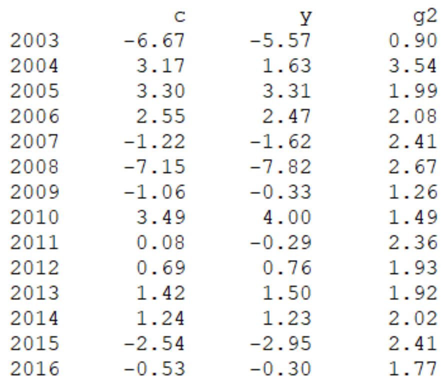
This will store the file simple.gbk in the Gekko working folder. Later on, you could read it with `read simple;`.

In sim-mode, any new variable not present in the model should be created first (unless the variable name starts with `xx`, in which case the variable will be auto-created as a convenience):

```
create g2;
g2 <2002 2016> = g;
```

The statement makes `g2` and `g` identical, for the given period. Another expample could be this:

```
pch(g2) <2003 2016> = pch(c) - pch(y) + 2;
prt <2003 2016 p> c, y, g2;
```



```
                c         y        g2
2003        -6.67     -5.57      0.90
2004         3.17      1.63      3.54
2005         3.30      3.31      1.99
2006         2.55      2.47      2.08
2007        -1.22     -1.62      2.41
2008        -7.15     -7.82      2.67
2009        -1.06     -0.33      1.26
2010         3.49      4.00      1.49
2011         0.08     -0.29      2.36
2012         0.69      0.76      1.93
2013         1.42      1.50      1.92
2014         1.24      1.23      2.02
2015        -2.54     -2.95      2.41
2016        -0.53     -0.30      1.77
```

The first statement (a SERIES statement) is set to start in 2003 — otherwise the result will be missing values, since the pch() function uses lagged values. Note also in the PRT statement the `p` inside the option brackets for indicating annual percentage growth rate. As it is seen, the annual growth rate of the new `g2` variable is set to the growth rate of `c` minus the growth rate of `y` plus 2 (percentage points).

The PRT/PLOT/SHEET/CLIP statements accept any expression, so the historical consumption rate may be printed like this:

```
prt <r> c/y;
```

```
                 c/y       (rp)
2004          0.6531      -0.21
2005          0.6758       3.47
2006          0.6667      -1.35
2007          0.6556      -1.65
2008          0.6828       4.15
2009          0.6798      -0.45
2010          0.6602      -2.88
2011          0.6673       1.08
2012          0.6604      -1.04
2013          0.6442      -2.46
2014          0.6736       4.57
2015          0.6673      -0.93
2016          0.6660      -0.20
```

Where the operator `r` indicates reference (historical) values: `prt @c/@y;` or `prt ref:c/ref:y;` could also be used. Another way of printing variables is the DISP statement. This statement does not accept mathematical expressions, but prints the equation (if it is an endogenous variable), variable explanations (labels), etc. Try:

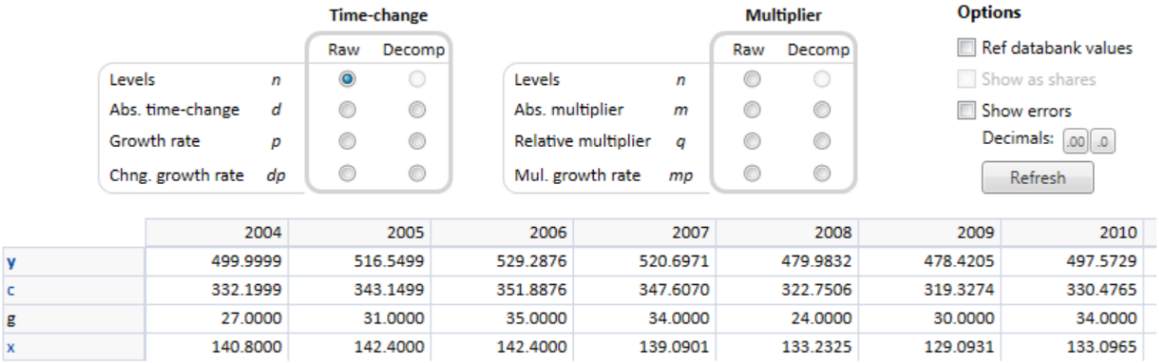
```
disp y;
```

```
=========================================================================
SERIES Work: y
Endogenous, Annual data from 2002 to 2016 (updated: 28-08-2019)
Gross domestic product (GDP)
(fixed prices/volumes)
Source: Gekko artificial data
Identity: y = c + g + x
Series source: 2004-2016: SIM gekko.frm (hash Ga5pUuM73FiarlDzuZLHrw)
Influences: c, x
-------------------------------------------------------------------------
FRML _I                y  =  c + g + x;
-------------------------------------------------------------------------
Show detailed equation
-------------------------------------------------------------------------
Period          value        %
2004        499.9999     1.63
2005        516.5499     3.31
2006        529.2876     2.47
-------------------------------------------------------------------------
10 periods hidden (show)
=========================================================================
```

You can see the GDP equation, and the label associated with the `y` variable (this label is taken from the .frm file, after the `VARLIST$` tag). You may follow the links by clicking the underlined variables, like in a web browser. Use the arrow buttons at the top-left of the Gekko window to browse backwards and forwards.

Another way of showing information regarding endogenous variables is using the `DECOMP` statement. Type:

```
decomp y;
```



Time-change

		Raw	Decomp
Levels	*n*	●	○
Abs. time-change	*d*	○	○
Growth rate	*p*	○	○
Chng. growth rate	*dp*	○	○

Multiplier

		Raw	Decomp
Levels	*n*	○	○
Abs. multiplier	*m*	○	○
Relative multiplier	*q*	○	○
Mul. growth rate	*mp*	○	○

Options

□ Ref databank values
□ Show as shares
□ Show errors
Decimals: .00 .0
Refresh

	2004	2005	2006	2007	2008	2009	2010
y	499.9999	516.5499	529.2876	520.6971	479.9832	478.4205	497.5729
c	332.1999	343.1499	351.8876	347.6070	322.7506	319.3274	330.4765
g	27.0000	31.0000	35.0000	34.0000	24.0000	30.0000	34.0000
x	140.8000	142.4000	142.4000	139.0901	133.2325	129.0931	133.0965

This opens up the decomposition window, showing the variable values, and the values of determining variables (precedents). Cells may be marked and copy-pasted to Excel or other spreadsheets. The first line of the table is the dependent variable (left-hand side of equation), whereas the following lines are the explanatory variables (right-hand side of equation).

If you click "Growth rate" (`p`) in the "Raw" column under "Time-change", the values are shown in annual percentage growth rates. For instance, in 2004, the simulated value of `y` grows with 1.63%, whereas private consumption `c` grows with 3.17%. The percentages regarding `c`, `g` and `x` obviously do not sum up to the 1.63%, which can be verified by marking the three cells, and looking at the status bar at the bottom of the window (reporting 8.27% in total). However, if you click the same button in the "Decomp" column, the percentages regarding `c`, `g` and `x` sum up, because the contributions in the equation are "decomposed". So, in 2004, the simulated value of GDP growth (1.63%) is composed of +2.07% of `c`-contribution, +0.41% of `g`-contribution, and -0.85% of `x`-contribution. That is, it is the private consumption (`c`) that is driving the growth in `y` in 2004, whereas net exports (`x`) pulls in the opposite direction (you may click "Show as shares" to easier spot which variables contribute the most to the changes in `y`).

If you alternatively click the "Abs. multiplier" (`m`) in the "Raw" column under "Multiplier", the absolute multiplier is shown (i.e., the absolute difference between the historical databank and the simulated values, corresponding to MULPRT or PRT<m>). For instance, in 2004, the simulated value of `y` is 33 units larger than the databank value. The multiplier part of the decomposition window is most often used in multiplier analysis, i.e. analyzing two simulated scenarios.

The full code

```
restart;
mode sim;
time 2004 2016;
model gekko;
read gekko;
prt <2004 2016> y, c, x, g;
#vars =  y, c, x, g;
list ?;
prt {#vars};
prt {'*'};
mulprt {#vars};
sim;
mulprt {#vars};
prt @y, y;
mulprt <v> y;
plot @y, y;
sheet @y, y;
prt <p> {#vars};
plot <p> {#vars};
prt <rp> {#vars};
prt <r> {#vars};
write simple;
create g2;
g2 <2002 2016> = g;
pch(g2) <2003 2016> = pch(c) - pch(y) + 2;
prt <2003 2016 p> c, y, g2;
prt <r> c/y;
disp y;
decomp y;
```

3.3.4 Multiplier analysis (shocks)

This section describes how to perform forecast scenarios, based on a historical databank. In order to follow the examples, you must first download the model and databank (click demo.zip and copy the two files gekko.frm and gekko.gbk into your Gekko working folder).

(See the bottom of this page for the full code).

Next start up Gekko in the working folder, and type:

```
restart;
mode sim;
time 2017 2040;
```

This clears up the workspace, and sets the global time period (for which we will later simulate the model).

```
model gekko;
read gekko;
```

The first statements load `gekko.frm` and reads the `gekko.gbk` databank file. The databank file only runs to 2016, since 2016 is the last historical data point. Like in the previous section, create a list containing the variables:

```
#vars =  y, c, x, g;
prt <2015 2040> {#vars};
```

As you can see, the variables contain all missings ('M') for the period 2017-2040. We will now try to simulate the model (for 2017-2040):

```
sim;
```

Gekko will complain about missing data (click the link in "There were missing values in 1 variables", and afterwards click the "Main" tab again to get back (or use Ctrl+M)). So as stated in the Output tab, the problem is that the value of `g` for the period 2017 is needed in order to simulate for 2017. Try setting `g` constant (for the global period 2017-2040) by means of the SERIES statement:

```
g %= 0;
sim;
prt {#vars};
```

The `%=` operator in the first (SERIES) statement sets annual percentage growth, and after this, the model can simulate. The model is really quite simple: as seen in the `x`-equation, `x` will always change unless `y` = 500. So this value for `y` is an equilibrium value (attractor), for instance corresponding to zero (or natural/structural) unemployment level.

The equilibrium level of the consumption equation is given by `c = 0.6/(1-0.1)*y`, corresponding to `0.6/0.9*500 = 333.33`. Net exports are given residually, from the GDP identity. Hence in equilibrium, `x = y - c - g = 500 - 333.33 - 31 = 135.67`. The interpretation is that in the long run there is perfect crowding-out regarding public consumption `g`, since 1 extra unit of `g` will entail 1 less unit of `x`. As it can be seen from the above PRT statement, the actual values converge slowly to these long-run values.

If the best prognosis regarding future government consumption is that it is unchanged at at 31 units, this simulation can be considered a reference scenario. In order to perform a multiplier analysis on top of this reference scenario, we need to put the scenario into the Ref databank (in order to be able to compare it to the alternative scenario later on). Try printing all values for the variable `y` (operator `v` means 'verbose'):

```
mulprt <v> y;
```

As it is seen, there are missing values in the Ref databank. The CLONE statement copies data from Work to Ref, so try this:

```
clone;
mulprt <v> y;
```

As it is seen, Work and Base databank values are now identical, and we are ready to perform an alternative simulation.

The experiment is the following:

```
g += 100;
```

The operator `+=` adds to the value in the Work databank, so in each year, `g` is augmented by an absolute amount of 100 relative to the baseline values. The alternative scenario is simulated:

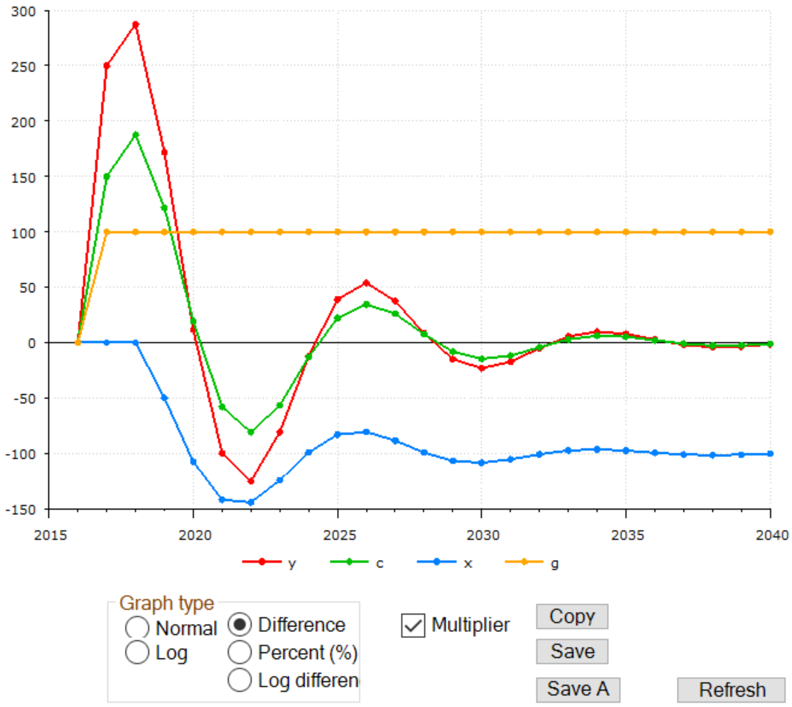
```
sim;
```

Instead of printing, we will use the graph facilities. Try the following statement:

```
time 2016 2040;
plot {#vars};
```

We start out changing the global time period to 2016-40, so that the last historical year is included in the following prints and plots. The plot shows the simulated values (levels), not the multiplier (differences). For instance, you can see that `g` changes from 31 in 2009 to 131 in 2016, as expected.

Now, in the graph window, try to select "Multiplier" and "Difference", in order to see the multiplier.



Graph type

○ Normal ● Difference

○ Log ○ Percent (%)

○ Log differen

☑ Multiplier

Government spending (`g`) changes permanently with 100 units, and in the first year this creates a Keynesian extra effect on production (`y`) which increases with 250 units. The effect on private consumption (`c`) is 150 units.

The effect is augmented a little bit more the following year, because of the lagged endogenous term in the `c`-equation. From 2019 and on, however, the net exports begin to decline (the effect propagates via the term `y(-2)` in the `x`-equation). The underlying interpretation could be that — with some delay — the rising production lowers the unemployment rate and thus augments the wage levels. This in terms puts an upwards pressure on the domestic price level, lowering net exports. This negative effect on `x` pulls `y` back downwards, and as it is seen, `y` oscillates slowly back to its former level (i.e. the multiplier is 0 in the long run, corresponding to full crowding-out).

So in the long run, `y` and `c` are unchanged, whereas `g` has been augmented by 100 and `x` reduced by 100. So, as mentioned above, in the long run the increased government consumption is exactly crowded out by an equivalent reduction in net exports.

If, instead of "Difference" you select "Percent (%)" in the graph window, you can see the percentage multiplier differences. The effect on `g` is quite large in percentages, whereas `y` and `c` follow each other quite closely in percentages.
It is worth noting that you may call PLOT with an operator to indicate how the data should be presented, for instance:

```
plot <m> {#vars};
```

The m operator code indicates absolute multiplier, i.e., the absolute difference between the two databanks. The same code can be used for PRT, for example:

```
prt <m> {#vars};
```

Also, instead of PRT or PLOT, you may use SHEET, provided that Excel is installed on your computer (otherwise use CLIP):

```
sheet <m> {#vars};
```

This creates an Excel sheet with the data. In Excel, a graph is easy to create from this table (for instance, in Excel 2019, you can click on any cell inside the table, choose "Insert", and then choose a graph). As a side note, you may create the Excel file automatically without opening up Excel:

```
sheet <m> y 'GDP', c 'Priv. cons.', x 'Net exp.', g 'Gov. cons.'
file = simple;
```

This will silently create the file simple.xlsx and put it into your working folder. Note the labels on each variable. You may also use expressions as you wish, for example:

```
sheet <m> c/y, x/y, g/y;
```

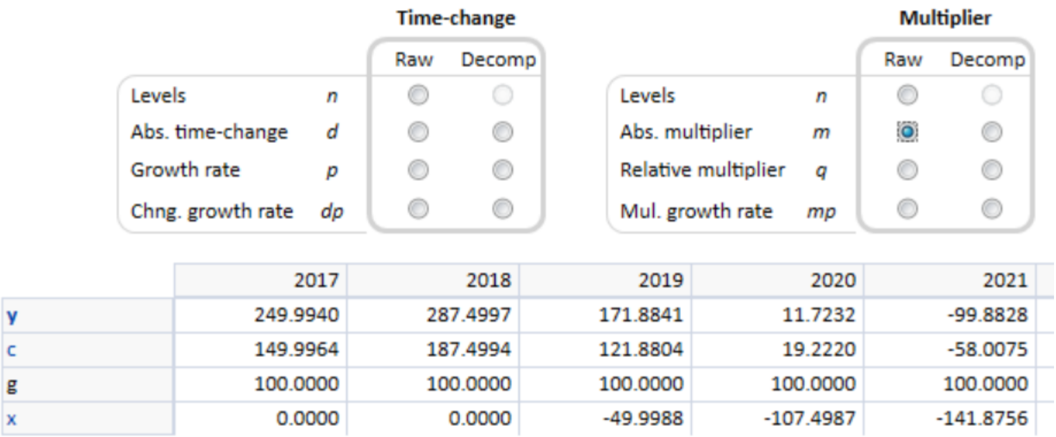
This show absolute multiplier changes in these three rates.

Other convenient operators in addition to m are p for annual percentage growth, and q for multiplier percentage change. The PRT and MULPRT statements also have so-called 'long' operators such as abs or pch. Please consult the PRT help file for more on this (you may for instance use PRT<abs>, PRT<pch> or MULPRT<pch>).

You may also use the DECOMP facility to analyze equations and contributions. For instance:

```
decomp <2017 2040> y;
```

This starts the decomp window: try clicking "Abs. multiplier" (m) in the "Raw" column of the "Multiplier" section. This will show multiplier values related to the y-equation.



Time-change		Raw	Decomp
Levels	*n*	○	○
Abs. time-change	*d*	○	○
Growth rate	*p*	○	○
Chng. growth rate	*dp*	○	○

Multiplier		Raw	Decomp
Levels	*n*	○	○
Abs. multiplier	*m*	●	○
Relative multiplier	*q*	○	○
Mul. growth rate	*mp*	○	○

	2017	2018	2019	2020	2021
y	249.9940	287.4997	171.8841	11.7232	-99.8828
c	149.9964	187.4994	121.8804	19.2220	-58.0075
g	100.0000	100.0000	100.0000	100.0000	100.0000
x	0.0000	0.0000	-49.9988	-107.4987	-141.8756

The full code

```
restart;
mode sim;
time 2017 2040;
model gekko;
read gekko;
#vars =  y, c, x, g;
prt <2015 2040> {#vars};
sim;
g %= 0;
sim;
prt {#vars};
mulprt <v> y;
clone;
mulprt <v> y;
g += 100;
sim;
time 2016 2040;
plot {#vars};
plot <m> {#vars};
prt <m> {#vars};
sheet <m> {#vars};
sheet <m> y 'GDP', c 'Priv. cons.', x 'Net exp.', g 'Gov. cons.'
file = simple;
sheet <m> c/y, x/y, g/y;
decomp y;
```

3.3.5 Add-factors etc.

In order to follow the examples, you must first download the model and databank (click demo.zip and copy the two files gekko.frm and gekko.gbk into your Gekko working folder). Next start up Gekko in the working folder, and type:

(See the bottom of this page for the full code).

```
restart;
mode sim;
time 2017 2040;
```

This clears up the workspace, and sets the global time period (for which we will later simulate the model). Next, issue these statements, in order to run the baseline simulation with 0% growth in government consumption.

```
model gekko;
read gekko;
g %= 0;
sim;
clone;
```

Gekko auto-creates add-factors depending on formula codes. Imagine some stochastic shock on the net exports (e) in 2017. The e-equation has equation code _GJ. You may read more about such codes in the MODEL help file, but suffice to say that the code means that the equation in reality reads:

```
FRML _GJD        dif(x) = -0.2*(y[-2] -
500);                    //original
                     x = x[-1] - 0.2 * (y[-2] - 500) +
JDx;      //result
```

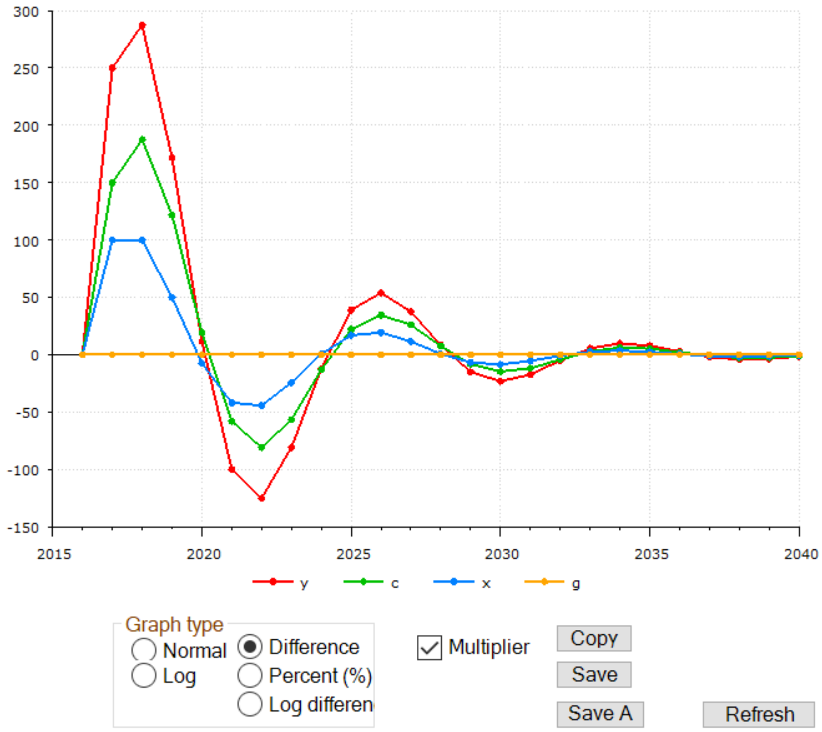
Note the add-factor JDx, and note also that the left-side dif()-function has also been resolved. This latter equation is the one that Gekko actually simulates, and these "detailed" equations can be shown by means of the DISP statement:

```
disp x;
```

Try click "Show detailed equation", in order to see the "full" equation. So even though the variable JDx is not directly seen in simple.frm, it exists as a hidden exogenous variable on the right-hand side of the x-equation. When simulating, the value is automatically set to 0 if no other value is given.

Now, we may try to change the add-factor in the year 2017 (i.e., a temporary shock), simulate and plot the result:

```
jdx <2017 2017> += 100;
sim;
plot <2016 2040 m> y, c, x, g;
```



In the first two periods (2017 and 2018), net exports `x` are augmented by 100, since the two period lagged y does not begin to affect the equation before 2019. (Since the equation is a difference equation (with `dif(x)` on the left side), augmenting the add-factor in one year works the same way as augmenting an add-factor permanently in a non-difference equation).

It is seen that there are no long-run effects — in the long run all effects are 0. So temporary shocks on net exports only creates temporary fluctuations, but these are still quite significant due to amplification via the Keynesian multiplier effect.

Gekko contains some more advanced possibilites regarding add-factors and formula codes. The `c` equation has equation code `_GJ_D`, i.e. there is a `D` in the 5th position. This implies that three variables are created: `Dc`, `Jc`, and `Zc`.The "full" `c`-equation is this: `c = (0.6*y + 0.1*c[-1] + Jc) * (1-Dc) + Dc*Zc`. The variables `Dc` and `Zc` can be used to exogenize an equation. If `Dc` is set to 1, the equation will reduce to `c = Zc`, so that `Zc` can be used to set the target for `c`. See also the last part of the MODEL help page.

The full code

```
restart;
mode sim;
time 2017 2040;
model gekko;
```

```
read gekko;
g %= 0;
sim;
clone;
disp x;
jdx <2017 2017> += 100;
sim;
plot <2016 2040 m> y, c, x, g;
```

3.3.6 Goal-search etc.

In order to follow the examples, you must first download the model and databank (click demo.zip and copy the two files gekko.frm and gekko.gbk into your Gekko working folder).

(See the bottom of this page for the full code).

Start up Gekko in the working folder, and type:

```
restart;
mode sim;
time 2017 2040;
```

This clears up the workspace, and sets the global time period (for which we will later simulate the model). Next, issue these statements, in order to run the baseline simulation with 0% growth in government consumption.

```
model gekko;
read gekko;
g %= 0;
sim;
clone;
```

Gekko can do goals/means analysis, i.e. exogenizing some endogenous variables, and endogenizing some exogenous variables. In other words: force some endogenous variables to attain specific values, by means of some other (exogenous) variables. In this scenario, how would you augment `y` with 250 units for a 7-year period relative to the baseline scenario, by means of changing `g`? This experiment is done quite easily, via the EXO and ENDO statements:

```
exo y;
endo g;
```

These statements state that `y` should be redefined as an exogenous (goal) variable, whereas `g` is redefined as an endogenous variable (means). Note that when you simulate, the number of exogenized and endogenized variables should always be the same, otherwise Gekko will complain. Note also that a small target icon appears at the bottom right, to remind you that goals/means are set. After this, try to change `y`:

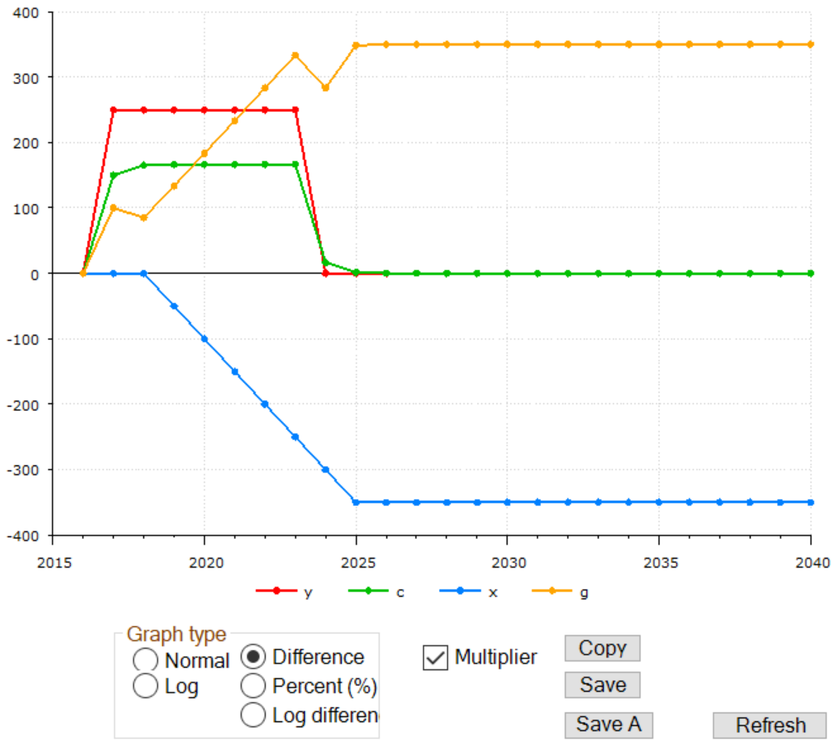
```
y <2017 2023> += 250;
```

You get a warning that you are updating a left-hand side variable. Now simulate (we assume the global time period is still 2017-2040):

```
sim<fix>;
```

To make the means/goals bind, you must use the `<fix>` option in the SIM statement. Plot the simulated variables (multiplier):

```
plot <2016 2040 m> y, c, x, g;
```



Graph type
○ Normal ● Difference
○ Log ○ Percent (%)
○ Log differen
☑ Multiplier

It is seen that in order to keep the production level 250 units higher for 7 consecutive years, government spending has to keep growing continually up to 2023. In that period, net exports (`x`) get squeezed out, and the effect on `x` is negative from 2019 on (implying a likely trade deficit, depending upon the price of imports and exports). When the `y`-effect ends after 2023, `g` stays at a permanently higher level (350 units higher). The mirror-image of this is that net exports has diminished by the same amout (-350). You may compare this graph to the similar experiment in section 4.

To deactivate all goals/means, simply use SIM instead of SIM<fix>, or use the UNFIX statement to remove the goals/means completely.

If you want to fix a certain endogenous variable at some predefined value, by means of its add-factor, you may use ENDO/EXO for that, too. But in that case, there is an often more convenient method, namely using the so-called exogenization dummies. If the equation has a formula code implying exogenization dummy (the character `D` at the 5th position), this is possible.

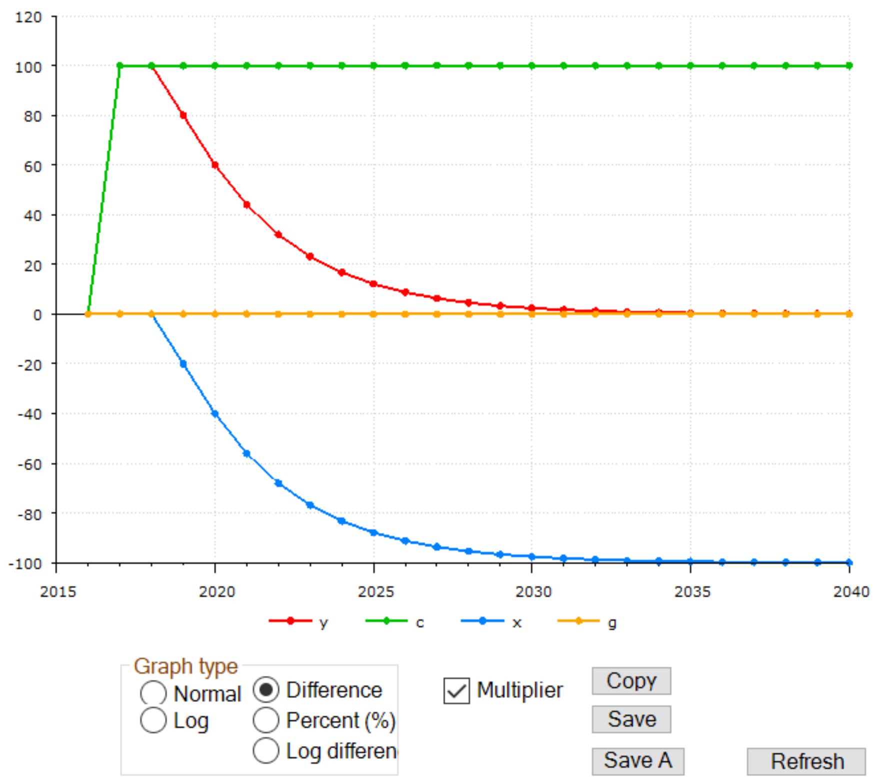
The private consumption function `c` has code `_GJ_D`, implying that it has an additive add-factor (the `J_` part of the code), and exogenization dummy (the last `D` in the code). So the "full" `c`-equation really is `c = (0.6*y + 0.1*c[-1] + Jc) * (1-Dc) + Dc*Zc`. The `J_` part of the code "produces" the add-factor `Jc`, whereas the last `D` in the code "produces" the terms containing the variables `Dc` and `Zc`. The variable `Dc` is an exogenization dummy (normally 0), and the variable `Zc` can be used to set a value for c (if `Dc = 1`, the equation reduces to `c = Zc`).

Try starting from scratch with the scenario again (you may mark a block of statements and run them all when you hit `[Enter]`):

```
restart;
mode sim;
time 2017 2040;
model gekko;
read gekko;
g %= 0;
sim;
clone;
```

Now try the following statements:

```
Dc = 1;
Zc += 100;
sim;
plot <2016 2040 m> y, c, x, g;
```



Graph type
○ Normal ● Difference
○ Log ○ Percent (%)
○ Log differen

☑ Multiplier

As you can see, `c` is now fixed at a value 100 higher than in the baseline databank. In the long run this crowds out net exports `x`, so that GDP (`y`) is unchanged near 2040. The functionality can for instance be used in forecasts, if some observations of endogenous variables are known ahead of time (possibly tentatively, from other sources, indicators etc.).

At this point it should also be mentioned that it is possible to re-endogenize the variable `c` in a convenient fashion. If, after the above simulation, `Dc` is set to 0 again, and the model simulated, one would expect `c` to return to its previous values. This is not the case however: the value of `c` would stay at +100, even though the dummy has been set back. The explanation is that the add-factor `Jc` is calculated in a way so that the `c`-equation keeps it's goal values. This trick of switching the dummy back and forth while simulating is quite advanced, and should only be used if the user fully understands the implications.

As you may have noted, in order to solve this goals/means problem, Gekko had to switch to another solving algorithm (Newton). This is done automatically, when EXO and ENDO statements are used. The Newton algorithm is more powerful that the Gauss algorithm (and allows changing the set of endogenous/exogenous variables). But for "normal" non-exogenized models, Newton is a bit slower, so for normal (non-goal) purposes, the Gauss algorithm is typically used.

If, however, you encounter solving problems with the Gauss algoritm, try switching to the Newton algorithm manually (`option solve method = newton;`). Also, the Newton algorithm is much more precise. This has to do with the fact that for each extra iteration, the Newton algorithm doubles the number of correct significant digits in the values of the endogenous variables. Gekko is designed for large-scale models, so you may indicate hundreds or thousands of simultaneous goals/means at the same time, if you like.

It should perhaps be noted that some kinds of equations can never be solved by means of a Gauss-Seidel algorithm, no matter how much damping etc. might be used. An example could be the one-equation model `z = 1.1*z - 0.2*h`. This model has the solution `z = 2*h`, but this solution can only be found with the Newton algorithm. If such equations are present in a model (possibly in hidden form), the solution space becomes a saddle point, and that requires a Newton solver. However, many macroeconomic models solve just fine with the (faster) Gauss solver.

The full code

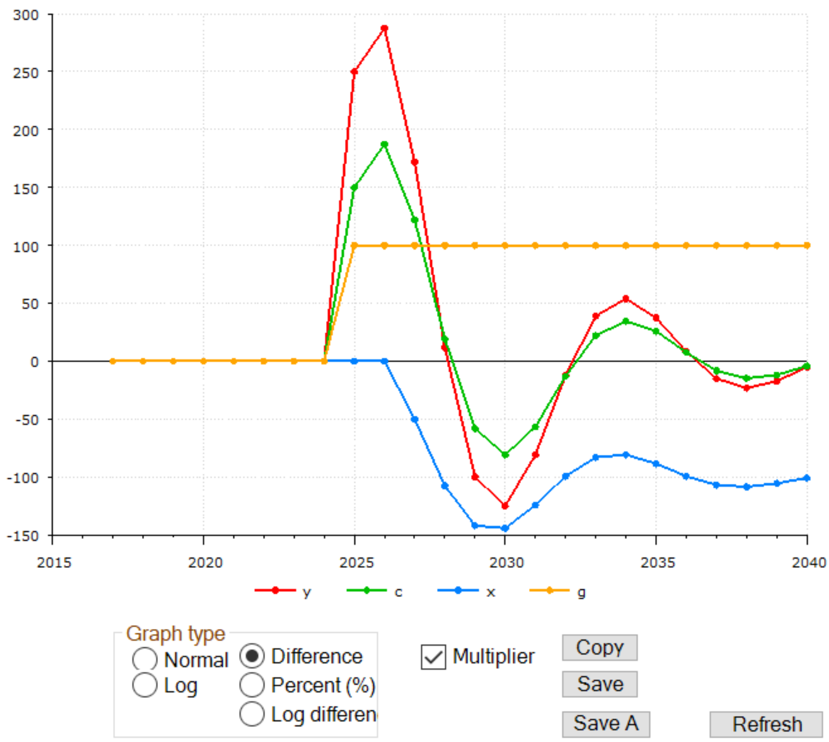
```
restart;
mode sim;
time 2017 2040;
model gekko;
read gekko;
g %= 0;
sim;
clone;
exo y;
endo g;
y <2017 2023> += 250;
sim<fix>;
plot <2016 2040 m> y, c, x, g;

// -------------------------------

restart;
mode sim;
time 2017 2040;
model gekko;
read gekko;
g %= 0;
sim;
clone;
Dc = 1;
Zc += 100;
sim;
plot <2016 2040 m> y, c, x, g;
```

3.3.7 Forward-looking

Gekko can handle forward-looking models, that is, models where one or more of the endogenous variables contain leads. There are different solvers regarding this, see under SIM, and also under OPTION solve forward ... . We will first try to perform a multiplier analysis without leaded variables, similar to the shock shown in section 4:

```
restart;
mode sim;
time 2017 2040;
model gekko;
read gekko;
#vars =  y, c, x, g;
g %= 0;
sim;
clone;
g <2025 2040> += 100;
sim;
plot <m> {#vars};
```

If you copy-paste these statements to Gekko, you may either execute them individually one by one by pressing `[Enter]`, or first mark them as a block and then press `[Enter]` to execute them at once. The "multiplier" (difference between the two simulations) looks like this:



Graph type
○ Normal ● Difference
○ Log ○ Percent (%)
○ Log differen

☑ Multiplier

This is the same effects as seen in section 4, were in this case, `g` is not augmented until the year 2025. In the model (gekko.frm), there is the following equation:

```
FRML _GJ_D          c  =  0.6*y + 0.1*c[-1];
```

So if `y` is augmented by 1, `c` will augment by 0.6 in the same year. We will now create a new model with a different equation regarding consumption, `c`.

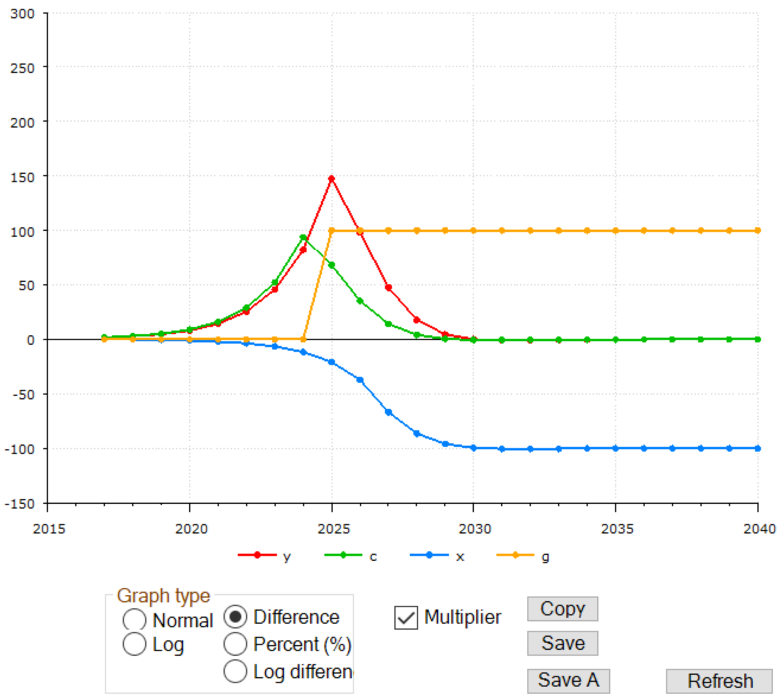
- In the file system, make a copy of gekko.frm and call it gekko2.frm
- In gekko2.frm, in the `c` equation, change `0.6*y` into `0.6*y[+1]`.

So the equation now looks like this:

```
FRML _GJ_D          c  =  0.6*y[+1] + 0.1*c[-1];
```

This means that current consumption, `c`, now reacts to the expected income next period, `y[+1]`. Next, run the following code:

```
restart;
option solve forward terminal = exo;  //changed
mode sim;
time 2017 2040;
model gekko2;  //changed
read gekko;
#vars =  y, c, x, g;
g %= 0;
y[2041] = 500;  //changed
sim;
clone;
g <2025 2040> += 100;
sim;
plot <m ymax = 300> {#vars};
```



Graph type
○ Normal ● Difference
○ Log ○ Percent (%)
○ Log differen

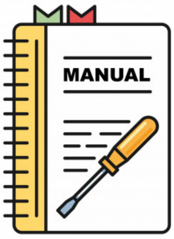
☑ Multiplier

This is quite different from before, and it is seen that both `c` and `y` start to increase well before `g` increases in 2025. In the figure, it is seen that `c` now reacts to `y` in the next period, for instance `y` decreases from 2025 to 2026, and hence `c` decreases the year before that, from 2024 to 2025.

In the above code, we have set y[2041] = 500, since this is the long-run equilibrium value (the only value for which the equation `dif(x) = -0.2*(y[-2] - 500)` is stationary). Hence, we are setting the terminal condition manually for this problem.

Part IV

4 Gekko User Manual



The Gekko User Manual contains detailed descriptions of statements, functions, and other components. In principle, these descriptions are exhaustive, but for some of the components, the explanations may provide too much detail.

If so, there are easier explanations available in the Gekko User Guide, aimed at beginners and new users, and highlighting only the main points.

The user manual contains the following sections:

- User manual introduction. Intro to some of the basic concepts**.**
- Syntax basics. The fundamental concepts of the syntax is described.
- Gekko statements. This section describes in detail the purpose of the different Gekko statements, the syntax to be used, the results produced, together with examples etc.
- Gekko functions. Gekko functions can be used in expressions. The input parameters and the output type is described. The functions are divided into categories.

4.1 User manual introduction

This section describes some of the basic concepts used in Gekko. Beware that Gekko also has a user guide, which is intended as an easier introduction with less details (for new users). For new users, it is perhaps best to start with the user guide, to avoid too many details.

The present introduction contains the following sections:

- Basic concepts. An overview of some of the main capabilities and concepts of Gekko.
- Time periods. Explaining how global time and local time works.
- Databank search. Describes how databanks are searched for variables in Gekko.
- Wildcards. Explains the special rules concerning wildcards and type/frequency symbols.
- Naked list. Explains the logic of 'naked lists' like `#m = a, b, c;`.
- Dynamic statements. How to handle statements like `x = x[-1] + 1;`?
- Missing values. The logic of missing values.
- Filenames. How filenames and paths are handled in Gekko.
- Help system. Description of the in-built help system.
- Under the hood. A short section on some of the main technologies/components used in Gekko.

4.1.1 Basic concepts

The following describes some of the fundamentals of Gekko.

Timeseries-oriented

Among other things, Gekko handles timeseries (often just called 'series' in this documentation). Gekko is a timeseries-oriented software system, that is, it has easy handling of timeseries as one of its main objectives. Because of this orientation, it is often not necessary to indicate a time period when dealing with timeseries variables, because the time dimension is implicitly understood. Gekko can operate on different frequencies, at the moment annual (`a`), quarterly (`q`), monthly (`m`), weekly (`w`), daily (`d`) or undated (`u`). IMPORT can handle (collapse) higher frequencies than daily, if needed. Note that week numbering follows the ISO 8601 standard.

Gekko handles other kinds of variable types, too, as described in the next section.

Databanks

Databanks are in-memory storage of variables types series, value, date, string, list, map, and matrix. The names of values, dates and strings (scalars) always start with the `%` symbol, whereas the names of lists, maps and matrices (collections) always start with the `#` symbol. Databanks are opened in a hierarchical list of open databanks, where the first-position databank has number 1 (cf. the F2 window: click F2 to see the list of open databanks).

Gekko can READ/WRITE such databanks as external Gekko databank files (.gbk extension), or IMPORT/EXPORT data from/to other file formats. Gekko always starts out with four empty databanks (in memory): `Work` (first-position bank), `Ref` (reference bank), `Local` (local variables), and `Global` (global variables). At start up, the `Work` databank has number 1 in the list of open databanks, with the `Ref` (reference) databank shown just below. In addition, more databanks with different names (so-called 'named' databanks) can be opened (OPEN), but note that the first-position and reference databanks have special capabilities regarding printing/plotting/comparing etc. The local and global databanks are used to store and access temporary variables (local), or store settings etc. (global). See the LOCAL and GLOBAL statements for more on this. Try to click F2 to see the list of open databanks (note that reference, local and global databanks are only shown in that window, when they contains data).

You may open other databanks as first-position databank with OPEN<first>/OPEN<edit>. You may use CLOSE to close a databank (and possibly write it to file, if it has been altered).

If you need to import data into a databank, you may use IMPORT for non-Gekko data, or READ for Gekko databanks. For instance, `IMPORT <xlsx> data.xlsx;` imports

Excel-data into the first-position databank, but you could alternatively use `IMPORT<ref xlsx> data.xlsx;` to import the data into the reference databank. For simulation purposes, the READ statement is often practical. You may use WRITE to write the contents of the first-position databank to an external .gbk file.

As mentioned, in the F2 window, the first-position databank has number 1, whereas other 'named' databanks have numbers 2, 3, etc. If the local databank contains data, it will show up in the list above the first-position databank, and if the global databank has data, it will show up last in the databank list. When issuing a statement like `prt x;` or `y = 2 * x;`, the way Gekko looks for `x` depends upon databank search options. In sim-mode, Gekko will only look for `x` in the first-position and local/global databank, whereas in data- and mixed mode, Gekko will look for `x` first in the local databank (first bank in the databank list), then in the first-position databank (number 1 in the databank list), then in other open databanks (numbers 2, 3, ... etc. in the databank list), and finally in the global databank (last bank in the databank list). Note here that the reference databank is never searched for variables without databank indication, since this databank is only used for comparison purposes, cf. MULPRT, PLOT<m>, COMPARE, and similar statements. You may refer to a variable in the reference databank with `ref:x` or the shorter `@x`. In general, you can use colon (`:`) to refer to a particular databank, for instance `b1:x` tries to find `x` in the `b1` databank.

Timeseries variables

A timeseries (or just: series) can can have frequency annual, quarterly, monthly, weekly, daily, or undated, and it may contain any number of observations. If data has been read for timeseries `x` regarding the period 2010-2015, printing out `x` for the period 2016 will show a missing value ('M'). Series can be lagged and leaded, for instance `x[-1]` or `x[+1]` in the sense that if `x` corresponds to `x(t)` , `x[-1]` corresponds to `x(t-1)`, and `x[+1]` corresponds to `x(t+1)`. Note that lags must start with the symbol `-`, and leads with `+`, these may no be omitted. Individual observations can be picked out with for instance `x[2020]`, or `x[2020q3]`, the latter being the third quarter of 2020, if `x` is a quarterly series. If you need accumulating lags like `x = x[-1] + 1`, consider using `x <dyn> = x[-1] + 1`, or a BLOCK with `series dynamic = yes`. Accumulating lags are explained in more depth here.

Array-series

An array-series is a special kind of multidimensional timeseries, where indexing with for instance `x[a]` or `y[b, c]` is possible. You may use `x['a']` or `y['b', 'c']` as synonyms in that case, and array-series are practical for many purposes, instead of for instance using naming conventions like `xa` or `ybc`. In general, you can perform the same operations with the array-series `x[a]` as you would be able to do with a normal timeseries. Array-series must be defined before they are used, for instance `x = series(1); y = series(2);` to state the dimensionality.

Scalars

Gekko scalars are the types value, date or string. Scalars names must begin with the symbol `%`. Scalars can be thought of as containing just one element: the single value, the single date, or the single string. Values are numeric values like `1.2` or `2e8`, dates are for instance `2020` or `2020q3`, and strings use single quotes like `'dk'`.

Collections

Gekko collections are of the type list, map or matrix. Collection names must begin with the symbol `#`. Collections can be thought of as a number of elements bundled together inside the collection. A list stores variables in a sequence (by numbers 1, 2, etc.), a map stores variables by name (like `dkk`, `eur`, `usd`), and matrices are 2-dimensional structures of numeric values, also ordered by numbers 1, 2, etc. So for a list, `#x[2]` refers to the second element. For a map, `#x[dkk]` refers to the element that has this name, and for a matrix, `#x[2, 1]` refers to the numeric value stored in row 2, column 1. It should be mentioned that it is possible to write `#x[dkk]` as `#x['dkk']` or `#x.dkk`, too.

Lists and maps can store any other variable types as elements, whereas matrices only store values (for the time being).

Banks, maps, and array-series

Note: all these three datastructures look up elements by means of names (also called look-up keys). For instance, `b2:x` looks up `x` in bank `b2`, `#m['x']` looks up `x` in the map `#m`, whereas `y['a', 'x']` looks up `('a', 'x')` in the two-dimensional array-series `y`. Inside Gekko, banks and maps are built in the same way, whereas array-series are a bit different in that they (a) only store series inside, and (b) allow several dimensions of look-up keys.

Strings as name references

Scalar strings (or a list of strings) may refer to other variables. Consider the string `%s = 'b2:x!m';`. If you state `prt %s;`, Gekko will just print out the raw string. But if you use `prt {%s};`, Gekko will instead print out the monthly series `x` from the `b2` databank (in a sense performing a *forwarding* operation, forwarding from the variable `%s` to the variable `b2:x!m`). Therefore, the curly {}-braces are handy regarding name composition. Also, if `%i = 'b'`, and `%j = 'd'`, the variable `a{%i}c{%j}e` is equal to `abcde`, and in general one should read the curly braces {} as if they are simply a sequence of unknown characters that are glued to other characters (or other {}-braces). See more on strings, or see the syntax diagrams.

A list of strings may be used in the same way. Consider the list of strings `#m = ('b2:x!m', 'y');`. If you state `prt #m;`, raw strings are printed out, whereas `prt {#m};` will print out the variables corresponding to the strings (monthly series `x` from the `b2` databank, and the series `y`, and again performing a forwarding operation). In general, list definitions are enclosed in parentheses, like `#m = ('a', 'b', 'c');`, but for simple strings, the equivalent 'naked' list `#m = a, b, c;` is legal, too. In the latter case (naked list), it should be emphasized that the list elements are still three strings `'a', 'b', 'c'`, not the variables themselves.

Sometimes the user may be in doubt whether he or she should use a normal string `%s`, or a name-reference `{%s}`? In such cases, the "abc test" may be performed. Would it be natural to use a quoted string like `'abc'` in the statement, or would it be natural to use a name like `abc` instead? If the former is the case, use `%s` -- if the latter is the case, use `{%s}`.

Wildcards

In general, wildcards are stated with the `['...']` or `{'...'}` patterns, depending upon the context. For instance, you can use `#m = ['a*x']` to obtain a list of strings of variables starting with `a` and ending with `x` (from the first-position databank, with the current frequency). If you need to for instance print out the variables in this list, `prt ['a*x']` will just print out the raw strings corresponding to the matched wildcard. Instead, `prt {'a*x'}` should be used to print out the variables themselves. Ranges can be stated as for instance `'pxa..pxe'`, or `'bank:pxa..bank:pxe'`.

In INDEX, COPY, RENAME, DISP and similar statements, you can omit the curlies and single quotes, for instance `INDEX a*x;` (`a*b` will not be interpreted as a mathematical product in that statement). You can also use `?` to select a single character, and wildcards can also be used to search for banks, frequencies, and indexes. See the INDEX and COPY statement for more on this.

Much more on wildcards on the wildcards page.

Analysis

In sim-mode, the reference databank is typically used for multiplier analysis (i.e., experiments). Say you read a databank and then perform some experiment. This experiment will only alter timeseries in the first-position databank, so after the experiment is finished, you can compare the timeseries in the first-position and reference databanks (Gekko has a lot of statements to do such comparisons, for instance MULPRT, DECOMP etc.). If, at some point, you wish to make sure that the first-position and reference databanks are identical (for instance after a simulation), you can use the CLONE statement. This statement clears the reference databank, and copies the first-position databank into it (in memory). You may alternatively read a file directly into the reference databank by means of READ<ref>.

There is also a cleanup-statement: RESTART. This statement clears the first-position and reference databanks, in addition to clearing models, variables, user functions, procedures, and other things. The operation provides a clean state of Gekko, as if it had been closed and reopened (if there is a file with the name `gekko.ini` in the program and/or working folder, this file will be re-read, so `gekko.ini` can be used to contain options and other statements, for instance MODEL and READ statements, that the user wishes to "survive" a RESTART). If you wish a clean state without any potential gekko.ini file, use RESET.

Creation and sim-mode

In sim-mode you have to CREATE a series before you update its values/observations with the SERIES statements (unless the timeseries starts with the letters `xx`, indicating that it is to be thought of as a temporary variable). However, it should be noted that when a databank is read (READ), after a model has been loaded previously (MODEL), any model variables not present in the databank will be auto-created as timeseries (with all observations set to missing values). Because of this, it may often be convenient to put MODEL statements before READ statements. In data- and mixed mode, timeseries are auto-created with the SERIES statement.

Periods

Note that statements involving series variables can include a local time period, like for instance `prt <2010 2020> x, y;`. The local time period will overrule the global time period, which can only be set via the TIME statement.

There are some details regarding periods. Most statements that involve timeseries use the global time period if no local time period is stated, for instance statements like PRT, IMPORT, EXPORT, etc. For some of these statements, you may use local option `<all>` to use all existing data points (observations). You cannot combine `<all>` with a local time period.

But for the statements COPY, READ and WRITE, omitting a local time period does not entail the use of the global time period. These statements will use all existing data points (observations) for all series, if no local time period is stated. If a local time period is stated, only the local sample is used, and if you need to observe the global time period, you can use the `<respect>` option. You cannot combine `<respect>` with a local time period.

For some statements, you may use a time period with another frequency than the series object used. In that case, Gekko will try to convert the frequency meaningfully. For instance, `prt <2010q2 2010q3> x!q, x!m;` will just use 2010q2-q3 for the quarterly series `x!q`, whereas 2010q2-q3 is converted to 2010m4-m9 for the monthly series `x!m` (covering from the start of q2 to the end of q3).

If you need to change the time period temporarily, you may use the BLOCK structure. Also, user defined functions and procedures may use a `<>`-field to indicate time period arguments.

Operators

The so-called operators are used in many places, in order to perform easy transformations (for instance percentage growth rate, or multiplier difference between first-position and reference databank values). The operators come in two versions: 'long' and 'short'. The 'long' ones are used in the PRT and MULPRT statements (for instance `prt<abs> var1;` to only print the absolute level, and not percentage growth), whereas the 'short' ones can be applied more generally (for instance `plot<p> var1;` to graph the growth rate of `var1`). The most important of the 'long' ones are `dif` and `pch`, and the most important of the 'short' ones are `d`, `p`, `m`, `q`. The functionality of the 'long' and 'short' operators overlap: see PRT for more details. The short operators will also show up in TABLEs and the DECOMP window, and can also be used in SERIES, PLOT and SHEET.

Models

Regarding models, it should also be noted that the list of endogenous variables in a model is simply the set of all the variables at the left-hand side of the equations. This may be changed afterwards by means of the ENDO and EXO statements. Regarding equation syntax, you may consult the latter part of the MODEL help file, if you need more information on this.

Files

Regarding file names, you may use relative paths like `\subfolder\data.txt`. Using relative paths makes it easier to move a system of Gekko program files to another location/computer if needed. Special user-paths can also be given by means of the `OPTION folder ...` settings. If the path or filename contains blanks or special characters, you may enclose it in single quotes.

Gekko program files/batch jobs

Gekko programs can either be run directly from the Gekko main window, or assembled in a program file (script file) for later execution. Gekko program files can be run with the RUN statement. This is also called a batch job (also possible by means of calling gekko.exe with parameters, see more in the RUN help file). You may track the execution of jobs via 'Utilities' --> 'Run status...' in the Gekko menu (or double-click on the traffic light in the lower right corner of Gekko).

Gekko also provides user-defined functions and procedures to deal with repetitive tasks.

4.1.2 Time periods

Many Gekko statements accept a local time period stated inside the <>-brackets. For such statements, omitting a local time period generally means that the global time period (cf. TIME) is used instead. There are the following exceptions to that rule:

List of statements where lack of local period means all observations

COPY	Handling variable objects, copying them inside the same databank or between databanks.
READ, WRITE	Read or write variables from an external file.

For instance, `copy x to y;` will copy the entire object, including all observations (and not just the observations corresponding to the global time period), whereas `copy <2010 2020> x to y;` will only copy the observations 2010-20. To copy only the observations corresponding to the global time period, use `copy <respect> x to y;`. Similarly, READ<respect> and WRITE<respect> can be used.

It should be noted that the similar statements IMPORT and EXPORT respect the global time period, in contrast to READ and WRITE. To force IMPORT and EXPORT to use all observations, IMPORT<all> and EXPORT<all> can be used.

Note

User defined functions and procedures may also use a <>-field to indicate time period arguments. This can be used to define a local time period to be used inside the function/procedure.

4.1.3 Databank search

Databank search is an important concept in Gekko, alleviating the burden of always having to state the databank name when variables in other databanks than the first-position databank are used.

Databanks search can be controlled with an option, or via the MODE statement, cf. the next section. In general, databank search can be practical, but the user should be aware of the pitfalls, especially if several open databanks contain variables with the same names.

The remainder of this page tries to answer the following question: In statements like `y = x;` or `prt x;`, where should Gekko look for the variable `x`, if it is not found in the first-position databank?

The option and MODE

The option that controls databank searching is the following:

```
option databank search = yes;  //yes|no
```

Per default, this option is set to `yes`, since Gekko starts up in data-mode. In sim-mode, the option is set to `no`, so `prt x;` will fail in sim-mode, if `x` is not found in the first-position or local/global databank (in that case, Gekko will not look for x elsewhere).

The mode can be switched like this:

```
mode sim; //sim|data|mixed
```

Among other things, the MODE statement controls `OPTION databank search`, and a few other options.

How does databank searching work?

When databank searching is active (`option databank search = yes`), Gekko will look for a variable `x` in the hierarchical list of open databanks (cf. the F2 window that is opened when pressing the F2 key). As shown on the page about OPEN, Gekko operates with the following databanks:

Number	**Searchable**	**Non-searchable**
	Local	
1.	First	Ref
2.	Another databank	
3.	Another databank	
...	...	

n'th	Last databank	
	Global	

So if databank searching is active, Gekko will first look for a variable x in the Local databank. This is often empty, since it is only used for temporary (discardable) variables. Next, Gekko looks in the first-position databank, which is often the Work databank. If not found in any of these databanks, Gekko looks in any other databank opened with the OPEN statement. If not found in any of these, the Global databank is queried at last. Note that the Ref databank is never searcheable, so a variable x without databank indicator (a 'bankless' variable) will never be looked for in the Ref databank.

The search hierarchy means that variables may shadow/mask each other. If searching is active, the user may put a variable x in the Global databank for later use in a system of Gekko program files (the Global databank is not affected by READ, CLEAR, etc. and is therefore practical for long-term storage of global variables). But if the system of program files creates a variable x, or opens a databank containing a variable x, the x in the Global databank is masked. For instance, the user may state x = 100;, creating a series x with the value 100 in the first-position databank. After this, prt x; will refer to this variable, not the variable in the Global databank (to refer to that variable, global:x could be used). So when databank searching is active, and databank indicators are omitted, the user should keep name collisions in mind.

There is another pitfall regarding databank searching, namely that deleting a variable may bring back another variable from being masked. Consider this example:

```
reset;
open <edit> bk1; clear bk1; x = 100; close bk1;
open <edit> bk2; clear bk2; x = 200; close bk2;
open bk1, bk2; //press F2 to see the databank list
prt x;
close bk1;
prt x;
```

The first PRT prints x as 100, whereas the second print prints x as 200. The reason is that in the first print, x is first found in the bk1 databank, whereas in the second print, x is first found in the bk2 databank (because bk1 was closed). So closing a databank, or deleting a variable may have the consequence that a variable with the same name is unmasked in some databank lower in the search hierarchy.

If the variable names in the different databanks are distinct, this is not a problem, and it is practical to be able to refer to variables without always having to write the databank name. Also, in some circumstances, databank masking can be used for selection. Consider two databanks, bk1 and bk2, where the quality of the data in bk2 is inferior to the quality of the data in bk1 (for instance because bk2 is an older databank). In that case, if the databanks are opened with bk1 before bk2, databank searching works as a quality filter. If a variable `x` exists in `bk1`, this version is always used. If it does not exist in `bk1`, Gekko will look for it in `bk2` instead. If it does not exist in `bk2` either, an error will be issued. In that way, the newest version of `x` is always found (or an error occurs).

Statements without databank search

A few statements disallow databank searching completely in order to avoid ambiguities. In these statements, `bk1:x` is still understood as `x` from the `bk1` databank, but `x` without databank indicator will be understood as `x` from the first-position databank, without looking elsewhere for the variable.

List of statements where bankless variables are never searched for

COPY, DELETE, RENAME	Handling variable objects
COUNT, INDEX, DISP	Finding and displaying variables.
EXPORT, WRITE	External file storage
DOC, REBASE, TRUNCATE	Similar to the left-hand side in assignments, therefore no databank searching.

Note: Wildcards without databank indicator are never searched for in other databanks than the first-position databank. So the table deals with 'normal' bankless variables. In some of these statements, `*:x` can be used to indicate the occurrence of `x` in all databanks.

As an example, `copy x to y;` only looks for `x` in the first-position databank, and if it is not found, an error is issued. If it is found, it is copied as a new variable `y`, also in the first-position databank. If the COPY statement allowed searching, the origin of `y` would be unclear, since it could origin from some other open databank than the first-position databank.

Note

Note that a bankless variable on the left-hand side of an expression is always interpreted as residing in the first-position databank. For instance, `x = 100;` will always put the series `x` into the first-position databank (implicitly using `first:x = 100`").

Note that the `Local` or `Global` databanks are always searchable, independent upon MODE etc.

4.1.4 Wildcards

Wildcards are used to search for variables in one or more databanks. Internally i Gekko, a wildcard search returns a list of variables in the form of strings, possibly with databank names and frequencies. There are special rules present regarding how these wildcards work regarding type and frequency symbols.

Array-series can be searched quite straight-forwardly with syntax like `x[a*]`, cf. the INDEX statement. Wildcards can also be used to search inside a list of strings (cf. LIST).

Beware that the keys `[Tab]` or `[Ctrl+Space]` offer autocompletion on timeseries names (cf. here), which implicitly amounts to a wildcard search.

In the following, the logic of databank search is described.

Basics

Wildcards for bank searching come in three flavors (here matching variables starting with 'x' and ending with 'y'):

- String wildcards: `['x*y']`. Returns matching strings
- Name wildcards: `{'x*y'}`. Returns matching names. Actually short for `{['x*y']}`, cf. the note at the end.
- Naked wildcards: `x*y`. Returns matching names, short for `{'x*y'}`.
- Note that wildcards can be used for array-subseries too, for instance `index x[x*y];` (cf. INDEX).

Ranges and single character matches are possible too, for instance `'x1a..x2z'` or `'x?y'`.

The difference between returning strings or names can be seen in this example:

```
time 2010 2012;
create x1y, x2y;  //only necessary in sim-mode
x1y = 1; x2y = 2;
prt ['x*y'];
prt {'x*y'};
prt x*y;  //fails
```

Result:

```
 ['x*y']
 'x1y', 'x2y'   [2 items]

                  x1y          %            x1y          %
  2010         1.0000          M         2.0000          M
  2011          1.0000      0.00          2.0000      0.00
  2012          1.0000      0.00          2.0000      0.00
```

So the first PRT prints out a list of strings equal to `('x1y', 'x2y')`, whereas the second PRT prints out the two timeseries `x1y` and `x2y`. The third PRT fails, since it expects to multiply two timeseries `x` and `y`.

However, in some statements like COPY, RENAME, INDEX, DISP, etc., naked wildcards are allowed, for instance `index x*y;` to get a list of variables starting with `x` and ending with `y`.

```
time 2010 2012; create x1y, x2y; x1y = 1; x2y = 2;
index {'x*y'};
index x*y;   //same: naked form
```

Both INDEX statements print out `x1y, x2y` as matching items, DISP would print the two series.

Wildcards without bank indicator only search for the variables in the first-position databank. To search in all databanks, use for instance `index *:x*y;` or `prt {'*:x*y'};`. As an example, consider the case where there are the following databanks present:

Databank	**Variables**		
1. Work	x1y	x2y	
2. bk1	x1y		x3y
3. bk2		x2y	a3y

Note: the variables in red are the ones that are found first in a databank search

```
time 2010 2012;
x1y = 1; x2y = 2;
open<edit>bk1; clear bk1; x1y = 10; x3y = 30; close bk1;
open<edit>bk2; clear bk2; x2y = 200; x3y = 300; close bk2;
open bk1, bk2;
prt <n> {'x*y'};
prt <n> {'*:x*y'};
prt <n> x1y, x2y, x3y;
```

Examples:

- `prt {'x*y'};` prints `Work:x1y`, `Work:x2y` (only variables from the Work bank)

- `prt {'*:x*y'};` prints `Work:x1y`, `Work:x2y`, `bk1:x1y`, `bk1:x3y`, `bk2:x2y`, `bk2:x3y` (all 6 variables in the table)

- `prt x1y, x2y, x3y;` prints the variables `Work:x1y`, `Work:x2y`, `bk1:x3y` (shown in red, provided that databank searching is active, else the statement fails regarding `x3y`).

While databank searching has advantages regarding concrete variables like `x1y`, `x2y`, `x3y`, using such a search logic regarding wildcards would be both confusing and error-prone.

Use of '%', '#', '!', and stars

In their most strict form, wildcards for bank searching are stated like this:

```
#m = ['x*y'];
```

This particular wildcard will return a list of strings containing the names that match the `'x*y'` wildcard, that is, names that start with 'x' and end with 'y'. This wildcard only matches variables from the first-position databank, with the current frequency. So if the first-position databank is `b1`, and the current frequency is annual (`!a`), the wildcard matches the same variables as `['b1:x*y!a']`. If you need to match all series of all frequencies (in all open databanks), you can use `['*:*!*']`. All scalars and collections are matched with `['*:%*]` and `['*:#*]`, respectively, so to match scalars or collections, you need to use `%` or `#` in the wildcard. However, to match all variables in a given databank, you may use the special `'**'` wildcard, so `['*:**']` matches all variables in all databanks.

The following finds all variables in all banks (as a list of string names):

```
#a = ['*:*!*'] + ['*:%*'] + ['*:#*']; //+ operator concatenates
#a = ['*:**'];                        //same: '**' matches all
variables in a bank
#a = ['***'];                         //same: '***' matches all
variables in all banks
```

Similarly, the following will match all items in the first-position databank:

```
#w = ['*!*'] + ['%*'] + ['#*'];
#w = ['**'];
```

whereas

```
#ws = ['*'];
```

matches all series with the same frequency as the current frequency in the first-position databank.

Bank ranges

Ranges work much like wildcards, using dots in a `['start' .. 'stop']`-range. For instance:

```
#az1 = ['xa'..'xz'];
```

will match all series of the current frequency in the alphabetical range `xa`-`xz` in the first-position databank. To match a range in another databank, use for instance:

```
#az2 = ['b1:xa'..'b1:xz'];
```

Note that you must state the bankname both before and after the dots.

List searching

You may use wildcards and ranges on lists of strings, for instance:

```
#m = xa, xay, xdy, xey;  //or: #m = ('xa', 'xay', 'xdy', 'xey');
#m1 = #m['x*y'];         //matches 'xay', 'xdy', 'xey'
#m2 = #m['xa'..'xe'];    //matches 'xa', 'xay', 'xdy'
```

When used on lists, wildcards and ranges work normally, that is, there are no special rules regarding bank colon `:`, frequency `!` or type symbols (`%` and `#`). The strings in the list are matched as they are.

Details: why the special logic?

The reader may wonder why wildcard search in databanks has a special kind of logic regarding symbols `%`, `#`, and `!`? This is explained below.

Imagine a databank containing these variables:

- `fy!a`, an annual series
- `fy!q`, a quarterly series
- `%y`, a string
- `#y`, a list

If we use 'naive' wildcards without special rules, we get this (for instance with `INDEX *;`):

```
INDEX *;  // --> fy!a, fy!q, %y, #y
```

Everything is matched. This may seem ok, but then what about this:

```
INDEX f*y;  // --> nothing
```

Here, the user may wonder why nothing is matched, but this is because of the frequency symbols `!`. If, instead, the search pattern ended on a star:

```
INDEX f*;  // --> fy!a, fy!q
```

Suddently the two series match again, because the star matches `!a` and `!q`. But if the star is first, we get:

```
INDEX *y;  // --> %y, #y
```

Now `!a` and `!q` are not matched, but on the contrary, the star matches `%` and `#`, so the string and list are matched.

The reader might object that one could just end the wildcard with `!*`, and the timeseries would be matched as expected. But the user has become accustomed to not having to write frequency indicators on timeseries of the same frequency as the global frequency. This is one of the advantages of Gekko, being able to write `prt fy;` and imply `fy!a` (if the global frequency is annual), so there would be the risk of users forgetting about frequencies when using wildcards (especially if they work in the same frequency most of the time).

Therefore, in Gekko 3.0, the `!`, `%` and `#` symbols are treated in a special manner when matching wildcards. In Gekko 3.0, the following is the case (withe the same four variables as above):

```
INDEX *;  // --> fy!a
```

Only the active frequency is matched (we assume it is annual). No starting `%` or `#` are matched.

```
INDEX *!*;  // --> fy!a, fy!q
```

Above, all frequencies are matched.

```
INDEX %*;  // --> %y
```

This is how to match scalars. Collections match with `#*`.

The rationale behind these rules is that much wildcard search takes places regarding series of a given frequency, and it is therefore beneficial that such wildcard search works as expected. The users would want to be able to write for instance `prt {'f*'};` or `prt {'*y'};` without worrying about frequency indicators and scalars/collection types.

Instead of the tedious `['*!*'] + ['%*'] + ['#*']`, matching all series, scalars and collections in a bank, `['**']` is offered as a shortcut to match all variables in a databank. In the same vein, `['***']` is a shortcut to `['*:**']`, matching all variables in all databanks, that is, 'everything'.

To match all series of all frequencies and all scalars in the first-position databank, you may use `**`:

```
INDEX **;  // --> All variables i first-position databank
INDEX ***; // --> Same, but would also match variables in other databanks
```

Note

The form `{'a*b'}` is actually short for `{['a*b']}`. In the last version, the inside of {} is seen to be a list of strings which is converted into a list of names, just like `{#m}` converts a list of strings `#m` into a list of names. For example, `#m = ['a*b']; prt {#m};` illustrates this, where `prt #m;` would just print the list itself, not the variables referred to by the list elements. Therefore, `prt {['a*b']}` prints the variables, and as noted, Gekko allows `prt {'a*b'};` as short for `{['a*b']}`.

4.1.5 Naked list

Naked lists are used to avoid unnecessary typing of parentheses and single quotes, for lists of strings or numbers. Naked lists can only be used on the right-hand side in assignments (typically list or series definitions), or on the right-hand side in FOR loop definitions. If a naked list contains only one element, it must contain a trailing comma (see the 'singletons' section below).

Naked list vs. normal list

- A **naked list** always appears on the right-hand side of an assignment statement, it never uses normal parentheses `(...)`, and it always contains at least one comma `,`. Examples: `#m = a, b, c;` or `#m += 1, 2, 3;`. Quotes are omitted for strings.
- A **normal list** can appear anywhere in an expression, it is always enclosed in normal parentheses `(...)`, and it contains at least one comma `,`. Examples: `#m = ('a', 'b', 'c');` or `#m += (1, 2, 3);`. Quotes must be used for strings.

Naked lists are naked in the sense that the normal list definition parentheses `(...)` are omitted, and strings inside the naked list are stated without single quotes. Therefore, the strict list definition `#m = ('a', 'b', 'c');` may be replaced by the 'naked' `#m = a, b, c;`, saving a bit of typing. A "naked" list of values can be used to construct either a list of values or a timeseries, cf. `#m = 1, 2, 3;` or `x = 1, 2, 3;`, where `#m` is a list, and `x` is a timeseries.

Curlies in naked lists

In a naked list definition, if a list `#m` is present inside {}-curlies, it is the *elements* of `#m` that are added, not the list `#m` itself. For instance:

```
#m1 = b, c;
#m2 = a, {#m1}, d;   //result: 'a', 'b', 'c', 'd'
```

So `#m2` does *not* become a nested list `('a', ('b', 'c'), 'd')`.

Example: naked list for strings:

```
#m = a, b, c;                //same as #m = ('a', 'b', 'c');
for string %i = a, b, c;   //same as ... = ('a', 'b', 'c')
  tell %i;
end;
```

Example: naked list for values:

```
y <2010 2012> = 1, 2, 3; //same as ... = (1, 2, 3)
#m = 1, 2, 3;              //same as #m = (1, 2, 3);
```

The following elements are legal in naked lists:

- Normal names like `a1`, including underscore character (`_`).
- Normal names with bank, frequency and index, like `b:a!q[i,j]`.
- Names/words starting with a digit, like `1a` or `1e5`.
- Normal integers like `123`
- Integers starting with zero, like `007`
- Floating point values like `1.2` or `1.2e5`
- Any character(s) may be replaced by {}-curlies, for instance `a{%s}b`, where the inside of {} may be any mathematical expression.
- If a list is present inside {}-curlies, the list items are added one by one, and characters may be prefixed or suffixed. For instance: `#m1 = b, c; #m2 = a, {#m1}, d;` will create the list `'a', 'b', 'c', 'd'`, whereas `#m1 = b, c; #m2 = a, x{#m1}y, d;` will create the list `'a', 'xby', 'xcy', 'd'`. (This will also work if the list items are numbers, but the primary use is for strings).
- Any element may use a prepended with a minus (`-`), for instance `-a1` or `-123` or `-1.23`.
- Any element may be repeated with `rep`, for instance `1, 2 rep 3, 3` or `1, 2, 3 rep *` (`rep *` is ignored for non-timeseries variables).
- Missings: use `m()` or `miss()`.

Special rules:

- If all elements are either normal integers or contain a decimal point (`.`), the list becomes a list of values. The elements may contain a minus sign (`-`). For instance, `1, 2` or `1.2, 3` or `1, 1.2e5` all become lists of values. But beware that a list like `12, 02` will become a list of the strings `'12', '02'`, and a list like `12, 1e5` becomes a list of the strings `'12', '1e5'`. The reason for this is stated below.
- An integer starting with `0` (except the integer `0` itself) is not interpreted as a value in a naked list. So for instance, `01` is not interpreted as the value 1, and `007` not as the value 7.
- An element composed of integers + `e`/`E` + integers will be interpreted as a string, for instance `1e5` is interpreted as the string `'1e5'`, not the value `100000`. On the contrary, `1.0e5` is interpreted as `100000`, not `'1.0e5'`.
- A list of integers like `1, 2, 3` will become a list of the values `1, 2, 3`, not the strings `'1', '2', '3'`. But you may easily and without loss transform such a list of

integers into the list of strings `'1', '2', '3'` via the `strings()` function. For instance: `#m = 1, 2, 3; #m = #m.strings();`.

- Single-element naked lists can be defined with trailing comma, for instance `#m = a,;`. See the 'singletons' section below.

The reason for the above rules is that naked lists are often used to define *codes*, for instance sequences of 3-character words consisting of alphanumerical characters, such as `ab7, 7dy, 638, 02e, 058, 1e5`. These are all three-character codes, and may represent, for instance, commodity codes for a large number of commodities. Regarding the last two codes, it would be unfortunate if a naked list consisting of `058, 1e5` was understood as a list of the values `58, 100000`, because then it could not be converted back into the list of strings `'058', '1e5'` via the strings() function. This could lead to subtle hard-to-find bugs. So in a sense, the naked list logic is loss-less. It may spring a surprise that for instance the naked list `483, 582, 3b5` becomes a list of strings, whereas the naked list `483, 582, 385`
becomes a list of integers. But these integers can be converted back into corresponding strings without loss or alteration of any kind.

To sum up, if you are dealing with lists of *codes*, you do not need to worry about some codes losing leading zeroes, or some codes being interpreted as mathematical exponents. If your list of codes contain digits only (without leading zeroes), it becomes a list of values, and Gekko will abort with a type error, if you try to use the elements as strings. In that case, you can just convert them into strings with the strings() function.

Singletons

Beware that single-element lists (singletons) are special. The following will not work:

```
#m1 = a;            //error
#m2 = 100;          //error
```

In that case, you can use a trailing comma to indicate that you are defining a list.

```
#m1 = a,;           //or: ('a',) or list('a')
#m2 = 100,;         //or: (100,) or list(100)
```

The use of trailing comma for singleton lists is ugly, but necessary for the clarity of the syntax (the user can always use the comma to spot whether a right-hand side is a list or not). Such an "ugly" trailing comma is not unheard of in programming languages, cf. for instance 1-element tuples in Python.

Conclusion

Regarding naked lists, there are three important things to remember. (1) If an element is an integer with leading superfluous zeroes (for instance `01` or `007`), all elements are interpreted as strings. (2) If an element is an integer followed by an `e` or `E` followed by an integer (for instance `1e5`), all elements are interpreted as strings. These rules are to avoid potential loss of or scrambling of information, for instance if the elements are *codes*. (3) If all elements are integers, these are transformed into values. Sometimes *codes* look like this, for instance `123, 234, 345`, but in that case, they can be transformed back into strings (without information loss) via the strings() function.

Note

Naked lists do not allow type symbols `%` or `#` (except if they are inside {}-curlies). This is to avoid confusion.

A naked list cannot contain elements with differing types, for instance `#m = a, 1.1;`. This will trigger an error, and again this is to avoid confusion.

4.1.6 Dynamic statements

Timeseries statements may contain accumulating lags like the following:

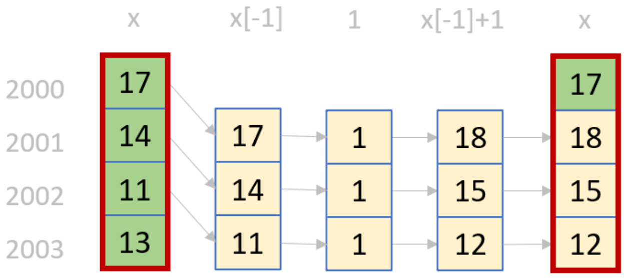
```
x = x[-1] + 1;
```

This basically says: take the value of `x` in the previous period, add 1, and assign the result back into `x`. If this is only done regarding one period, there is no ambiguity. But with several periods, how should such accumulation work?

The problem

In Gekko 3.0/3.1.x, such an assignment statement (containing the lagged dependent/endogenous variable on the right-hand side) will operate on blocks of data in a vector-like fashion. This is shown in the following diagram, running a series statement `x = x[-1] + 1` over the time period 2001-2003.

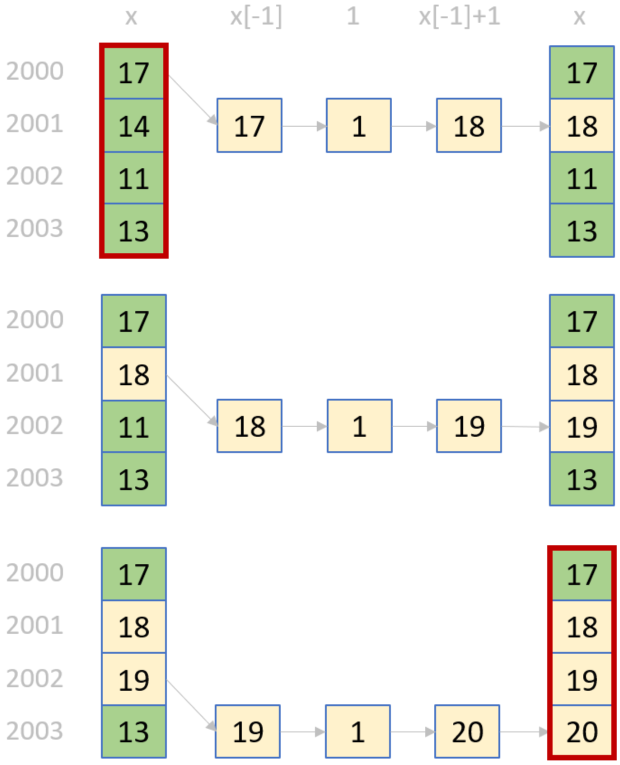
Figure 1: vector-like



Imagine the series `x` containing the values 17, 14, 11, and 13 over the period 2000-2003 (shown as red vector of values on the left). Now, when the statement `x = x[-1] + 1` is executed over the period 2001-2003, first an intermediate vector `x[-1]` is calculated, shifting the values of `x` one position downwards. This is shown as the yellow values 17, 14, and 11, and when 1 is added to these, we obtain the expression `x[-1] + 1` as the fourth vector of values (18, 15 and 12). These values are assigned back into the original series `x`, and the end result is shown as the red vector of values on the right. Except for the first period 2001, this does not accumulate as the user might expect; the expectation being that `x` augments by 1 for each of the years 2001, 2002, and 2003.

If instead we run the statement `x = x[-1] + 1` period by period, we get the following three tempi, corresponding to 2001, 2002, and 2003:

Figure 2: time looping



In the first period (2001), `x[-1]` is fetched as the value 17, `x[-1]+1` is therefore 18, and 18 is immediately fed back into the `x` vector. Therefore, when the second period (2002) is about to be calculated, the lagged value `x[-1]` is not 14 as in Figure 1, but instead the just calculated value 18. This goes on for 2003 too, and the end result (the red vector at the bottom right) is that `x` augments by 1 each year over the period 2001-2003. Prior to this calculation, only `x[2000]` has relevance regarding the result, since this period acts as a starting point.

As it should hopefully be clear, the way lags are handled in Figure 1 and 2 are different, and hence the results are also different in this particular case. But note that these differences only occur if there are more than one period calculated, and if the left-hand side variable `x` appears with lags on the right-hand side. Therefore, lags in an expression like `y = x[-1] + 1` or similar would yield the same result in Figures 1 and 2. Some timeseries-oriented software packages like for instance AREMOS or Gekko 2.x will accumulate as in Figure 2, where timeseries expressions are basically run period by period in an outer time loop, instead of operating on blocks/vectors of data.

There are many advantages regarding the calculation of timeseries expressions in a vector-like fashion, because it makes it clear exactly what the input and the output of a particular function or expression is, and there are no ambiguities as to whether some particular expression ought to be run only once (for instance a string or value expression), or should be calculated several times successively over a time period because timeseries are involved. Handling timeseries calculations in a vector-like fashion makes the flow of timeseries in and out of functions and procedures work in a much more transparent and encapsulated fashion. In addition to this, vector-like calculations run much faster than implicit time-looping, and the underlying Gekko source code becomes more simple, too.

Dynamizing with <dyn>, and the speed penalty

In Gekko 3.0/3.1.x, Gekko will handle series expressions in a vector-like fashion unless stated otherwise. So out of the box, dynamic statements with lagged dependent/endogenous variables will work as shown in Figure 1. However, it is possible to run a series expression with implicit time looping, corresponding to Figure 2. There are two equivalent ways to do this:

```
x <dyn> = x[-1] + 1;
```

Here, the `<dyn>` local option activates time looping. Run this way, you would get 18, 19 and 20 in the example in the beginning of this page, where `x` is augmented by 1 year period. Alternatively the following yields the same:

```
block series dyn = yes;
  x = x[-1] + 1;
end;
```

The effect of using `block series dyn = yes` is exactly the same as using the `<dyn>` option. There does not exist a any `option series dyn = yes` to set time looping globally, because it is deemed important to make sure that this functionality is only used where it is really necessary, so `block series dyn = no` is intended for a block of series statements that all should be run dynamically, and where it would be tedious to put `<dyn>` tags on each of these statements. Unnecessary dynamic tags (`<dyn>` or `block series dyn = yes`) waste a lot of time, and therefore slows down the execution Gekko programs. This is important enough to warrant a bullet point:

- When running a series statement over *n* observations, unnecessarily using `<dyn>` or `block series dyn = yes` on this statement typically makes the statement take *n* times longer to evaluate than without dynamization. Therefore, if you are evaluating series statements over, say, a period like 1970-2020, you should expect these to run around 50 times slower, if these expressions are unnecessarily dynamized (like in `y <dyn> = x[-1] + 1;`, where `<dyn>` is completely unnecessary).

Such inefficiency would make Gekko seem sluggish, and therefore it is not possible to dynamize via a general option. In general, dynamization should only be used when necessary, and be omitted when not. Often dynamization is not necessary at all, cf. the next paragraph.

Alternative syntax

Before moving on, it is worth mentioning that quite a lot of accumulating behavior can be implemented using operators or left-hand side functions (cf. descriptions on the SERIES page). For instance, the following are all equivalent:

```
x <dyn> = x[-1] + 1;
x <d>= 1;
x ^= 1;
dif(x) = 1;
```

For simple accumulations (relative accumulation is possible, too, cf. SERIES), it is often easier to use operators or left-hand side functions (note that the three last expressions do not need `<dyn>` keyword). Also note that accumulations like `dif(x) = dif(x) + 1;` (adding 1 to the absolute growth) or `pch(x) = pch(x) + 1;` (adding 1%-point to the percentage growth) work fine out of the box, and should be used without `<dyn>` tags.

But for more complicated accumulations, or when the equations are taken from model equations, using `<dyn>` may be the best way to go.

Dynamics checking and dynamics errors

As a convenience for the users, Gekko versions starting from version 3.1.7 implement automatic checking for the presence of lagged dependent/endogenous variables in a series statement. Therefore, in Gekko >= 3.1.7, the following statement:

```
x = x[-1] + 1;
```

will fail with a 'dynamics error', explaining that the user must decorate the statement with one of the following:

- `<dyn>`
- `<dyn = no>`
- `block series dyn = yes`
- `block series dyn = no`

In other words, when a lag like `x[-1]` is present in a series statement, the user must take a stand on whether the statement is to be calculated in a vector-like fashion (like Figure 1 above), or as an implicit time loop (like Figure 2 above). The following conditions will trigger the dynamics error on an assignment statement (that is, a statement of the form `something = something;`).

- The right-hand side must evaluate to a series, and the time period must have > 1 observations.
- The left-hand side must be a series variable.
- The left-hand side series must be used with a lag on the right-hand side. Lags may be in the form `x[-1]`, `x[-2]`, etc., or alternatively formulated as `x.1`, or `lag(x, 1)`.
- Right-hand side functions that can be interpreted as containing implicit lags like `dif()`, `pch()`, `movavg()`, etc. will not trigger the dynamics error.
- Left-hand side functions like `dif()` or `pch()` do not trigger the dynamics error either (these functions are built to deal with dynamics in an accumulating way).

In conclusion, the dynamics error is triggered if the user forgets to deal with the existence of a 'normal' endogenous lagged variable on the right-hand side, and this is the most common error regarding dynamic statements.

If you prefer to switch off automatic dynamics checking, you may use `option series dyn check = no;`, and in this case a statement like `x = x[-1] + 1` will proceed without error (as in Gekko versions prior to 3.1.7), and will be evaluated like in Figure 1 above (vector-like).

Backwards incompatibility, or how to ignore

In Gekko 3.1.7 and above, automatic checking for the existence of lagged dependent/endogenous variables is implemented and switched on, since `option series dyn check = yes` is set as default value. This may break existing programs in the sense that programs written for Gekko versions lower than 3.1.7 may allow a 'naked' statement like `x = x[-1] + 1` (omitting `<dyn>` or `block series dyn`), whereas in Gekko 3.1.7 and above, a 'naked' dynamic statement like that will fail with a dynamics error.

Because dynamics checking is not backwards compatible, it is entirely possible that a system of Gekko program files runs without error in a Gekko version < 3.1.7, whereas running the same system on Gekko >= 3.1.7 results in dynamics errors. If this is the case, there are different possibilities:

- Ignore 1: The most radical fix for such problems is to put an `option series dyn check = no` in the beginning of the program file or system of program files. This way, the behavior of Gekko < 3.1.7 is emulated completely, and the errors should dissolve.
- Ignore 2: Less dramatic is to decorate the problematic statements one by one with `<dyn = no>`, which would emulate pre 3.1.7 behavior, too. When in a hurry, this could be a short term fix, but the statements ought to be marked for later investigation.

- Fix: Better than ignoring the problems, the user could try to fix them. This means taking a look at the statements and judging whether the lagged dependent/endogenous variable really should not accumulate (that is, work as in Figure 1), or -- more likely -- whether it is indeed an accumulating expression. In the former case, the user can just insert a `<dyn = no>`, after which the error would dissolve, and Gekko would replicate former results. In the latter case, the user should insert a `<dyn>`, which will dissolve the error, but not necessarily replicate former results. Not replicating is of course annoying, but most likely it will stem from dynamics being erroneously omitted from some series statements, and therefore the check might identify errors that otherwise might have gone unnoticed.

Conclusion

Dynamics checking in Gekko versions 3.1.7 and above is convenient in the sense that the user needs to worry less about forgetting to put `<dyn>` option on normal accumulating statements like `x = x[-1] +1`. This eliminates a rather large source of potential bugs.

On the other hand, dynamics checking may also be inconvenient, when upgrading from a Gekko version < 3.1.7 to a Gekko version >= 3.1.7. In that case, dynamics errors may be reported, and these errors may be real errors that have gone unnoticed hitherto. Fixing errors is always a good thing, so even if upgrading to a Gekko version >= 3.1.7 may be inconvenient because of sudden dynamics errors, getting the potential errors sorted out is still good.

Technical note

When issuing a `<dyn>` or `block series dyn = yes`, over some time period `%t1` to `%t2`, Gekko basically runs the expression first for the period `%t1` to `%t1`, then for `%t1+1` to `%t1+1`, and so on, ending up with `%t2` to `%t2`. As noted above, there is a speed penalty regarding this, but the change of time periods may also affect the way the expression is interpreted in some (probably rare) cases. For instance:

```
time 2000 2003;
x = 17, 14, 11, 13;
time 2001 2003;
#m = 1, 2, 3;
x <dyn = no> = x[-1] + #m;  //18, 16, 14 = 17+1, 14+2, 11+3
```

Note that omitting the `<dyn = no>` tag in Gekko versions >= 3.1.7 would fail with a dynamics error, because of the lag `x[-1]`. If the user instead wants this to accumulate, the following could be tried as an alternative to the last line:

```
x <dyn> = x[-1] + #m;  //error
```

Now, this line fails with an error stating that Gekko expects that the list has 1 element only, because Gekko runs the expression via implicit time looping in three tempi: 2001-2001, 2002-2002, and 2003-2003. Therefore, in cases like this, decorating with `<dyn>` or `block series dyn = yes` may fail because of the implicit time looping. Such cases are probably quite rare, and in this particular case, the remedy could be to use a series to store the three numbers 1, 2, and 3 instead:

```
time 2000 2003;
x = 17, 14, 11, 13;
time 2001 2003;
m = 1, 2, 3;
x <dyn> = x[-1] + m;  //18, 20, 23
```

For instance, for the period 2001-2001 in the implicit time loop, Gekko knows which value to fetch out of `m`, since `m` is a timeseries, not a list. The result is as expected, where `x` first augments with 1, then with 2, and finally with 3.

4.1.7 Missing values

Missing values can be stated using the `m()` function in Gekko. For instance:

```
time 2021 2023;
x = 1, m(), 3;
prt x;

//                         x
//   2021            1.0000
//   2022                 M
//   2023            3.0000
```

If, in the above example, you set `x[2020] = m();` and write a databank, the databank will be exactly the same as if you had not set the missing value for 2020. This is because, for timeseries, 'real' data starts from the first non-missing value, and ends with the last non-missing value. This 'real' data period is 2021-2023 in the above example, and setting `x[2020]` to missing does not change that. After running the above example, before 2021 and after 2023, the `x` series can be thought of as containg an infinite number of missing values, and setting a missing value before 2021 or after 2023 does not change that. To get the 'real' start and end period for a timeseries, you may use the functions `x.fromseries('datastart')` and `x.fromseries('dataend')`.

You may transform missing values and non-missing values into 1's and 0's using the isMiss() function.

```
time 2021 2026;
x = m(), 1, m(), 3, m(), 5;
prt <n> x, x.isMiss(), x.isMiss('all');
```

```
//                                                 x.
//                    x     x.isMiss()  isMiss('all')
//  2021              M              M         1.0000
//  2022         1.0000         0.0000         0.0000
//  2023              M         1.0000         1.0000
//  2024         3.0000         0.0000         0.0000
//  2025              M         1.0000         1.0000
//  2026         5.0000         0.0000         0.0000

prt x.ismiss('all').sumt();  //3
```

Without `'all'` argument, the isMiss() function only looks at the 'real' data period (2022-2026), whereas with `'all'` option, the 2021 value is identified as missing, too. The last PRT statement prints the sum of the third column above, identifying 3 missing values (including the 2021 value). This can be used to check for the existence of missing values, for instance`if (x.ismiss('all').sumt() > 0)`.

All this is relatively straightforward. Questions arise, however, when comparing missing values with other missing values.

```
reset; time 2001 2003;

x1 = 1, 2, 3;
x2 = 1, 2, 3;
y1 = 1, m(), 3;
y2 = 1, m(), 3;

if (x1 == x2); tell '1'; else; tell '0'; end; // 1
if (x1 <> x2); tell '1'; else; tell '0'; end; // 0
if (x1 <  x2); tell '1'; else; tell '0'; end; // 0
if (x1 <= x2); tell '1'; else; tell '0'; end; // 1
if (x1 >= x2); tell '1'; else; tell '0'; end; // 1
if (x1 >  x2); tell '1'; else; tell '0'; end; // 0

if (y1 == y2); tell '1'; else; tell '0'; end; // 1     (*)
if (y1 <> y2); tell '1'; else; tell '0'; end; // 0
if (y1 <  y2); tell '1'; else; tell '0'; end; // 0
if (y1 <= y2); tell '1'; else; tell '0'; end; // 0
if (y1 >= y2); tell '1'; else; tell '0'; end; // 0
if (y1 >  y2); tell '1'; else; tell '0'; end; // 0
```

The first six IF statements are hardly surprising. However, regarding `y1` and `y2`, it is seen that the `==` and `<>` operators treat the missing value in 2002 as if it was just some special number. However, this is only the case regarding `==` and `<>`, not the operators `<`, `<=`, `>=` or `>`. For these operators, comparing a missing value with a missing value always results in false (or `0`). Note that for an IF involving two timeseries to be true, the IF must be true for *all* observations (cf. IF).

Compatibility issues

In Gekko versions < 3.1.8, the IF statement marked with `(*)` would have returned 0 instead of 1, because the two missing values in 2002 would have been considered non-equal. This has been changed in Gekko versions >= 3.1.8, and if Gekko detects that an IF contains a series comparison with missing values, a warning will be issued. The old behavior can be emulated by using IF_OLD instead of IF, but be warned that IF_OLD may not live on forever, and it is therefore better to think about the problem and adapt to the 'new' IF logic regarding missing values. If you have been using the idiom `if(x == x); ... ; end;` to check for the existence of missing values in the series `x`, this will no longer work in Gekko >= 3.1.8, and you should replace it with `if(x.ismiss('all').sumt() > 0); ... ; end;`, where the meaning is also much more clear. A radical solution to the compatibility issue would be to set `option bugfix missing = no`, which would emulate Gekko < 3.1.8 completely regarding IF and series with missing values. Setting this option generally is not recommended, however.

Perhaps the following three steps could be sensible, if a warning is encountered.

- If in a hurry, set `option bugfix missing = no`. Afterwards, the programs should run as before. However, this is not a good long-term solution, and doesn't really fix the issue.
- When more time is available, remove `option bugfix missing = no` and identify the IF statements where there is a difference (when Gekko warns about the problem, it shows a list with the problematic if statements). In these statements, replace `if(...)` with `if_old(...)`, which should replicate the former results. This, too, is not a good long-term solution either.
- Finally, resolve the `IF_OLD(...)` statements one by one. This may imply using `if(x.ismiss('all').sumt() > 0)` instead of `if(x == x)` to test for missing values, or in other ways changing the program to handle the presence of missing values.

The compatibility issue does not affect scalar comparisons, so `if(%x1 == %x2)` or `if(y1[2002] == y2[2002])` behave in exactly the same way in Gekko < 3.1.8 and Gekko >= 3.1.8 (the latter of these IF's is true in all Gekko 3.1.x versions).

4.1.8 Filenames

Gekko accepts relative paths, relative to the Gekko working folder. Consider, for instance, that you have a Gekko program file `job.gcm` with the following READ-statement inside the `job.gcm` file:

```
read \banks\data.gbk;
```

Now, Gekko will add the sub-folder `\banks` to the Gekko working folder path. If the Gekko working folder is `C:\Projects\Model1`, the READ statement is translated into:

```
read c:\Projects\Model1\banks\data.gbk;
```

You may use strings to compose file paths and names:

```
%s1 = 'Projects';
%s2 = 'Model1';
%s3 = 'banks';
%s4 = 'data';
read c:\{%s1}\{%s2}\{%s3}\{%s4};
read 'c:\{%s1}\{%s2}\{%s3}\{%s4}';
```

The two READ statements are equivalent: you may always use a string as a filename. More on string in the section on the STRING statement. Paths must use the `\` (backslash) or `/` (forward slash), and it is recommended to begin a relative path with the `\` or `/` character for clarity. It may be omitted though: for instance `read banks\data;` is equivalent to `read \banks\data;`. Double dots (`..`) can be used to indicate a parent folder, for instance `\..\banks\bank.gbk`. Example using forward slashes:

```
read c:/Projects/Model1/banks/data.gbk;
```

Valid file names consist of alphanumeric characters or the `_` character. If the file name contains blanks or special characters (for instance the Danish `æ`, `ø` or `å`), you may enclose the file name in single quotes (`read 'last year.gbk';`).

See also the `root()` function that makes it easier to work with relative paths, making it possible to write for instance `read {root()}\banks\data.gbk;`., where the root is determined from the location of a special `root.ini` file.

Note that a path may "pass through" a zip file, like `read c:\Projects\Model1\banks.zip\data.gbk;`, where `data.gbk` is automatically extracted from the zip-file `banks.zip`. Such zip-file paths are only possible regarding the *reading* of different kinds of data files, and will not work for writing (repacking a file into a zip file).

At some point it may be preferable to add the sub-folder to the path of the executing Gekko program file, rather than to the Gekko working folder. Choosing between the two ways of interpreting relative path's is not completely obvious, however.

4.1.9 Help system

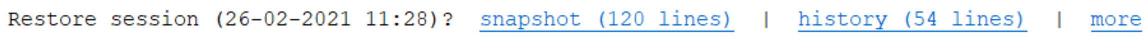
It may be difficult to remember all the statements and the exact syntax for each statement. The function key F1 (or typing `help;`) accesses the Gekko help system. If you cannot remember the exact syntax for a particular statement, you can try typing `help [commandname];`, for instance `help sim;` (or you may search the help files for particular phrases).

The help system is contained in a file `gekko.chm`. (Note: opening this file stand-alone from a network drive may sometimes pose problems on Windows, due to security reasons).

The help system is also available online here, and a pdf version is also available.

4.1.10 Restore session

Gekko remembers statements from the previous interactive Gekko session. After the Gekko application has been closed and reopened, a line like the following allows old statements to be reloaded into the current input window (the "more" link points to the current help system page):



```
Restore session (26-02-2021 11:28)? snapshot (120 lines)  |  history (54 lines)  |  more
```

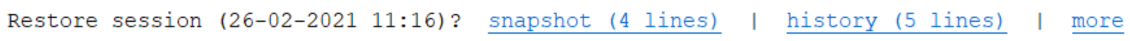
The statements are stored in two versions:

- **Snapshot**. The contents of the entire input window (the lower part of the main Gekko window) is stored as a snapshot.
- **History**. The issued Gekko statements are stored/recorded as a chronological list of statements.

The difference between "snapshot" and "history" can be illustrated in the following way (using a freshly opened Gekko):

1. In the input window, type `x = 2;` followed by `[Enter]`. (Gekko will add the `;` if omitted).
2. In line 2, type `y = 2 * x;` followed by `[Enter]`.
3. In line 3, type `x = 3;` followed by `[Enter]`.
4. Move the cursor two lines up to line 2 and issue the `y = 2 * x;` line again with `[Enter]`.
5. Move the cursor two lines down to line 4, and type `prt y;` followed by `[Enter]`.

The output window will show `y` with the value 6 (we will think of `x` = 3 and `y` = 6 as the current "state" of Gekko). Try closing and reopening the Gekko application. When starting up, you should now see something like the following:



```
Restore session (26-02-2021 11:16)? snapshot (4 lines)  |  history (5 lines)  |  more
```

If you click the "snapshot" link, the following lines will be put into the input window:

```
x = 2;
y = 2 * x;
x = 3;
prt y;
```

This is a snapshot of how the input window looked when ending the previous Gekko session. If you mark all the statements (`Ctrl+A`) and run them with `[Enter]`, the output window will show `y` with the value 4. So you cannot restore the exact previous state by issuing these snapshot statements in succession.

In contrast, the "history" link can restore the state. After clicking the "history" link, the following lines will be put into the input window:

```
x = 2;
y = 2 * x;
x = 3;
y = 2 * x;
prt y;
```

This is a chronological recording of the statements that were issued. If you mark all these statements (`Ctrl+A`) and run them with `[Enter]`, the output window will show `y` with the value 6. In that sense, the "history" link permits reestablishing the state of Gekko, as it was when ending the previous Gekko session. Obviously, the first two lines of the above history can be (and should be) deleted, so sometimes the history may be longer than really needed to recreate the state. This is normal when using Gekko in an interactive way.

When restoring, sometimes it may be most practical to just recreate the input window as a snapshot like it was at the end of the previous session, and sometimes the history may be more useful to recreate the state of the program.

Details

- Gekko stores snapshot and history in external files each time statement(s) are issued with `[Enter]` in the input window.
- If Gekko crashes badly or becomes unresponsive, you should still be able to restore snapshot/history.
- The history is recorded from when the Gekko application starts up. The history is cleared each time a RESET or RESTART statement is encountered, so that the history does not become unnecessarily long.
- When the snapshot or history link is clicked, existing code in the input window is first cleared.
- The statement history of the current session is generally available from the Gekko menu: Edit --> statement history... .

- The files that store the restore info are called `GekkoSnapshot.gcm` and `GekkoHistory.gcm`. The location of these can be found in the Gekko menu: choose Help --> About... --> and see the path under "Temporary files".

4.1.11 Under the hood

Language, licence etc.

Gekko is written in C#.NET, which together with VB.NET and Java are among the most used programming languages for pc's. Due to C# being object-oriented, development and redesign is flexible and efficient. The software is written for Windows .NET, so in order to run on Mac or Linux, the user has to use virtualization software. Gekko is open-source (public domain, GNU GPL licence), implying that anybody can use the code for free, but any enhancements must be put into the public domain as well.

Parser, structure etc.

The databanks and variables in Gekko are object-oriented internally. There can be any number of databanks, with any number of time series for any given periods (including quarters, months and other frequencies), only constrained by working memory. All values and calculations are double-precision (64-bit) internally, and missing values are handled consistently. Timeseries variables can contain labels, source etc., and the underlying data structures are dynamically resizing arrays, in order to keep the system fast. Models and command-scripts (Gekko program files, .gcm) are parsed and dynamically translated into C# code by means of ANTLR, providing fast and reliable parsing.

A model can be loaded dynamically without leaving Gekko. This means richer options for using different models at the same time, if needed. It also permits for instance optional fail-safe mode, where the model checks more strictly for illegal values while running (at a small speed penalty).

Solvers

At the moment, four algorithms for solving a model are provided.:

- First, standard Gauss-Seidel, where damping is supported via the formula codes. The program solves a large model like ADAM quite quickly with the Gauss-Seidel algorithm.
- In addition, a Newton method with line-search is implemented. This method does not depend upon the distinction between left- and right-hand side variables, and so can be used to solve difficult models or goals/means problems. The Newton solver uses a decomposition of the simultaneous block into a feedback set and the rest of the simultaneous block, reducing the dimension of the jacobian matrix

considerably. The Newton solver can handle any number of means/goals simply by changing the set of endogenous variables.
- The Fair-Taylor method ('fair') is used if the model contains leaded endogenous variables
- Newton Fair-Taylor ('nfair') is used for harder problems, using the Newton method to accelerate the Fair-Taylor iterations.

Graphics, tables

Graphics (PLOT) are done with gnuplot as the underlying engine. Gnuplot is installed together with the rest of the program. Printing and plotting uses the same syntax/options and underlying code. Graphs can be exported to Word via the clipboard, or saved to disk as for instance .emf or .svg files. Data tables can be exported directly to Excel, or via the clipboard to any spreadsheet software accepting tab-delimited input.

File formats and interfaces

Gekko databanks (.gbk) are zipped protobuffers, so the format is open, well-documented and easy to interface. Protobuffers are also used internally for caching models, so that they load faster.

Gekko 3.0 uses an internal Excel engine to read and write to Excel. This is fast and reliable, but only works for the newer .xlsx format. To read/write the older .xls files, an interface to Excel via COM Interop is possible, too.

The R interface is deliberately without COM Interop, but relies instead upon simple file exchange, and the gnuplot and X12A interfaces are similar.

Name

Why was Gekko called Gekko? One of the first versions, from early 2008, was called Echo. The intention was to find a suitable acronym afterwards, where 'ec' would be 'economic' or 'econometric'. However, Echo sounded a bit too much like the Danish shoemaker Ecco. Thus, the similarly sounding Gekko was chosen, partly because a gecko is a nice and helpful animal, and intentionally choosing the Danish spelling to distinguish it from, among other things, the Gecko browser engine. The intention was still to find a suitable acronym, with the 'e' being 'economic' or 'econometric', but the search for a suitable acronym is still ongoing. Gekko supposedly means something like 'moonlight' in Japanese (which gives better associations than, for instance, Gordon Gekko, who did not inspire the name).

4.2 Syntax basics

This section describes some of the syntax rules in Gekko 3.0, including the differences relative to the 'older' syntax of Gekko 2.0 (2.4) and earlier. For an easier and more hands-on introduction to the syntax and the most basic concepts, see the elevator pitch and the user guide section.

The section contains the following sections:

- Basic syntax rules. A section on syntax basics.
- More about syntax. More details about how the syntax works.
- Indexing: list, matrix, map. How indexing works for these variable types.
- Syntax diagrams. Diagrams the explain the basic components of the syntax.

4.2.1 Basic syntax rules

This section tries to explain some of the syntax basics.

See the page with syntax diagrams if the basics of names, expressions, etc. is confusing.

Basic syntax

Most statements start with a **statement name**, for instance PRT (for printing). You can see the statements sorted into categories here, or the alphabetical list of statements here. Beware that user-defined procedures may look similar to statements. Assignments like SERIES, VAL, LIST, etc. may omit the statement name.

Many statements accept an **option field** right after the statement name, for instance `prt <2015 2020>`. The option field always uses angle brackets <>, and is often used to state the local time period used in the particular statement. But many other options may be set, for instance `prt <p>` for percentage printing, or `prt <filter=avg>`. In `prt <filter = avg>`, the **option type** is `filter` and **option value** is `avg`, whereas in `prt <p>`, the option type is `p`, and the option should be understood as short-hand for `p = yes`. Many of the options are of yes/no-type (boolean).

After the option field, some **variables or expressions** are typically stated, like for instance `prt <2015 2020> x, y;`. In this case, the timeseries `x` and `y` are printed. To delimit elements, you typically use a **comma** (`,`), but blanks may also be used as delimiter.

All statements end with a **semicolon** (`;`), and the statements may span multiple lines. (If you need a multi-line statement in the user interface, use `[Ctrl+Enter]` to add newlines, and then mark the whole block and press `[Enter]`). Note that you may change the way `[Enter]` vs. `[Ctrl+Enter]` works, see `option interface edit style = ...` , or under `Edit --> Editor style` in the Gekko menu.

You may sometimes add **extra options** at the end of the statement, for instance `prt <2015 2020> x, y file = print.lst;`. Such extra options use the equal sign (`=`), similar to options in the <>-option field.

Gekko operates with seven types of variables: scalars (value, date or string), collections (list, map, matrix), or series. When referring to a scalar, you must use the `%`-symbol, for instance `%v`. When referring to a collection, you must use the `#`-symbol, for instance `#m`, whereas timeseries do not use prefix symbols. Using such symbols is helpful when reading expressions like `x + %y + #z[2]`, because `x` is known to be a timeseries, `%y` is known to be a scalar (probably a value, else the expression will fail), and `#z` is known to be a collection (from which the second item is selected, so in this case `#z` is probably a list). In Gekko 3.0, the prefix symbols must also be stated on the left-hand side of assigments like for instance `%v = 100`.

As anticipated above, you can use **[]-brackets** to select items. For timeseries, []-brackets can be used for lags/leads, for instance `gdp[-1]` or `gdp[+1]`, or for picking out an observation like `gdp[2015]` or `gdp[2015q3]`. For lists, []-brackets are used for selecting items in the list, for instance `#m[2]` or `#m[1..%n]`, and for maps, brackets are used to select elements by name (for instance `#m[d]` or the equivalent `#m['d']` or `#m.d`). Matrices use two dimensions, for instance `#a[2..3, 1..%n]`. You can use brackets on strings when selecting individual characters, for instance `%s[3]` or `%s[3..5]`.

Wildcards either use the `['...']` or `{'...'}` pattern or are 'naked'. For instance, `prt {'y*'};` will print all timeseries starting with `y`. Such wildcards can also be used with lists, for instance `#m['y*']`, selecting all elements starting with `y`. In some statements, the stand-alone brackets are not mandatory, for instance `index y*;` instead of the more tedious `index {'y*'};`. The reason why for instance `{'a*b'}` is used in `prt {'a*b'};` is that otherwise the expression `prt a*b;` would be ambiguous (does it mean the mathematical product of two timeseries, or is it a wildcard matching variables starting with `a` and ending with `b`?). See more on the wildcard page.

The **colon** (`:`) is used to access open databanks, for instance `prt bk7:pxa;`, where `bk7` is the databank, and `pxa` is the timeseries. When writing `prt pxa;`, the first-position databank is implicitly understood if databank searching is inactive, and if databank searching is active (which is default), Gekko will first look for `pxa` in the first-position databank (or in the Local databank if this contains variables), and afterwards in the other open databanks, ending with the `Global` databank (the `Ref` is never searched like this). You may use `prt bank2:pxa;` to obtain the values from the `bank2` databank. Alternatively, use the **at symbol** (`@`) to indicate the reference databank, for instance: `prt @pxa;`.

You may use **dot** (`.`) to indicate lags, for instance `prt pxa.1;` instead of `prt pxa[-1];`. Dots can also be used to select items from a MAP, for instance `#m.x` picks out element `x` in the map (alternatively, `#m[x]` or `#m['x']` can be used). Dots are also used for functions, for instance `%s.length()` for the length of a string `%s`.

Exclamation mark (`!`) is used to indicate frequency, for instance `prt pxa!q, pxa!m;` refers to the quarterly or monthly versions of the series `pxa`.

Strings should always be stated inside **single quotes** (`'`), for instance `%s = 'Hello from Gekko.';`. Double quotes (`"`) are not used to define strings in Gekko, but may be put inside Gekko strings (the string `'The name "Peter" har 5 characters'` is legal). If you need to insert a scalar or an expressions into a string, the most practical way is via `{}`-braces, for instance `'the {%s} car'`, where `%s = 'blue'`.

This is more readable than the alternative: `'the ' + %s + ' car'`. Note also that if `%s` is a string, there is the equivalence `'{%s}'` = `%s`.

{}-braces are also used for name-composition. For instance `prt px{%s};` will be equivalent to `prt pxa;`, if `%s = 'a'`. When reading Gekko 3.0 code containing {}-curlies, these curlies can be thought of as some sequence of characters, for instance `abc`. So if in doubt regarding the use of `{...}`, for instance whether some string `%s` must be put inside `{...}` or not, try to first consider whether the statement/expression would use a name like `abc`, or a string like `'abc'`? If you would use the former, you must correspondingly use `{%s}`, and if you would use the latter, you must correspondingly use `%s`. Note that there is the following equivalence: `abc = {'abc'}`, so in a way the {...}-curlies 'eat' the single quotes belonging to a string, and inside the `{...}`-curlies, you may put any expression, as long as it evaluates to a string. Another interpretation is that the {}-curlies perform a *forwarding* operation. If `%s = 'abc'`, the expression `{%s}` forwards from the variable `%s` to the variable `abc`. See also the syntax diagrams.

Functions use normal parentheses, for instance `movavg(x, 3)`. You may define your own functions (see here). All functions, both in-built and user-defined, implement so-called UFCS, so `movavg(x, 3)` can alternatively be written as `x.movavg(3)`, using a dot and putting the first argument on the left.

Power operators are either `**` or `^`, for instance `prt a**b;` or `prt a^b;`.

Logical operators use `<`, `<=`, `==`, `>=`, `>`, `<>`; note in particular that the equivalence operator is `==` and not `=`, see also IF.

$-conditionals can be used in the same way as in the GAMS software package. So you can write for instance `%x = 1 $ (%x < 0);` which sets `%x = 1` if `%x < 0`. This is equivalent to `if(%x < 0); %x = 1; end;`. The $-conditionals are often used in conjunction with lists, for instance `y[#i] = 100 $ (#i in #i1);` which sets the array-timeseries `y[#i]` to 100 for the elements of `#i` that are part of the subset `#i1`.

Variable names must start with `%`, `#`, a letter or an underscore, and are subsequently composed of letters, underscore or digits, for instance `f16`, `_temp`, `%f16`, `%_temp`, `#f16`, `#_temp`. Names may also contain `{}`-braces. Timeseries names starting with `xx` are often of temporary nature (see CREATE).

`//` and `/* ... */` are used for out-commenting lines of code, or blocks of code.

4.2.2 More about syntax

Below, some of the main concepts of the Gekko 3.0 syntax are explained in more detail. The descriptions in this section are quite advanced. So as mentioned before, for new users it is perhaps better to read the easier to understand Gekko guides, for instance starting out with reading the elevator pitch, and then moving on to the Data Management User Guide, perhaps followed by the Modeling User Guide.

See the page with syntax diagrams if the basics of names, expressions, etc. is confusing.

Banks, symbols, names, frequencies, indexes

A variable may be referred to in the following way:

```
bank + : + symbol + variable + ! + freq
```

First a databank name (optional), followed by a colon (`:`). Next an optional prefix symbol (`%` for scalars, `#` for collections, and no symbol for timeseries). The the variable name followed by an optional `!` and a frequency (only for timeseries). For instance, `b1:x!q` refers to the quarterly (`!q`) series `x` in the `b1` databank.

Array-series allow indexing, for instance `b1:x!q[a, b]` would refer to the sub-series `[a, b]` (that is, with two-dimensional indices `a`, `b`) of `x!q`. But other variable types allow indexing too, for instance `b1:#m[2, 3]`, picking out row 2, column 3 from the matrix `#m` residing in the databank `b1`.

If the databank is omitted on a variable in a statement or on the right-hand side of an expression, the following will take place (depending upon mode, cf. also the databank search page):

- sim-mode: If sim-mode is active, Gekko will look for the variable in the first-position or local/global databank.
- data- and mixed mode: Gekko will first look for the variable in the local databank, then in the first-position databank, then in subsequent open databanks, and finally in the global databank. Gekko will never search for a bank-less variable in the reference (`Ref`) databank.

In some cases, omitting the databank is silently interpreted as adding `first:` to the name, independent of mode settings. For instance `copy x to y;` is interpreted as `copy first:x to first:y;`, where `first:` refers to the first-position databank (often `Work`).

If the frequency is omitted for variables of series type, the current frequency will be silently added. So if the frequency is set to quarterly (`option freq q;`), you may use `x1` as short-hand for `x1!q`.

For array-series, the array indexes may sometimes be omitted, so that you may write `prt x;` instead of `prt x[#i, #j];`, printing out all the elements inside the array-series.

Names and quotes

In general, a string is enclosed in single quotes, for instance: `'x'`, whereas a name is not, for instance: `x`. Because of the use of prefix symbols in Gekko (`%` and `#` to start scalar and collection names), the single quotes may omitted in those cases where a series would not make sense. For instance, for array-series, using the shorter `x[a]` instead of the more strict `x['a']` is legal, because in the former variant it would not make sense to use an index with a series argument. Using `x[%a]` is another story, because `%a` could be a string, so the rule only applies to simple names without prefix symbols (that is, sequences of characters that are either alphanumeric or '_').

In the same manner, a lot of options accept string arguments, for instance `COMPARE <sort=rel>;`, where `rel` is the argument (relative sorting). It would not make sense for `rel` to be a timeseries, since a string is expected, and therefore the shorter `<sort=rel>` can be used as short-cut for the more strict `<sort='rel'>`. If the type needs to be controlled, you could use a string variable, so `%s = 'rel'; compare <sort=%s>;` would work fine. This is still work in progress.

Omitting single quotes is possible regarding list definitions and loops too, as seen in the following section.

Names, lists and loops

In general, lists are defined as comma-separated variables, enclosed in parentheses. For instance, `#m` may be a list of strings:

```
#m = ('a', 'b', 'c');   //strict
#m = a, b, c;           //naked list, NOT equal to (a, b, c)
```

As seen, a naked list variant is allowed, in the special case where all of the list elements are simple strings or simple values. Note that the syntax for a naked list of strings is always without parentheses in the list definition. The list `#m = a, b, c;` is interpreted as three strings `'a'`, `'b'`, `'c'`, whereas the list `#m = (a, b, c);` is different, containing three series variables (objects): `a`, `b`, `c`. The list `#m = 1, 2, 3;` becomes the three values `1`, `2`, and `3`.

The same goes for FOR, so the two following are equivalent.

```
for string %i = ('a', 'b', 'c'); prt {%i}; end;  //strict
for string %i = a, b, c; prt {%i}; end;            //naked
```

A one-element list (singleton) is special:

```
#m = a,;  //or: ('a',) or list('a')
```

The empty list is special too:

```
#m = list(); //note: using () may become legal later on
```

For a one-element list with string element `'a'`, you cannot use `#m = a;` or `#m = ('a');`. In the first case, the right-hand side is interpreted as a series (`a`), and assigning a series directly to a list will fail. In the second case the expression evaluates to `#m = 'a';`, assigning a string directly to a list (which will fail, too). Using a trailing comma like `#m = a,;` makes it a list.

You can read more on lists, in particular naked lists, here.

Indexes [...]

Regarding indexes of array-series or other variables, single quotes on a string can in general be omitted (both the following are valid):

```
prt x['a', 'b'];  //strict
prt x[a, b];      //short
```

Indexes are often used on lists to pick out items (so-called slicing), like `#m[3]` to pick out the third element of the list `#m`. For maps, you can also omit the single quotes like the following (all three are equivalent):

```
prt #m['a'];   //strict
prt #m[a];     //short
prt #m.a;      //dot syntax
```

Maps can be thought of as a kind of mini-databank. Note the similarity between for instance `b1:%x1` picking out the scalar `%x1` from the databank `b1`, and `#m1.%x1`, picking out the scalar `%x1` from the map `#m1`.

Name-substitution {...}

The {}-curlies are used for name-composition, and in general you may think of `{...}` as representing simply a sequence of (unquoted) characters, like `x22` or `y_15_sum`. When used, the inside of `{...}` must evaluate to a string or list of strings, for instance `{%s}` or `{#m}`, for instance:

```
%s = 'x';
#m = ('y', 'z');  //or: #m = y, x;
prt a{%s}, a{#m};
```

This is equivalent to `prt ax, ay, az;`. In a sense, `{...}` curlies eat/remove single quotes, so that `{'x'} = x`, transforming the string `'x'` into the variable/series `x`.

As seen, the {}-curlies can also be used together with other characters (or other curly braces), for instance `x{%i}a`. If `%i = 'e'`, this amounts to `xea`.

Often, instead of using array-series, normal series may be used to the same effect, using name-composition. So instead of the array-series `x[i1, j1]`, the user may use simply a series called `xi1j1`. If the list `#i` contains the string `'i1'` and the list `#j` contains the string `'j1'`, you may print the array-series `x[i1, i2]` like this: `prt x[#i, #j];`. Or with normal series and name-composition you can print the series `xi1j1` like this: `prt x{#i}{#j}`. There is a similarity between `x[#i, #j]` and `x{#i}{#j}`, and most of the time you can use both variants. Array-series, however, have more capabilities than name-composition.

Gekko 3.0 no longer allows omitting the `%`-symbols inside {}-curlies, so you cannot use for instance `x{i}a` instead of `x{%i}a`. Using `x%i|a` as synonym for `x{%i}a` is no longer endorsed in Gekko 3.0, but it still works.

See also the syntax diagrams.

More on indexes

Indexes can be used for:

- Array-series (mentioned above), for instance `x[a, b]` or `x['a', 'b']`. Integers may be used, if the dimension is compatible with an integer, for instance age dimension. When using integers, trailing zeroes are allowed, so for an array-series, `x[007]` is understood as `x['007']`, not `x['7']`.

- Lags/leads, for instance `x[-1]` or `x[+1]`. Note that a lag or lead must contain a `+` or `-` as the first character after the bracket. So if you define `%i = 2`, you may use `x[-%i]` or `x[+%i]`, but `x[%i]` will not work as a lag or lead (even if `%i` is negative). Instead, if `x` is a normal series, `x[%i]` will be understood as `x[2]`, which again is understood as the year 2 (two years after the birth of Christ). If `x` is instead an array-series, `x[%i]` will be understood as `x['2']` which could, for instance, represent 2-year olds (if `x` contains population data).

- Period reference: `x[2020q1]`, first quarter of 2020.

- Positions in lists: `#m[2]` picks out the second element of the list `#m`.

- Names in maps: `#m[a]` or `#m['a']` picks out the variable named `a` in the map `#m`. For simple names, `#m.a` can also be used (the variable `a` is a series).

- Matrix references (row/column), for instance `#m[2, 1]` picks out the numeric value in row 2, column 1.

- Searching: `#m['a*']` finds all elements matching the pattern `'a*'`.

- Note that in Gekko 3.0, you cannot use `#m[0]` to get the number of elements of the list `#m`. Use `length(#m)` or `#m.length()` instead.

- Ranges can be used for picking out elements, for instance `#m[2..4]` picks out elements 2 to 4 (inclusive), or `%s[2..4]` takes characters 2 to 4 from the string `%s`.

Syntax details

Some syntax from the 2.x Gekko versions has been deprecated, in order to clean up the syntax.

- Using `{i}` as short-hand for `{%i}` is no longer possible, for instance in a name like `x{i}` instead of `x{%i}`. First and foremost, using `i` instead of `%i` would go against the Gekko 3.0 principle that the type symbol is an integrated part of the name and hence cannot just be omitted. Next, a further problem with `{i}` is that any expression is allowed inside {}-braces, and this fact makes the treatment of `{i}` as `{%i}` somewhat confusing. For instance, consider this expression: `x{i[2020]}`. If, for instance, `i` is a series with value 100 in the period 2020, the name `x{i[2020]}` will be `y100`. Now, in contrast, the name `x{i}` will not try to use the series `i`, but will instead look for the scalar `%i`. So just removing the []-index from `i` means that `i` is suddenly understood as `%i`. Additionally, since using `x[a]` instead of `x['a']` is possible for array-series and in other indexes, the user may think that `x{a}` is short for `x{'a'}`, not `x{%a}`.

- Using `x%i` as short-hand for `x{%i}`, or `x%i|y` as short-hand for `x{%i}y` is no longer endorsed (but still works). There are several reasons for this non-endoresement. First, strings do not support this notation, so `'x%i'` will not have `%i` in-substituted, whereas `'x{%i}'` will (hence, for instance, `prt {'x%i'};` will not work, whereas `prt {'x{%i}'};` will. Because strings support `x{%i}` notation inside, it is easy to transform a name like `x{%i}` into the corresponding string; just add quotes: `'x{%i}'` (and vice versa). Second, the notation is illogical (or at least complicated). For instance, if `%i = 'a'`, we have in the strict {}-notation that `x{%i} = xa`. Here, we can easily prepend a prefix symbol `%`, for instance `%x{%i} = %xa`, and it is similarly easy to append a character, for instance `x{%i}b = xab`. And if we wish to omit the `x` and `b` we just toss them: `{%i} = a`. Now, with the short-hand notation it gets complicated. We have that `x%i = xa` which is fine. But if we prepend a type symbol, we have to use `%(x%i)`, otherwise Gekko will issue an error (`%x%i` is illegal). If we append a character, we have to use the concatenator: `x%i|a`, since `x%ia` will look for the scalar `%ia`. And if we want to loose the `x`, we have to use `{%i}`, since a naked `%i` returns a scalar string, not the series corresponding to this name. So to sum up, using the short notation entails cases where the user has to using adding parentheses, concatenator, or curly braces, which is error-prone, especially for less experienced users. Finally, there is readability. Whereas `x%i` is simple enough to read, how about `x%i|a%i%k|b` compared to `x{%i}a{%i}{%k}b`? Or `%(x%i|a)` compared to `%x{%i}a`? For these reasons, the `x%i|y` notation is no longer recommended in Gekko 3.0.

- Using `#m[%s]` as a logical condition is no longer possible. The idea is that `#m` could be a list of strings, and `#m['a']` could return 1 if `'a'` is a member of `#m`, and 0 otherwise. This syntax is used by GAMS, but the problem is that in Gekko 3.0, lists may contain values, so should `#m[3]` also mean a membership check (if the number 3 is one of the list elements)? But this syntax collides with `#m[3]` being used to fetch the element in position 3 in the list. Instead, in Gekko 3.0, the user can use `%s in #m`, or `#m.contains(%s)`. The `... in ...` syntax is used in quite a lot of programming languages, by the way.

- Omitting scalar or collection symbols on the left-hand side is no longer possible, for instance using `val v = 100;` or `list m = ('a', 'b', 'c');` is no longer legal. In Gekko 3.0, the `%` or `#` symbol is considered an integrated part of the variable name, as if these symbols were just special characters alongside `a`, `b`, `c`, etc. In order to comply with this logic, the symbols can never be omitted. Instead, the correct assignments are `val %v = 100;` or `list #m = ('a', 'b', 'c');`, but in Gekko

3.0 the types may be omitted, so `%v = 100;` or `#m = ('a', 'b', 'c');` is legal, too.

- List definitions are generally stated with parentheses, for instance `('a', 'b', 'c')`. But for convenience reasons, you may use a 'naked' list definition, for instance `#m = a, b, c;` to put the three strings `'a'`, `'b'`, `'c'` into the list `#m` or `y = 1, 2, 3;` to put the three values 1, 2, 3 into the series `y`. This also works in FOR loops and is convenient in many cases (remember that a naked list with only one element must have a trailing comma). For such naked lists, Gekko accepts elements composed of letters and digits (and some symbols like `_`, `-`, `:`, `!`, `[`, `]`), so `for string %i = 38, 007, 1e10, 2001q1;` is equivalent to `for string %i = ('38', '007', '1e10', '2001q1');`. See more about naked lists.

- Beware that the list definition `#m = (a, b, c);` is very different from the list definition `#m = a, b, c;`. The former finds the three timeseries `a`, `b`, and `c`, and puts them into the list as individual objects (of series type). The latter is a naked list (because it has no parentheses), and just inserts three simple strings into the list. In the former case, you may use `prt #m;` to print the series objects, whereas you must use `prt {#m};` in the latter case, if you want to refer to the variables corresponding to the string names. Using lists of strings to refer to variables is often more practical than using lists of series objects. As an example, you can use the syntax `prt bank1:{#m};` to print `bank1:a`, `bank1:b`, and `bank1:c` (that is, from the databank `bank1`), or the syntax `prt {#m}!q;` to print out the quarterly series `a!q`, `b!q`, and `c!q`. Gekko contains many inbuilt functions to handle such lists of variable names represented as strings.

- Concatenating and inserting strings. When combining (contatenating) variables into a string, there are generally two ways to do it. The first one is using the '+' operator, for instance `%s1 = 'blue'; %s2 = 'The ' + %s1 + ' car';`. The other way is to use {}-braces: `%s1 = 'blue'; %s2 = 'The {%s1} car';`. This is easier to read, and has another advantage. If `%s2` was for instance a value, the first variant would demand an explicit string conversion, for instance `%s2 = 'Number ' + string(%s1) + ' car';`, whereas this is not necessary regarding the last variant: `%s2 = 'Number {%s1} car';`. The reason for this is that the {}-braces already try to convert the inside into a string.

- Operators now include `=` symbol. In the Gekko 2.x versions, you could use for instance `series y % 3;`, to set the growth rate of `y` to 3% per period. In Gekko 3.0, you must use `y %= 3;`. Among other things, this has to do with the omission of the SERIES keyword in Gekko 3.0. It may seem that `y % x;` is reasonably clear, augmenting the series `y` with the percentages from the series `x`. But remove the blanks, and we have `y%x;`, amounting to the same, but this looks like a composed name `y%x`, floating in free air. To make sure that all assignments are recognizable, in Gekko 3.0 all assignments must include the `=` symbol, separating the left-hand and right-hand sides. Besides this, something like `x + 1;` just does not look like an assignment, adding 1 to `x`, whereas `x += 1;` feels much more understandable (and is, incidentally, prevalent in most modern computer languages).

4.2.3 Indexing: list, matrix, map

Gekko lists, matrices and maps are all containers of data, where the data is organized in some structure.

- A Gekko list is one-dimensional, but can be nested (lists inside lists), and may contain any Gekko variable type.
- A Gekko matrix is two-dimensional and can only contain values.
- A Gekko map is like a list where the elements are not ordered and hence not accessed by number index (for instance `#m[1]`, `#m[2]`, etc.), but instead by name (`#m['gdp']`, `#m['vat']`, etc.). In a map, the elements are not ordered sequentially, but instead strings are used to look up the elements. A Gekko map can be thought of as a mini-databank.

The strict syntax for defining a list uses soft parentheses `()`, for instance `(1, 2)` to define a two-element list. Note that a singleton list (with only one element) must use a trailing comma, for instance `(1,)`. The matrix equivalent would be `[1, 2]`, which is a 1 x 2 matrix (row vector), or alternatively `[1; 2]`, which would be a 2 x 1 matrix (column vector). A nested list could be stated like `((1, 2), (3, 4))`, which for a matrix would be `[1, 2; 3, 4]`, because `;` is used for matrices to separate rows. A list like `(1, 2)` has no awareness of being a row or a column or anything else; it is just a sequence of numbers that can be indexed by position.

```
#m1 = ((1, 2), (3, 4));
#m2 = [1, 2; 3, 4];
prt #m1, #m2;

//Result: --------------------------------------

#m1
(1, 2), (3, 4)

 #m2
                    1              2
     1         1.0000         2.0000
     2         3.0000         4.0000
```

Regarding the list `#m1`, it contains two sub-lists. Each of these sub-lists contains two values. So the list is nested, whereas the matrix `#m2` is organized in a two-dimensional structure of rows and columns.

In general, a nested list is indexed like for instance `#m1[2][1]`, picking out the value 3, and a matrix is indexed like `#m2[2, 1]`, also picking out the value 3. However, for nested lists of lists like `#m1`, Gekko allows the alternatively syntax `#m1[2, 1]`, too. So when selecting an individual element `%i`, `%j` in a nested list, there is no difference between `#m1[%i][%j]` and `#m1[%i, %j]`.

Things get more complicated when *ranges* are used:

```
//  1  2  3
//  4  5  6
//  7  8  9
// 10 11 12
#m = ((1, 2, 3), (4, 5, 6), (7, 8, 9), (10, 11, 12));
%v1 = #m[2, 3];       //6
%v2 = #m[2][3];       //6
#m1 = #m[2, 2..3];    //(5, 6)
#m2 = #m[2][2..3];    //(5, 6)
#m3 = #m[2..4, 2];    //(5, 8, 11)
#m4 = #m[2..4][2];    //(7, 8, 9)
#m5 = #m[2..4, 2..3]; //((5, 6), (8, 9), (11, 12))
#m6 = #m[2..4][2..3]; //((7, 8, 9), (10, 11, 12))
```

Here, `#m` is a four-element list, where each element is itself a three-element list. It can be represented visually as the 2d matrix shown in the comments, but beware that the nested list has no inherent notion of rows or columns. Both `#m1` and `#m2` amount to `(5, 6)`. In both cases, the second row is singled out, and elements 2-3 (inclusive) are selected from this. But `#m3` and `#m4` are different: the former selects rows 2-4 in column 2, which is `(5, 8, 11)`, whereas `#m4` evaluates to `(7, 8, 9)`. To understand `#m4`, we will split it up into `#x = #m[2..4]; #m4 = #x[2];`. Here, `#x` evaluates to `((4, 5, 6), (7, 8, 9), (10, 11, 12))` since it picks out elements 2-4 (inclusive) of the `#m` list. Next, from `#x`, the second element of this is selected, which is `(7, 8, 9)`. Perhaps not surprising, `#m5` and `#m6` are different, too. The former selects rows 2-4 and columns 2-3, resulting in the nested list `((5, 6), (8, 9), (11, 12))`, cutting out a part of the 2d matrix shown in the comments. In contrast, `#m6` evaluates to `((7, 8, 9), (10, 11, 12))` . We can reuse the `#x` temporary list again: `#x = #m[2..4]; #m6 = #x[2..3];`. So this time, `#x[2..3]` picks out elements 2-3 from `#x`, that is, `((7, 8, 9), (10, 11, 12))`.

To sum up, for nested lists of lists, Gekko allows the indexing syntax `[... , ... ]` in addition to the standard `[...][...]` indexing. When the first part of the former kind of indexing is a single value, there is no confusion. However, when the first part of such indexing is a range, the `[... , ...]` syntax selects elements in the same manner as matrix selection, whereas the `[...][...]` variant selects something altogether different.

The reason why nested lists allow `[... , ... ]` indexing syntax in Gekko is to make it possible to select elements in a similar manner to matrices, making it easier to use nested lists to represent for instance spreadsheet cells, tables or other 2d structures with mixed contents (for instance text, dates, and values). Another reason is to comply tightly with Python arrays (NumPy library), where such indexing is possible. Python also has a matrix library, but this is being deprecated in favor of using NumPy arrays instead (also for linear algebra calculations), among other things because arrays generalize naturally to n dimensions ("tensors"), in contrast to 2-dimensional matrices.

Arrays in Python

Since lists in Gekko follow most of the conventions of Python, the Python NumPy library also inspires some of the intricacies of multidimensional objects. First, we will have a look at the `ndarray` (n-dimensional array) variable type in Python.

```
import numpy as np
m = [[1, 2, 3],[4, 5, 6], [7, 8, 9], [10, 11, 12]]
a = np.array(m)
```

In the following Python code, `m` is a standard nested list, whereas `a` is an array. Here, `m[0]` will pick out the list `[1, 2, 3]`, and `a[0]` will pick out the array `[1, 2, 3]`, note that indexes are 0-based in Python. Both `m[0][0]`, `a[0][0]` and `a[0, 0]` will pick out 1, but `m[0, 0]` will fail with an error Selecting one of the row elements and a range of column elements (Python uses `:` instead of `..` for ranges) produces this:

```
m1 = m[1, 1:3] #type error
m2 = m[1][1:3] #[5, 6]
a1 = a[1, 1:3] #[5, 6]
a2 = a[1][1:3] #[5, 6]
```

Again, the list does not allow `[... , ...]` notation, but apart from this, everything is as expected (the Python range `1:3` means elements 2 and 3) . Now we try to select a range of rows and a fixed column:

```
m3 = m[1:4, 1] #type error
m4 = m[1:4][1] #[7, 8, 9]
a3 = a[1:4, 1] #[5, 8, 11]    <-- note!
a4 = a[1:4][1] #[7, 8, 9]
```

In this case, `m4` and `a4` still only obtain the second row, whereas `a3` obtains the second column (and `m3` fails with an error). Selecting several rows and columns at the same time:

```
m5 = m[1:4, 1:3] #type error
m6 = m[1:4][1:3] #[[7, 8, 9],[10, 11, 12]]
a5 = a[1:4, 1:3] #[[5,6],[8, 9],[11, 12]]    <-- note!
a6 = a[1:4][1:3] #[[7, 8, 9],[10, 11, 12]]
```

In this case, there is no difference, apart from the expected type error regarding `m5`.

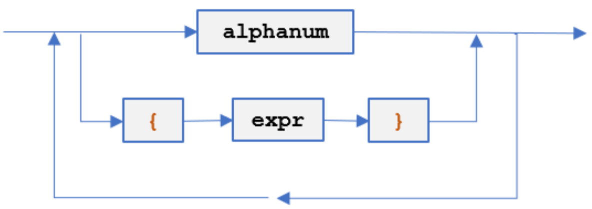
As it is seen above, the logic of ranges and nested lists in Gekko resembles the Python/NumPy/ndarray logic.

4.2.4 Syntax diagrams

Gekko 3.0 has a more strict syntax than Gekko 2.x and earlier. The following diagrams illustrate some of the fundamental building blocks of the syntax of 3.0. So whenever Gekko refuses one of your expressions, and the syntax error does not make sense, you may consult the following diagrams and perhaps understand the issue by means of these. The blue boxes below provide examples.

One of the most fundamental building blocks of Gekko is the `name`.

name (normal name)

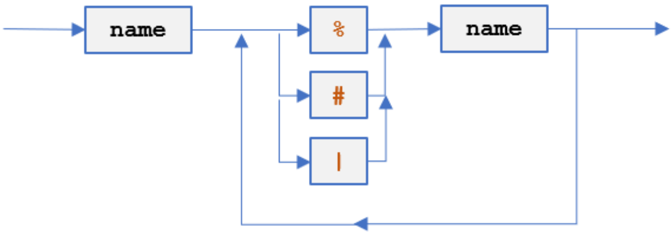


```
a
{%s}
a{%s}b
{%s1+%s2}b
a{#m}b
```

Here, `alphanum` means alphanumerical characters: letters, digits, and underscore, whereas `expr` is any legal Gekko expression (for instance a mathematical expression). Gekko will evaluate whatever is inside the {}-curlies, and will expect the inside to be a string or a list of strings. Note that `alphanum` excludes `%`, `#`, `!`, `:` and other symbols.

To make it possible to write for instance `x{%i}` shorter as `x%i`, a "complicated name" (`cname`) is introduced:

cname (complicated name)

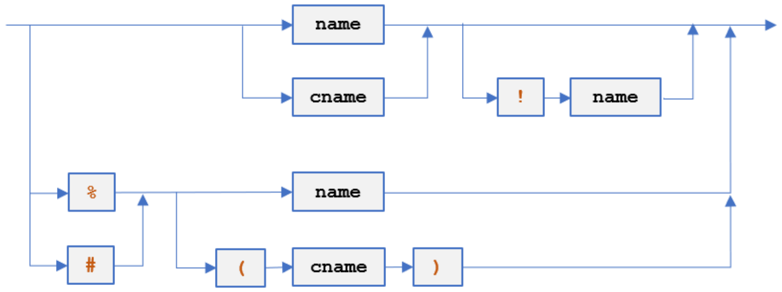


```
a%s|c
a#m
a%s1%s2
```

In many cases, such a `cname` can be used instead of a normal `name`. Note that the `name` part of the `cname` may contain {}-curlies, not just alphanumeric characters. The `cname` is mostly used to avoid typing too many {}-curlies, cf. the examples in the blue box. In Gekko program files, procedures and functions, it is recommended to use `name` instead of `cname`, for readability and maintainability. That is, it is recommended to use `x{%s}y` rather than `x%s|y`.

A variable name is a precise reference to an object residing in a particular Gekko databank. It may include prefix symbols `%` or `#` (for scalars or collections), or frequency `!` (for timeseries). The upper part of the diagram illustrates timeseries, which have no prefix symbols and may include frequency. The lower part of the diagram illustrates scalars and collections, starting with a type symbol.

varname (variable name)

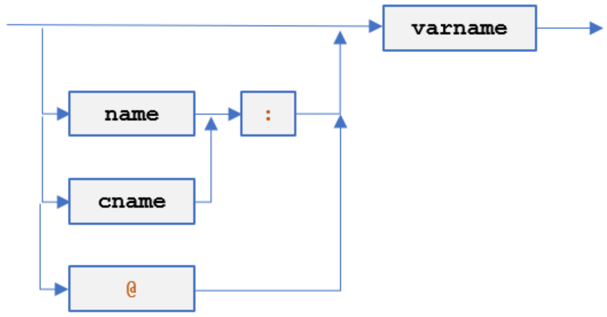


```
a
a%s
a!q
{%s1}!{%s2}
%s
#m
%{%s}
%(a%b)
```

Note that if you want to compose a scalar or collection name using a `cname`, you must use parentheses. For instance, `%(a%b)` designates a scalar name, where the name itself (excluding the `%`) is `a%b`. But in general it is much clearer to use the equivalent `name` version `%a{%b}` instead of the `cname` version `%(a%b)`. Note that `%a%b` is not legal syntax, since it would be too confusing.

A `varname` can reside in any databank (or MAP), and a `bankvarname` is hence designated as follows:

bankvarname (variable name with databank)

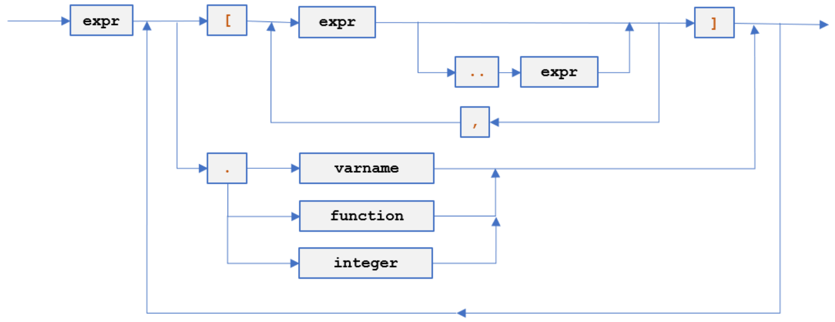


```
a
b:a
b:%s
b:#m
{%s1}:{%s2}
@x
@x!m
@%s
@#m
```

So either there is no bankname, else a colon (`:`) is used, or `@` can be used to imply the reference databank (`@x` should be read as short-hand for `ref:x`).

Indexing can be done with either []-brackets, or with a dot (`.`). You can use `..` to designate a sequence inside the []-brackets.

index (use of []-indexing)

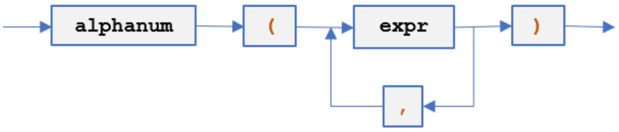


```
a[2020]
#m[1..2, 3..10]
(a + b)[2020q1]
a[’b’][2020]
#m.x
#m.f(’a’)
a.1
a[’b’].1
a!q[2020q1]
```

The dot (`.`) is used in three ways. The expression `#m.x` picks out the series `x` from the map `#m` (alternatively, `#m[x]` or `#m['x']` does the same thing). The expression `x.f(a)` is equivalent to `f(x, a)`, because Gekko implements UFCS. Finally, an expression like `x.1` is equivalent to `x[-1]`, that is, a one period lag.

The function syntax is completely standard:

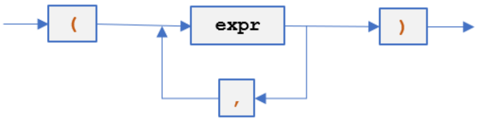
function (call of function)



```
f(100)
f(’a’, 100)
f(1+2, 3+4)
```

Lists are defined in the following way (this is a strict list definintion using soft parentheses; note that a naked list definition without parentheses is possible, too):

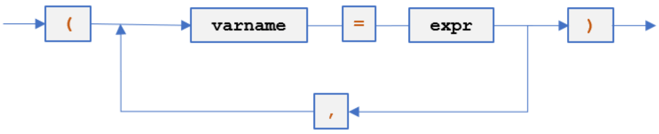
list (list definition)



```
NOTE1: ”(expr)” is not
considered a list, use
”(expr,)” for a one-element
list. In general, a trailing
comma is allowed. See also
naked lists.

(1, 2, 3)
(1, 2, 3,)
(2,)
(’a’, ’b’, ’c’)
(2020q3, 2020q4, 2021q1)
(1, ’a’, 2020q3)
(1+2, %s+’a’, 2020q3+2)
(x, %s, #m)
```

Maps are defined in the following way:

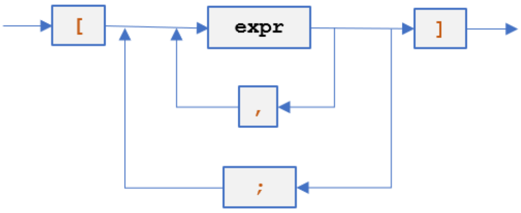
map (map definition)



```
NOTE: Dollar or indexer may be
used with varname. Type may also
be indicated, for instance (VAL %v
= 2). As for list, trailing comma
is allowed

(%s = 'a', %v = 2, %d = 2020)
(x = x1/x2, %v = x3[2020])
(#m1 = (1, 2), #m2 = [1, 2])
(series<2020 2021>x = (1, 2))
```

Matrices are defined in the following way:

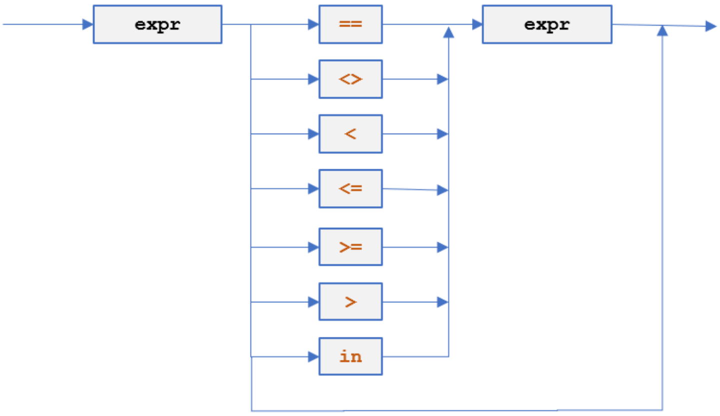
matrix (matrix definition)



```
[1, 2; 3, 4]
[1]
[1, 2, 3]
```

A logical statement:

logical (logical expression, eg. used in IF or with $-conditional)

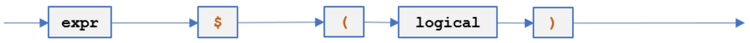


```
2 > 1
'a' == 'b'
'a' in #m
%x

The last one tests if %x is
0 or not. Logicals can be
combined with and, or, not,
for instance:

%x >= 10 and %x <= 20
```

The keyword `in` checks if the first `expr` is a member of the second `expr`.

Dollar-conditionals:

dollar (dollar conditions a la GAMS)



```
a $ (b > 100)
(a + b) $ (x[2020q1] == 100)
a $ ('a' in #m)
```

Note that parentheses are always used, and membership uses the `in` keyword. A GAMS expression like `x(i) $ i0(i)` is thus translated into `x[#i] $ (#i in #i0)`. Using `x[#i] $ #i0[#i]` or `x[#i] $ (#i0[#i])` will not work. See the "Details" section of this page for an explanation.

4.3 Gekko statements

This section describes in detail the purpose of the different Gekko statements including syntax, together with examples etc. Please select a statement on the menu at the left (you may need to fold out the menu tree first).

Regarding general syntax, the reader may consult the short section on this here, or the syntax diagrams here.

Apart from the statement sections, the section contains an overview:

- Statement overview. The statements are listed by category, and you may choose to see MODE-specific versions of the this list: sim-mode or data-mode. See the section 'Gekko commands' for an alphabetical list of statements, and the functions section to see functions.

4.3.1 Statements overview

Note: You may consult the specialized overviews regarding sim- and data-modes here:

- Sim-mode statements overview
- Data-mode statements overview.

Introduction

Below, all Gekko statements are listed, grouped together by functionality (regarding functions, see the section on these: 'Gekko functions').

Databanks

At startup, Gekko operates with two databanks; 'Work' (first-position, working bank) and 'Ref' (reference, baseline bank). There are the following statements related to databanks:

READ	Reads a databank file (typically gbk) into the first-position and reference databanks.
WRITE	Writes the first-position databank to a gbk file
IMPORT	Merges a databank file (typically non-gbk) into the first-position databank
EXPORT	Writes the first-position databank to a non-gbk file
OPEN	Opens a databank file (typically gbk). May use OPEN<edit> or OPEN<ref>.
CLOSE	Closes 'named' databanks (cf. OPEN)
CLONE	Makes the reference databank an exact copy of the first-position databank.
DOWNLOAD	Retrieves timeseries from a web-based database
COPY	Copies timeseries between banks (or inside the first-position databank)
RENAME	Renames timeseries.
INDEX	Uses wildcards to search for timeseries in databanks.
COUNT	Uses wildcards to count timeseries in databanks.
COMPARE	Finds differences between the first-position and reference databanks.
FINDMISSINGDATA	This statement finds timeseries with missing values.
HDG	Inserts a heading (description) into a gbk databank
UNLOCK	Sets a databank editable
LOCK	Sets a databank non-editable

Timeseries

Timeseries exist as objects in a databank. Frequency can be annual, quarterly, monthly, weekly, daily or undated.

TIME	Sets global time for timeseries operations.

TIMEFILTER	Omits or averages certain periods in output
CREATE	Create a new timeseries
DELETE	Delete an existing timeseries
SERIES	Transform a timeseries using mathematical expressions or data values
COLLAPSE	Convert e.g. quarterly timeseries into annual timeseries etc.
INTERPOLATE	Convert e.g. annual timeseries into quarterly timeseries etc.
SMOOTH	Fills in missing values in a timeseries
SPLICE	Splices two timeseries into one.
REBASE	Calculates an index series
TRUNCATE	Removes observations in a timeseries outside the stated sample.
ANALYZE	Computes cross-correlations etc.
DOC	Change meta information (label, source and date stamp)

Lists, scalars, matrices etc.

Gekko can put names of timeseries into a list, in order to reuse the list for different purposes (or make the program file easier to read). In addition, scalar variables like strings, dates and values can be used.

LIST	Create and delete lists
DATE	Scalar variable of date type
STRING	Scalar variable of string type
VAL	Scalar variable of value type
MATRIX	Define a matrix
MEM	Shows a list of scalar variables and their values

Show data

Gekko can show data in several ways, including printing on the screen, graphs, or showing the data in an Excel sheet. In addition, there is a special table-like decomposition window (DECOMP). The DISP statement also functions as an in-built equation browser. You may prefix a variable with '@' to indicate the reference ('Ref') databank, for instance @gdp. Or else use colon to indicate a databank, for instance mybank:gdp.

PRT	Prints timeseries or expressions in different ways
MULPRT	Prints multipliers: differences between the first-position and reference databanks.
DISP	Prints info regarding timeseries, and starts equation browser
PLOT	Show a graph of timeseries (using gnuplot)
SHEET	Like PRT, but shows timeseries data in Excel
CLIP	Like PRT, but puts timeseries data on the Windows clipboard
DECOMP	Decompose an equation
FIND	Find a variable in equations
TELL	Prints text strings on the screen

Model

A model can be loaded directly from a .frm file. After the model is loaded, a number of statements can be used:

MODEL	Load, parse and compile a model from file.
SIM	Simulates the model (also if there are goals/means)
ENDO	Endogenize variables (means)
EXO	Exogenize variables (goals)
UNFIX	Removes ENDO/EXO goals/means.
CHECKOFF	Skip convergence check for chosen variables (Gauss)
ITERSHOW	Show iterations in detail for chosen variable (Gauss)
SIGN	For signing models with signatures.

Gekko program files

Larger tasks can be run by means of program files (.gcm). There are the following statements related to such files:

RUN	Runs a .gcm program file. Use the EDIT statement to edit these files.
PIPE	Direct output to an external file instead of screen
INI	Runs gekko.ini if located in the program and/or working folder

Functions/procedures

You may use user-defined functions or procedures to avoid repetitive tasks and encapsulate functionality.

FUNCTION	Defines a user-defined function.
PROCEDURE	Defines a user-defined procedure.

Cleanup

The principal cleanup-statement is the following

RESTART	Clears all databanks, lists, scalars, models, etc. and runs any gekko.ini files.
RESET	Same as RESTART, but without running any gekko.ini files.
CLEAR	Clearing databanks
CLS	Clears main window (short for 'clear screen')
CUT	Closes all PLOT and DECOMP windows

Control flow

Gekko supports basic control flow like loops, conditional statements etc. At the moment the possibilities are quite limited, but will be augmented as the software matures.

FOR	For-loop over lists/strings, values or dates, parallel loops are possible.
IF	Conditional statement (IF-ELSE-END).
END	Ends loop (FOR), conditional statement (IF) or FUNCTION/PROCEDURE.
RETURN	Returns from the program file or function defintion.
STOP	Stops execution completely.
EXIT	Stops execution completely, and terminates Gekko.
ACCEPT	Input data interactively
PAUSE	Waiting for the user to click [Enter]
GOTO	Transfers execution to the corresponding TARGET
TARGET	Receives execution from the corresponding GOTO

Tables/menus

TABLE	Prints out a predefined table (xml)
MENU	Opens up a menu (html)

Econometrics

OLS	Single-equation linear regression

Python and R interfaces

PYTHON_RUN	Runs the decorated Python file, and returns matrices back to Gekko
R_RUN	Runs the decorated R file, and returns matrices back to Gekko

Miscellaneous

The following statements did not fall into the above categories, and so are gathered here:

MODE	Set Gekko mode to sim/data/mixed
HELP	Access the help system
OPTION	Sets different options
EDIT	Edit a file via Notepad
XEDIT	Edit a xml file via XML Notepad.
SYS	Access the system shell if needed
TRANSLATE	Translates syntax from Gekko 1.8 or AREMOS

From the menu items (‘Utilities’), you can also compare two databanks, check residuals, and compare variables in model/databank/varlist.

4.3.1.1 Sim-mode statements overview

> Note: You may consult the general overview regarding all statements here:
> - General statement overview

Introduction

Sim-mode (cf. MODE) is focused on solving models, comparing scenarios etc. Below, the different Gekko simulation related statements are listed, grouped together by functionality (regarding functions, see the section on these: 'Gekko functions'). The statements listed below are the core statements regarding model simulation.

Databanks

At startup, Gekko operates with two databanks; 'Work' (first-position, working bank) and 'Ref' (reference bank). There are the following statements related to databanks:

READ	Reads a databank file (typically gbk) into the first-position and reference databanks.
WRITE	Writes the first-position databank to a gbk file
IMPORT	Merges a databank file (typically non-gbk) into the first-position databank
EXPORT	Writes the first-position databank to a non-gbk file
CLONE	Makes the reference databank an exact copy of the first-position databank.
COMPARE	Finds differences between the first-position and reference databanks.
FINDMISSINGDATA	This statement finds timeseries with missing values.
HDG	Inserts a heading (description) into a gbk databank

Timeseries

Timeseries exist as objects in a databank. Frequency can be annual, quarterly, monthly, weekly, daily or undated.

TIME	Sets global time for timeseries operations.
TIMEFILTER	Omits or averages certain periods in output
CREATE	Create a new timeseries
DELETE	Delete an existing timeseries
SERIES	Transform a timeseries using mathematical expressions or data values

Lists, scalars, matrices etc.

Gekko can put names of timeseries into a list, in order to reuse the list for different purposes (or make the program file easier to read). In addition, scalar variables like strings, dates and values can be used.

LIST	Create and delete lists

DATE	Scalar variable of date type
STRING	Scalar variable of string type
VAL	Scalar variable of value type
MEM	Shows a list of scalar variables and their values

Show data

Gekko can show data in several ways, including printing on the screen, graphs, or showing the data in an Excel sheet. In addition, there is a special table-like decomposition window (DECOMP). The DISP statement also functions as an in-built equation browser. You may prefix a variable with '@' to indicate the reference (baseline) databank, for instance @gdp. Or else use colon to indicate a databank, for instance mybank:gdp.

PRT	Prints timeseries or expressions in different ways
MULPRT	Prints multipliers: differences between the first-position and reference databanks.
DISP	Prints info regarding timeseries, and starts equation browser
PLOT	Show a graph of timeseries (using gnuplot)
SHEET	Like PRT, but shows timeseries data in Excel
CLIP	Like PRT, but puts timeseries data on the Windows clipboard
DECOMP	Decompose an equation
FIND	Find a variable in equations
TELL	Prints text strings on the screen

Model

A model can be loaded directly from a .frm file. After the model is loaded, a number of statements can be used:

MODEL	Load, parse and compile a model from file.
SIM	Simulates the model (also if there are goals/means)
ENDO	Endogenize variables (means)
EXO	Exogenize variables (goals)
UNFIX	Removes ENDO/EXO goals/means.
CHECKOFF	Skip convergence check for chosen variables (Gauss)
ITERSHOW	Show iterations in detail for chosen variable (Gauss)
SIGN	For signing models with signatures.

Gekko program files

Larger tasks can be run by means of program files (.gcm). There are the following statements related to such files:

RUN	Runs a .gcm program file. Use the EDIT statement to edit these files.
PIPE	Direct output to an external file instead of screen
INI	Runs gekko.ini if located in the program and/or working folder

Cleanup

The principal cleanup-statement is the following

RESTART	Clears all databanks, lists, scalars, models, etc. and runs any gekko.ini files.
RESET	Same as RESTART, but without running any gekko.ini files.
CLEAR	Clearing databanks
CLS	Clears main window (short for 'clear screen')
CUT	Closes all PLOT or DECOMP windows

Control flow

Gekko supports basic control flow like loops, conditional statements etc. At the moment the possibilities are quite limited, but will be augmented as the software matures.

RETURN	Returns from the program file.
STOP	Stops execution completely.
EXIT	Stops execution completely, and terminates Gekko.
ACCEPT	Input data interactively
PAUSE	Waiting for the user to click [Enter]

Tables/menus

TABLE	Prints out a predefined table (xml)
MENU	Opens up a menu (html)

Miscellaneous

The following statements did not fall into the above categories, and so are gathered here:

MODE	Set Gekko mode to sim/data/mixed
HELP	Access the help system
OPTION	Sets different options
EDIT	Edit a file via Notepad
XEDIT	Edit a xml file via XML Notepad.
SYS	Access the system shell if needed
TRANSLATE	Translates syntax from Gekko 1.8 or AREMOS

From the menu items (‘Utilities’), you can also compare two databanks, check residuals, and compare variables in model/databank/varlist.

4.3.1.2 Data-mode statements overview

Note: You may consult the general overview regarding all statements here:

- General statement overview

Introduction

Data-mode (cf. MODE) is focused on databanks, handling of timeseries, data revision and similar purposes. Below, the different Gekko data related statements are listed, grouped together by functionality (regarding functions, see the section on these: 'Gekko functions'). The statements listed below are the core statements regarding data handling.

Databanks

At startup, Gekko operates with two databanks; 'Work' (first-position, working bank) and 'Ref' (reference bank). There are the following statements related to databanks:

IMPORT	Merges a databank file (typically non-gbk) into the first-position databank
EXPORT	Writes the first-position databank to a non-gbk file
OPEN	Opens a databank file (typically gbk). May use OPEN<edit> or OPEN<ref>.
CLOSE	Closes 'named' databanks (cf. OPEN)
DOWNLOAD	Retrieves timeseries from a web-based database
COPY	Copies timeseries between banks (or inside the first-position databank)
RENAME	Renames timeseries.
INDEX	Uses wildcards to search for timeseries in databanks.
COUNT	Uses wildcards to count timeseries in databanks.
UNLOCK	Sets a databank editable
LOCK	Sets a databank non-editable

Timeseries

Timeseries exist as objects in a databank. Frequency can be annual, quarterly, monthly, weekly, daily or undated.

TIME	Sets global time for timeseries operations.
DELETE	Delete an existing timeseries
SERIES	Transform a timeseries using mathematical expressions or data values
COLLAPSE	Convert e.g. quarterly timeseries into annual timeseries etc.
INTERPOLATE	Convert e.g. annual timeseries into quarterly timeseries etc.
SMOOTH	Fills in missing values in a timeseries
SPLICE	Splices two timeseries into one.
REBASE	Calculates an index series
TRUNCATE	Removes observations in a timeseries outside the stated sample.
ANALYZE	Computes cross-correlations etc.
DOC	Change meta information (label, source and date stamp)

Lists, scalars, matrices etc.

Gekko can put names of timeseries into a list, in order to reuse the list for different purposes (or make the program file easier to read). In addition, scalar variables like strings, dates and values can be used.

LIST	Create and delete lists
DATE	Scalar variable of date type
STRING	Scalar variable of string type
VAL	Scalar variable of value type
MATRIX	Define a matrix
MEM	Shows a list of scalar variables and their values

Show data

Gekko can show data in several ways, including printing on the screen, graphs, or showing the data in an Excel sheet. In addition, there is a special table-like decomposition window (DECOMP). The DISP statement also functions as an in-built equation browser. You may prefix a variable with '@' to indicate the reference databank, for instance @gdp. Or else use colon to indicate a databank, for instance mybank:gdp.

PRT	Prints timeseries or expressions in different ways
DISP	Prints info regarding timeseries, and starts equation browser
PLOT	Show a graph of timeseries (using gnuplot)
SHEET	Like PRT, but shows timeseries data in Excel
CLIP	Like PRT, but puts timeseries data on the Windows clipboard
TELL	Prints text strings on the screen

Gekko program files

Larger tasks can be run by means of program files (.gcm). There are the following statements related to such files:

RUN	Runs a .gcm program file. Use the EDIT statement to edit these files.
PIPE	Direct output to an external file instead of screen
INI	Runs gekko.ini if located in the program and/or working folder

Functions/procedures

You may use user-defined functions or procedures to avoid repetitive tasks and encapsulate functionality.

FUNCTION	Defines a user-defined function.
PROCEDURE	Defines a user-defined procedure.
LIBRARY	Store functions/procedures inside a library .zip file for general use.

Cleanup

The principal cleanup-statement is the following

RESTART	Clears all databanks, lists, scalars, models, etc. and runs any gekko.ini files.
RESET	Same as RESTART, but without running any gekko.ini files.
CLEAR	Clearing databanks
CLOSEALL	Closes all 'named' databanks, and clears Work and Ref.
CLS	Clears main window (short for 'clear screen')
CUT	Closes all PLOT or DECOMP windows

Control flow

Gekko supports basic control flow like loops, conditional statements etc. At the moment the possibilities are quite limited, but will be augmented as the software matures.

FOR	For-loop over lists/strings, values or dates, parallel loops are possible.
IF	Conditional statement (IF-ELSEIF-ELSE-END).
END	Ends loop (FOR), conditional statement (IF) or FUNCTION/PROCEDURE.
RETURN	Returns from the program file or function defintion.
STOP	Stops execution completely.
EXIT	Stops execution completely, and terminates Gekko.
ACCEPT	Input data interactively
PAUSE	Waiting for the user to click [Enter]
GOTO	Transfers execution to the corresponding TARGET
TARGET	Receives execution from the corresponding GOTO

Tables/menus

TABLE	Prints out a predefined table (xml)
MENU	Opens up a menu (html)

Econometrics

OLS	Single-equation linear regression

Python and R interfaces

PYTHON_RUN	Runs the decorated Python file, and returns matrices back to Gekko
R_RUN	Runs the decorated R file, and returns matrices back to Gekko

Miscellaneous

The following statements did not fall into the above categories, and so are gathered here:

MODE	Set Gekko mode to sim/data/mixed
HELP	Access the help system
OPTION	Sets different options
EDIT	Edit a file via Notepad
XEDIT	Edit a xml file via XML Notepad.
SYS	Access the system shell if needed
TRANSLATE	Translates syntax from Gekko 1.8 or AREMOS

From the menu items (‘Utilities’), you can also compare two databanks, check residuals, and compare variables in model/databank/varlist.

4.3.2 ACCEPT

ACCEPT is used to show a dialog box in Gekko, where a scalar value can be input. See also PAUSE.

Syntax

ACCEPT type variable message;

type	Choose between val, date or string. For string type, you do not need to enclose the input in quotes.
variable	The name of the variable
message	Text string to be displayed (please remember single quotes). You can use '\n' to insert a new line.

Examples

The statement may contain text inside single quotes:

```
accept string %n 'Variable name';
accept string %s 'Label';
accept date %d 'Date';
accept val %v 'Value';
create {%n}; //if it does not exist
doc {%n} label = %s;
{%n}[%d] = %v;
disp <%d-1 %d+1> {%n};
```

The four ACCEPT-input might be the following:

```
'Input variable name' --> vat
'Input label --> Value added tax
'Input date' --> 2016
'Input value' --> 0.25
```

This will create the series `vat`, with the label 'Value added tax', and the value 0.25 in 2016.

If you need to accept list items, you may accept them as a comma-separated string, and afterwards use the `split()` function to split the string into a list of strings.

To control program flow dynamically depending upon user input, you may use code like the following:

```
accept string %yesno 'Do you want to proceed? (y/n, default: n)';
if(%yesno == 'y');
  tell 'proceeding...';
  //program continues
else;
  tell 'aborting...';
end;
```

A dialog box is shown, and if the user just presses [Enter], `%yesno` will be an empty string, and the program does not proceed. The program only proceeds when the user inputs `y` or `Y` (Gekko string comparisons are case-insensitive).

Related statements

RETURN, STOP, EXIT, PAUSE

4.3.3 ANALYZE

ANALYZE calculates statistics on timeseries (mean, standard deviation, etc), including correlation coefficients between the variables.

For each variable (expression), Gekko prints out mean, standard deviation, and min and max values. In addition, cross-correlations are computed, and put into the matrix `#corr`.

Syntax

```
analyze <period> variables;
```

period	(Optional). Local period, for instance `2010 2020`, `2010q1 2020q4` or `%per1 %per2+1`.
variables	A list of variables (timeseries expressions)

- If no period is given inside the <...> angle brackets, the global period is used (cf. TIME).
- If a variable without databank indication is not found in the first-position databank, Gekko will look for it in other open databanks if databank search is active (cf. MODE).

Examples

Analyze the growth rate of the three variables x, y, z:

```
analyze <1980 2015> pch(x), pch(y), pch(z);
```

Note

The cross-correlations are computed as Pearson product-moment correlation coefficients.

If you square the cross-correlation matrix (`multiply(#corr, #corr)`), these squared values correspond to the R2 value you obtain by pairwise linear regression between the variables, for instance `ols x2 = x1;`.

Related statements

OLS

4.3.4 BLOCK

Four types of options
Gekko operates with four types of options: global options, general options, block-options and local options. **Global options** are read when Gekko launches and cannot be altered later on (they have the form `option global ...`). **General options** are stated with the OPTION command and apply until changed. **Block options** are stated with the BLOCK command and apply for a block of Gekko statements. **Local options** are stated inside the `<>` local option field and apply to that single statement only. See more here.

A BLOCK structure is used to set the time period and/or other options temporarily (block options). A block can for instance be used inside a function or procedure definition, where the time period, frequency or other options may be changed, but where these changes should be undone after leaving the function/procedure. A block could be used together with LOCAL variables to avoid changing the state of the program when calling a function/procedure.

Using a `block series dyn = yes; ... ; end;` is the only way to set the <dyn> option on several expressions at the same time. This is because `option series dyn` should only be used when really needed, that is, for expressions like `x = x[-1] + 1;` and similar. So using the option together with a BLOCK makes sure the option is turned off again.

Syntax

```
block option1, option2, ...;
  ...statements...
  ...statements...
  ...statements...
end;
```

option	Can be one of two kinds: • A time period, using the TIME keyword, for instance `block time 2020 2030; ... ; end;` This corresponds to `time 2020 2030;`. • An option setting (OPTION statement without the 'OPTION' keyword), for instance `block freq q; ... ; end;` to change the frequency, corresponding to `option freq q;`. The options (including time period) can be stated as a comma-separated list, for instance `block freq q, time 2020q1 2025q4, print width = 200;`. The BLOCK keyword must always be ended with an END.

Examples

The following is an example of nested blocks that set the time period

```
time 2001 2003;
block time 2011 2013;
  y1 = 100;             //y1: 2011-13
  block time 2021 2023;
    y2 = 100;           //y2: 2021-23
  end;
  y3 = 100;             //y3: 2011-13
  end;
y4 = 100;               //y4: 2001-2003
```

This is an example of setting two options for printing (corresponding to `option print fields ndec = 1; option print fields pdec = 1;`).

```
time 2001 2003;
y1 = 1.17; y1 <2002 2003> %= 1.27, 1.37;
block print fields ndec = 1, print fields pdec = 1;
  prt y1;  //printed with 1 decimal
end;
prt y1;    //printed with default decimals

//  Result:
//                   y1          %
//  2001            1.2          M
//  2002            1.2        1.3
//  2003            1.2        1.4
//
//                   y1          %
//  2001         1.1700          M
//  2002         1.1849       1.27
//  2003         1.2011       1.37
```

BLOCK can also be used to change frequency temporarily. The following example will create the quarterly series `y1!q` defined over 2001q3-2023q2. After the BLOCK, the time period will be back to annual 2021-23, therefore `y2!a` is defined over these three years.

```
reset;
time 2021 2023;
block freq = q, time 2021q3 2023q2;
  y1 = 100;
end;
y2 = 100;
prt <n> y1!q, y2;  //y2 same as y2!a, since we are back at annual
frequency;
```

Note

A designated time period no longer has to be the first item on a list of block options. [New in 3.1.4].

Related statements

LOCAL, OPTION, TIME

4.3.5 CHECKOFF

The statement puts variables on an ignore-list, so that they do not influence convergence using Gauss-Seidel iterations.

Syntax

```
checkoff ;
checkoff variables ;
checkoff ? ;
```

[empty]	If no variables are stated, i.e. a CHECKOFF without arguments, the list of non-checked variables is cleared.
variables	Variable names or list
?	Prints the list of currently ignored variables concerning convergence in Gauss-Seidel method.

Example

CHECKOFF accepts variable names or lists (including wildcards), for instance:

```
checkoff x;
checkoff {#m};  //where #m is a list of names (strings)
```

Currrently ignored variables can be seen with

```
checkoff ?;
```

There is no CHECKON statement. The CHECKOFF statement is non-additive (like the ENDO and EXO statements). To eliminate a CHECKOFF-variable, just remove it from the list given to the CHECKOFF statement. To clear the CHECKOFF-list, issue a CHECKOFF statement with no arguments. An alternative to this is setting "OPTION solve gauss conv ignorevars = no". In that case the list will be ignored.

Note

In order for this statement to work, "OPTION solve gauss conv ignorevars" must be set to 'yes' (which is its default value).

CHECKOFF is also the related to the ITERSHOW statement. Sometimes a particular variable, or a type of variables, may postpone the convergence of the Gauss-Seidel algorithm. To avoid that, such variables may be put on the CHECKOFF list, and they will be ignored regarding convergence check.

Related statements

SIM, OPTION, ITERSHOW

4.3.6 CLEAR

The CLEAR statement is used to clear databanks in memory (that is, delete all variables inside the databanks). If CLEAR is used without argument, the first-position and Reference databanks are cleared (but not, among other things, the Global or Local databanks). Se also CLOSEALL.

Syntax

```
clear ;
clear databank ;
clear <FIRST  REF> ;
```

databank	The name of the databank (click F2 to see the list of databanks -- note that the Ref databanks does not show up in the F2 window if it is empty).
FIRST	Clears the first-position databank
REF	Clears the reference databank

Examples (clearing databanks)

To clear a particular databank, use:

```
clear mybank;
```

In particular, you may clear the Work and/or Ref databanks like this:

```
clear work;
clear ref;
```

To clear both the first-position and reference databanks, use CLEAR without arguments:

```
clear;
```

Alternatively, there are these local options:

```
clear<first>;
```

Clears the first-position databank (which is often 'Work'), whereas

```
clear<ref>;
```

Clears the reference databank (which is always `Ref`).

Note

To delete individual variables, see the DELETE statement. To clear the entire workspace, see the RESET and RESTART statements.

Since user functions, procedures or models do not live in databanks, CLEAR does not clear these. Use RESET/RESTART to that end. Also, CLEAR without arguments does not clear the local or global databanks.

Related statements

RESET, RESTART, DELETE, CLOSE, CLOSEALL, CLS, CUT

4.3.7 CLIP

CLIP has the same syntax and functionality as SHEET, so please see this statement.

Instead of sending the result to Excel as SHEET does, CLIP sends the result to the clipboard. Thus, the cells can be pasted into any spreadsheet (or other applications) accepting tab-delimited cells from the clipboard. Formatting of the cells is lost in comparison with SHEET, but otherwise the cells are the same. The loss of formatting may even be considered a benefit in some cases, for instance when pasting cells into different locations in the same spreadsheet.

The functionality is very similar to the 'Copy' button in the Gekko user interface. This button copies the last PRT/MULPRT or table to the clipboard (as tab-delimited cells).

CLIP uses the same internal component as PRT, so regarding operators and other details, also see the PRT help page.

Syntax

Please see the SHEET statement regarding syntax.

Note

The decimal separator used when copying to the clipboard can be changed by means of the option shown below. (This option will also apply to the 'Copy' button).

Related options

OPTION interface clipboard decimalseparator = [comma|period].

Related statements

SHEET, PRT, PLOT

4.3.8 CLONE

The CLONE statement copies the first-position databank into the (cleared) reference databank. After this, all variables in the two banks are identical, and all MULPRT, PLOT<m>, COMPARE, etc. will show no differences.

Syntax

```
clone;
```

Example

You may use the CLONE statement in the following way:

```
model m;
read data;
time 2015 2050;
sim;
clone;
vat += 0.01;
sim;
mulprt gdp;
```

The CLONE statement makes sure that the first-position and reference databanks are identical after the model is simulated for the first time. Hence, the differences (the 'multiplier') regarding the two scenarios can be printed with MULPRT statement.

Note

The READ statement always creates the reference databank as an exact copy of the first-position databank after reading. You may use READ<first> or READ<ref> to read data into the first-position or reference databank exclusively. The READ statement is equivalent to READ<first> followed by CLONE.

Related statements

READ, OPEN, MODEL, MULPRT, SIM, DECOMP

4.3.9 CLOSE

The CLOSE statement is used to close databanks in memory.

If the contents of the databank have been altered, these changes are written back til the databank file. This is often used in combination with `open <edit> databank;`, where the changes are later on saved to disk after a `CLOSE databank;`.

Syntax

```
close < SAVE=... TRACE=... > databanks;
```

SAVE=	(Optional). With `close <save=no>`, Gekko will not write the databank to file, even if the databank contents has changed. See also `open <save=no>`.
TRACE=	(Optional). Use `yes` or `no`, `yes` is default. With `trace=no`, no data traces are written.
databanks	The databank(s) to be closed. A star (`*`) indicates all open databanks opened by means of the OPEN statement (cf. also CLOSEALL). You may provide a list of banks like `close db1, db2;`

Example

Use this syntax to close a databank:

```
close mybank;
```

Closes databank 'mybank' (that has been opened by means of `open mybank;` and writes any changes to the databank back to the databank file).

```
close *;
```

Closes all databanks opened by means of the OPEN statement (and writes any changes to the databanks back to their databank files). After this, the Work databank will be in first position (Work cannot be closed).

To clear up the workspace, you may use:

```
close *; clear;
closeall;          //equivalent
```

If you issue `close *; clear;`, all opened databanks will be closed, Work will be in first position, and both Work and Ref are cleared (there is a shorter alias for these two statements, namely CLOSEALL). To reset the workspace entirely, see RESET/RESTART.

Close of a list of particular databanks (separate with commas):

```
close db1, db2;
```

Note

CLOSE cannot close Work or Ref databanks, and neither the Local or Global databanks. See the closely related OPEN statement.

Related statements

OPEN, CLEAR, CLOSEALL, DELETE

4.3.10 CLOSEALL

The CLOSEALL statement closes all open databanks, and clears the Work and Ref databanks. The statement is exactly the same as issuing `close *; clear;`.

Contrary to RESET/RESTART, a CLOSEALL does not clear the Global or Local databanks, and options, models, libraries, global time period, etc. are not touched. So among other things, global settings stored in the Global databank will still be available after af CLOSEALL.

Syntax

```
closeall;
```

Example

The statement is simply:

```
closeall;
```

Note

In a sense, CLOSEALL is a less destructive version of RESET.

Related statements

CLEAR, CLOSE, DELETE, OPEN, RESET, RESTART

4.3.11 CLS

CLS clears the output window.

Syntax

```
cls;
```

Example

The RESTART (or RESET) statement will not clear the output window (but clears everything else in the workspace), so you may use CLS before (or after) your RESTART statement:

```
cls; cut; restart;
```

This clears the output window, closes any plot or decomp windows, and restarts Gekko.

Related statements

RESET, RESTART, DELETE, CLOSE, CLEAR, CUT

4.3.12 COLLAPSE

COLLAPSE transform (aggregates) higher-frequency timeseries to lower-frequency timeseries, for instance from quarterly to annual data. You may use INTERPOLATE to do the inverse transformation.

There are two things to note about the COLLAPSE statement. Firstly, the statement ignores the global time period, and a local time period cannot be set. Therefore, the left-hand side becomes a full collapsed mirror image of the data in the right-hand side timeseries. Secondly, if there are missing values in the high-frequency series, this will be reflected as missing values in the lower-frequency series. So if you need COLLAPSE to create for instance an average of the first three quarters of a year, you must first fill in the fourth quarter with some guessed data (for daily frequency, there are options to ignore/skip missing days when collapsing).

Instead of the COLLAPSE statement, you may alternatively use the similar `collapse()` function, for instance `x!q.collapse();` (see under functions). If you need to collapse a collection of Excel data "points" (any pairs of Excel dates + values), you may use IMPORT<collapse>.

Syntax

`collapse < MISSING = ... > lowfreq = highfreq method;`

lowfreq	Lower frequency timeseries. Frequency can be indicated with suffix `!a`, `!q`, `!m` or `!w`. Banknames may be used. Lists of names may be used, like for instance `{#m}`.
highfreq	Higher-frequencey timeseries. Frequency can be indicated with suffix `!q`, `!m`, `!w` or `!d`. Banknames may be used. Lists of names may be used, like for instance `{#m}`.
method	(Optional). Choose between: • `total`: The higher-freq observations are summed (default). • `avg`: The higher-freq observations are averaged. • `first`: The first higher-freq observation is used. • `last`: The last higher-freq observation is used. Note: default is `total`. You can alter the default with `option collapse method = ... ;` (cf. OPTION).
MISSING=	(Optional). Only relevant when collapsing from daily (`d`) data. You can set `collapse <missing=flex>` to avoid too many frustrations with single missing days.

	• `strict`: missing days will entail missing data in the collapsed series (default). • `flex`: missing days are ignored when collapsing. Note: default is `strict`. You can alter the default with `option collapse missing d = ... ;` (cf. OPTION).

- If a variable on the right-hand side of `=` is stated without databank, Gekko may look for it in the list of open databanks (if databank search is active, cf. MODE).
- Looping: with a list like for instance `#m = x1, x2;`, you may use `collapse {#m}!a = {#m}!q;` to collapse `x1!q` into `x1!a`, and `x2!q` into `x2!a`.

Example

Use this to convert frequency:

```
reset; time 2020 2021;
x!q = 1, 2, 3, 4, 5, 6, 7, 8;
collapse x!a = x!q; //10 and 26
prt x!a;
```

Since the method is `total` as default, this will create the annual timeseries `x!a` where each annual observation is the sum of the corresponding quarters in `x!q`. Instead, you may use `avg` like this:

```
collapse x!a = x!q avg; //will be 2.5 and 6.5 instead
```

With option `avg`, the quarters are averaged. If you have existing data in the `x!a` timeseries and only want to collapse a part of the `x!q` series into `x!a` (for instance from 2018-20), you may use a temporary variable like this:

```
reset; time 2017 2020;
x!a = 11, m(), m(), m();  //only data for 2017
x!q = 1, 2, 3, 4, 2, 3, 4, 5, 3, 4, 5, 6, 4, 5, 6, 7;
collapse temp!a = x!q;
x!a <2018 2020> = temp!a;
prt x!a;  //the 2017 value of 11 is preserved
```

In this case, it is easier to use the `collapse()` function, which is also convenient in general for printing/plotting etc. The `collapse()` and `interpolate()` functions (cf. INTERPOLATE) can be combined like the following example (calculating each quarter's share of all quarters):

```
reset; option freq q; time 2001 2002;
x1 = 1, 2, 3, 4, 5, 6, 7, 8;
x2 = x1/x1.collapse().interpolate();
prt x1, x2;  //x2 = 0.1, 0.2, 0.3, 0.4 and 0.19, 0.23, 0.27, 0.31
```

When collapsing daily timeseries, you may use `<missing=flex>` in order to ignore/skip missing days, cf. the following example.

```
reset; option freq d; time 2021m1d1 2021m1d10;
x!d = 1;
collapse x1!w = x!d;  //default is <missing=strict>
collapse <missing=flex> x2!w = x!d;
prt <2020w53 2021w1 n> x1!w, x2!w, x!d;

//                   x1!w           x2!w            x!d
//  2020w53              M         3.0000
//  m12d28                                             M
//  m12d29                                             M
//  m12d30                                             M
//  m12d31                                             M
//  m1d1                                          1.0000
//  m1d2                                          1.0000
//  m1d3                                          1.0000
//
//  2021w1          7.0000         7.0000
//  m1d4                                          1.0000
//  m1d5                                          1.0000
//  m1d6                                          1.0000
//  m1d7                                          1.0000
//  m1d8                                          1.0000
//  m1d9                                          1.0000
//  m1d10                                         1.0000
```

This produces data for `x1!w` and `x2!w` for the two weeks `2020w53` and `2021w1`, where only the latter week is collapsed from a full 7-day week of data (note that `2020w53` has four days in 2020 and three days in 2021). Therefore, in `2002w53`, `x1!w` has a missing value (because collapse is strict per default), whereas `x2!w` ignores the missing values and attains the value 3. (Weeks are defined and numbered following the ISO 8601 standard, where days around New Year may belong to week 52, week 53 or week 1. Active 'workdays' will be implemented, omitting for instance weekends and holidays. For now, see the getSpecialDay() function).

Note

If a frequency indicator is omitted, Gekko will use the current frequency.

You can also use PRT<collapse> to get similar transformations in prints (freqs `q` and `m` only), and for printing/plotting etc., the `collapse()` function is convenient.

See also IMPORT<collapse> regarding Excel spreadsheets with higher frequencies than daily, or data with irregular frequencies.

Related options

OPTION collapse method = total; [total|avg|first|last]
OPTION collapse missing d = strict; [strict|flex]

Related statements

INTERPOLATE, IMPORT, PRT, SMOOTH

4.3.13 COMPARE

COMPARE compares variables in the first-position and reference databanks. The comparison is only done for timeseries of the same frequency as the global frequency setting. The comparison is done over the given period (or the global period if a period is not provided), and the user may provide a list of variables that are checked (if no list is given, all variables are checked).

COMPARE will per default put the output in the file compare_databanks.txt (this filename can be changed). You may set thresholds regarding absolute or relative differences (options ABS, REL and PCH), and you may dump a list `#dif` with the different series names (cf. DUMP).

The COMPARE statement is similar to the menu item 'Utilities' --> 'Compare two databanks...' in the Gekko user interface.

Syntax

```
compare < period  ABS=...  DUMP  REL=...  SORT=...  PCH=... TYPE=...
 DUMP> variables  FILE=... ;
```

period	(Optional). Local period, for instance `2010 2020`, `2010q1 2020q4` or `%per1 %per2+1`.
ABS=	(Optional). Absolute differences smaller than the value are not shown, for instance `<abs = 150>`.
MISSIN G=	(Optional). Choose `m` (default) or `zero`. If `zero`, any missing values (NaN) will be treated as if they were = 0. [New in 3.1.15].
REL=	(Optional). Relative differences smaller than the value are not shown, for instance `<rel = 0.01>` equivalent to 1%. You may alternatively use PCH for the same purpose.
SORT=	(Optional). Choose between alpha (default), abs or rel. The first sorts alphabetically (which is default), the next sorts after absolute differences, and the last sorts after relative differences. The sorting and the use of ABS=, REL=, and PCH= are independent of each other.
PCH=	(Optional). Percentage differences smaller than the value are not shown, for instance `<pch = 1.0>` corresponding to 1%. You may alternatively use REL for the same purpose.
TYPE=	(Optional). Choose between `type=normal` (default) or `type=hist` (history). The latter computes relative changes differently. If `x` is a

	timeseries from the first-position databank and `@x` is a timeseries from the reference databank, `type = normal` computes the relative difference as `rel = abs(x - @x) / @x`. In contrast, with `type=hist` the relative difference is computed as `rel = abs(x - @x) / ((abs(@x - @x[-1]) + abs(@x[-1] - @x[-2])) / 2)`, where the numerator is the same, but where the denominator is the average of absolute time-changes in the current period (`abs(@x - @x[-1])`) and the previous period (`abs(@x[-1] - @x[-2])`). So with `type=hist`, the variability of `@x` is used to indicate what a "large" difference is supposed to mean, and the denominator has some similarities with a standard deviation measure. The two methods will normally return different results for some variables. (For SIM Gauss-Seidel convergence check, a procedure almost exactly similar to `type=hist` is used for relative convergence checks).
DUMP	(Optional). If this option is set, a list `#dif` will be constructed, containing the list of different timeseries.
variables	A list of variable names. If no variables are given, the full databanks are compared. The names are separated by comma (like `x, y, z`), and a list `#x` of names should be used with {}-braces: `{#x}`. Regarding array-series, you may either indicate the name of the array-series itself (`x`), in which case all sub-series are checked, or you may state individual elements (like `x[a, k]`).
FILE=	Filenames may contain an absolute path like `c:\projects\gekko\bank.gbk`, or a relative path `\gekko\bank.gbk`. Filenames containing blanks and special characters should be put inside quotes. Regarding reading of files, files in libraries can be referred to with colon (for instance `lib1:bank.gbk`), and "zip paths" are allowed too (for instance `c:\projects\data.zip\bank.gbk`). See more on filenames here.

- If no period is given inside the <...> angle brackets, the global period is used (cf. TIME).

Example

Compare all variables for the global period, or a given period:

```
compare;  //global period
compare <2010 2020>;  //for this given period
```

Do the same, with a user-chosen filename:

```
compare <2010 2020> file=dif.txt;
```

Sort the result by relative differences:

```
compare <sort=rel>;
```

Only compare series names from the list `#x`:

```
#x = x1, x2, x3;
compare <2010 2020> {#x};
compare <2010 2020> x1, x2, x3;  //same as above
```

Do not show relative differences smaller than 0.02 (that is, 2%):

```
compare <2010 2020 rel=0.02>;
```

You may 'dump' a list `#dif` containing the names of the timeseries that are different:

```
compare <dump>;
plot <q> {#dif};  //plots the percentage differences
```

Array-series are supported, consider this example:

```
reset;
time 2001 2002;
xx = series(2);
xx[a, x] = 100, 100;
xx[b, x] = 200, 200;
xx[a, y] = 300, 300;
xx[b, y] = 400, 400;
yy = series(1);
yy[i] = 1000, 1000;
#m1 = a, b;
#m2 = list('a'); //the easiest way to state a 1-element list
clone;
xx[b, y] = 400.4, 402;
yy[i] = 1000.2, 1004;
yy[j] = 2000;
compare <dump sort = rel>;
plot <q> {#dif};
prt #dif; //print out the names of the different timeseries as a
flat list.
compare xx[b, y]; //comparing only this particular element.
```

The file compare_databanks.txt will contain the following output:

```
Comparing first-position and reference databanks

There are the following 5 series in both banks:
xx[a, x], xx[a, y], xx[b, x], xx[b, y], yy[i]

There are the following 1 series in the first-position databank, but not
in Ref databank:
yy[j]

There are the following 0 series in the Ref databank, but not in the
first-position databank:
[none]

Out of the 5 common series, there are differences regarding 2 of them:

xx[b, y]       WORK       REFERENCE             ABS DIFF      % DIFF
    max =     0.50
-------------------------------------------------------------------
2001       400.4000       400.0000                    0.4000        0.10
2002       402.0000       400.0000                    2.0000        0.50

yy[i]          WORK       REFERENCE             ABS DIFF      % DIFF
    max =     0.40
-------------------------------------------------------------------
2001      1000.2000      1000.0000                    0.2000        0.02
2002      1004.0000      1000.0000                    4.0000        0.40
```

At the right of each comparison, the value that is sorted after is shown ('max') -- largest differences are shown first. In this case, max = 0.50 means that the maximal percentage difference is 0.50% (in 2002) for the array-series `xx[b, y]`.

Note

Note: local option <rel> and <pch> cannot be used at the same time. If <abs> and <rel>/<pch> are used at the same time, series with differences less than the abs or rel/pch criterion are not shown.

This functionality was previously only accessible from the Gekko menu, but is now statement-driven.

Related statements

MULPRT, PRT

4.3.14 COPY

The statement is used to copy variables, either inside a databank, or between databanks.

Note that 'naked' wildcards are allowed in this statement, so you may for instance use `a*b?c` as wildcard instead of the more cumbersome `{'a*b?c'}`.

Syntax

```
copy < period  RESPECT  FROMBANK=... TOBANK=... ERROR=... PRINT >
 names1  TO  names2;
```

period	(Optional). Local period, for instance `2010 2020`, `2010q1 2020q4` or `%per1 %per2+1`.
RESPECT	(Optional). With this option, if no period is given, the global period is used. (This option can also be set as a general option with `option copy respect = yes;`).
FROMBANK=	(Optional). A databank name from where the list of timeseries are copied from.
TOBANK=	(Optional). A databank name to where the list of timeseries are copied to.
ERROR=	(Optional). With COPY<error=no>, Gekko will try to copy the items, but will not fail with an error if some of the items cannot be found.
PRINT	(Optional). With this option set, Gekko will print a list of which variables are copied to where, but without actually copying anything. The option can be practical for debugging.
names1	Variablename(s) or list(s) (wild-cards are allowed). You may prepend a databank name as bank:variable.
TO	(Optional). If the `to` part is omitted, the variables will be copied to the first-position databank (with the same names), and in that case, the COPY statement is implicitly interpreted as if it included a `... to first:*`, where `first` indicates the first-position databank.
names2	(Optional). A corresponding list with the new names. You may prepend a databank name as `bank:variable` (or use `bank:*` to keep the same names).

- If no period is given inside the <...> angle brackets, no time period is used.

- If a variable is stated without databank, the databank is assumed to be the first-position databank.
- Looping: with a list like for instance `#m = a, b;`, you may use `copy x{#m} to y{#m};` to copy `xa` to `ya`, and `xb` to `yb`.

If the RESPECT option is active, and the new name exists as a timeseries beforehand, it is only the observations inside the local time period that are copied into the existing timeseries (and not any meta-information like labels, etc.).

Examples

Inside the first-position databank

To copy items inside the first-position databank, consider the following examples:

```
reset;
a1 = 1; b1 = 2; c1 = 3;
copy a1 TO a2;
copy a1, b1, c1 TO a2, b2, c2;
#list1 = a1, b1, c1;
#list2 = a2, b2, c2;
copy {#list1} TO {#list2};       //note that "copy #list1 TO
#list2;" would copy the list itself
```

If you use the RESPECT option, only the observations inside the global time period are used. For instance:

```
copy <respect> a1 TO a2;
```

Else

```
copy <2010 2020> a1 TO a2;
```

will copy observations belonging to that particular period.

Note that a list inside {}-curlies auto-expands if there is a name part before of after the {}, so that the example could have been done like this instead:

```
reset;
#m = a, b, c;              //or: #m = ('a', 'b', 'c');
a1 = 1; b1 = 2; c1 = 3;
copy {#m}1 TO {#m}2;       //a1, b1, c1 to a2, b2, c2
```

From other databanks to the first-position databank

In these cases, you typically omit the TO keyword, if you are preserving the same names.

You may copy timeseries from other databanks (either the reference databank, or databanks opened with the OPEN statement), by using a colon:

```
copy mybank:a1, mybank:a2;
```

This will copy the two variables `a1` and `a2` from the databank `mybank` to the first-position databank (with the same names). For several items, using a list may be easier:

```
#m = a1, a2;
copy mybank:{#m};         //note that "copy mybank:#m;" will try to
find a list #m in mybank
```

where `#m` is a list with the timeseries names. Or alternatively, you may use the <from=...> option:

```
copy <frombank=mybank> a1, a2;       //this works too: copy
<frombank=mybank> {#m};
```

If you are copying from the reference databank into the first-position databank, you may use this:

```
copy @{#m};
```

Between arbitrary databanks

In this case, the `frombank=` and `tobank=` options can be practical, for instance:

```
copy <frombank=bank1 tobank=bank2> a1 TO a2;
```

This copies `bank1:a1` to `bank2:a2`. You may use lists instead of these names. This will do the same thing:

```
copy bank1:a1 TO bank2:a2;
```

Or with lists:

```
copy bank1:{#m1} TO bank2:{#m2};
```

where `#m1` is the list of names to be copied, and `#m2` is a list of the resulting names (that is, a renaming list). If the names are the same, you can just use `TO bank2:*`.

Wildcards and ranges can be used, for instance:

```
copy bank2:a* TO bank1:*;
copy bank2:a1..bank2:a5 TO bank1:*;
```

The first statement will copy all timeseries starting with `a` from `bank2` to `bank1` (you could have used `<frombank=... tobank=...>` as well to denote the databanks. The second line does the same thing, but only regarding the name range 'a1' to 'a5'.

Copying timeseries `a1` from databank `bank2` to the reference databank can be done with:

```
copy bank2:a1 TO @*;                //or copy bank2:a1 TO ref:*
```

Wildcards and ranges

It is often practical to use wildcards to copy items. You may for instance copy all the items starting with 'fx' from the open bank `mybank` to the first-position databank with this statement:

```
copy mybank1:fx*;
copy mybank1:f?a;                //single character wildcard
copy mybank1:pxa..mybank1:pxqz;  //a range of names
```

You may copy an entire databank into the first-position databank like this:

```
copy mybank1:**; //double star matches all variable types and all
frequencies
```

If you for instance need to replace all the variables in the first-position databank with the variables in the reference databank, you may use this:

```
clear<first>;
copy @**;         //or "copy ref:**"
```

Regarding syntax rules of wildcards, see more in the INDEX section. See also the wildcards page.

Note

If you use the `from=` or `to=` options together with explicit databank indicators (colon), the explicit databank indicators will override the `from=` or `to=` options.

If preferred, you may use `copy ... AS ...` instead of `copy ... TO ...` .

Related options

OPTION copy respect = no; [yes|no]

Related statements

CLONE, RENAME, INDEX, DELETE

4.3.15 COUNT

The statement is used to search for variables in databanks, using wildcards.

The COUNT statement is essentially a compact INDEX statement without the output.

When counting timeseries, remember that default COUNT only counts series of the currently active frequency from the first-position databank. You may use `:` and `!` to augment, for instance `count *:*!*;` to count all timeseries of all frequencies from all databanks.

Note that 'naked' wildcards are allowed in this statement, so you may for instance use `a*b?c` as wildcard instead of the more cumbersome `{'a*b?c'}`.

Wildcard logic, including double and triple stars etc., is explained more generally on the wildcards page.

Syntax

count <BANK=... > type *wildcards* ;

BANK=	(Optional). A databank name indicating where the variables are to be located.
type	(Optional). Restrict the type of variables.
wildcard	The variables to be searched for. You may use banknames to indicate a particular bank, and you may separate the wildcards with commas. In general, wildcards are of the form a*x to find all variables starting with 'a' and ending with 'x', or a?x to match only one character.

- If a variable is stated without databank, the databank is assumed to be the first-position databank.

The following provides a list of all variables in all databanks:

```
count ***;                    //all variables in all banks
count *:**;                   //same as above
count *:%*, *:#*, *:*!*;      //same as above
```

Example

The following COUNT statement will look for timeseries beginning with 'f' in the first-position databank (and with the current frequency):

```
reset;
fa = 1; fb = 2; fc = 3;
count f*;  //result: 3
```

Note

See the INDEX statement for more examples.

If you use variable names without wildcards or ranges, an existence check is performed (count = 1 if it exists, 0 otherwise).

See also the second half of this page regarding wildcards, syntax, etc.

Related statements

LIST, INDEX

4.3.16 CUT

Closes any open PLOT or DECOMP windows.

This can also be done via the "Close all PLOT and DECOMP windows" button in the Gekko main window, which does the exact same thing.

When using this button, and if the Gekko main window is out of focus, you may have to click the button two times (the first time brings the Gekko main window back in focus).

Closing PLOT or DECOMP windows can also be done with the interactive menu Window --> Close --> ... .

Syntax

```
cut;
```

Related statements

RESET, RESTART, DELETE, CLOSE, CLEAR, CLS

4.3.17 CREATE

This statement creates a new series in the first-position databank. The series contains no data, but can be used afterwards.

If you use Gekko in a data revision setting, consider using `mode data;`, where options are set so that you avoid a lot of CREATE statements ("MODE data" will set "OPTION databank create auto = yes;", "OPTION databank search = yes;", and others).

Syntax

create *variables* ;
create ? ;

variables	Variablename(s) or list(s) (wild-cards is allowed)
?	Prints a list of all created variables

- If a variable is stated without databank, the databank is assumed to be the first-position databank.

The reason for CREATE in sim-mode is to avoid accidentally creating a new variable because of misspelling etc. Imagine a model with exogenous variable b_vat = 0.25. The user thinks that the variable name is just vat (which might be what the VAT was called in an older version of the model). Without mandatory CREATE, setting vat = 0.26 will just create a new series that has no relation to the model, and hence does not affect any endogenous variables. With mandatory CREATE, setting vat = 0.26 will result in an error, and the user will hopefully discover that the proper name is b_vat.

There is an exception to the create rule: names beginning with 'xx' can always be auto-created (useful for temporary series variables).

Examples

In sim-mode, variables cannot be created on the fly, for instance:

```
reset;
mode sim;
x = 100; //fails
```

Here, x cannot be auto-created. The following will work:

```
reset;
mode sim;
create x;
x = 100; //ok
```

Series beginning with 'xx' are always auto-created.

Related options

OPTION databank create auto = no; [yes|no]
OPTION databank create message = yes; [yes|no]

Related statements

SERIES, DELETE

4.3.18 DATE

The DATE statement is used to assign a date to a scalar variable of date type. Date names always start with the symbol `%`, like the other scalar types val and string. Using the DATE keyword is no longer mandatory in Gekko 3.0.

Dates are used in combination with series variables, setting the periods over which these are calculated, printed, etc. See also the TIME statement. Date values are written like `2020`, `2020q3`, `2020m3` or `2020m3d25`.

Syntax

```
%d = expression;
date %d = expression;
date ?; //print string scalars
```

It is no longer legal to use for instance `date d = 2020;`, omitting the '%'. As the right-hand side, quarterly, monthly and undated dates are supported with 'q', 'm', and 'u' indicators, for instance 2020q4 or 2020m12.

Normally, the DATE keyword can be omitted, if the right-hand side is a date like for instance 2020q4. But in the case `%d = 2020;`, `%d` will actually become a value. To avoid that, you can use `date %d = 2020;`, `%d = date(2020);`, or `%d = 2020a;` (2020a1 will work, too). In most cases, `%d = 2020;` should work fine though, since Gekko can auto-convert integers into annual dates.

There are a number of in-built date functions to compose and extract dates.

Date combining functions

Function name	*Description*	*Examples*

date(d, f, opt)
date(d, f)
	Converts the date d into a new date with frequency f (string). The optional option can be omitted, or be `'start'` or `'end'`. Using `'start'` or `'end'` is only relevant when converting from a lower to a higher frequency.
	Beware that week numbers are special around New Year.
		`%d = 2021q1;`
		`prt %d.date('a'); //2021`
		`prt %d.date('m'); //error!`
		`prt %d.date('m', 'start'); //2021m1`
		`prt %d.date('m', 'end');    //2021m3`
		`prt %d.date('w', 'start'); //2020w53 !!`
		`prt %d.date('w', 'end');    //2021w13`
		`//Note that the first week that fully`
		`//contains the first quarter of 2021 starts`
		`//in 2020!`

	Returns: date	

date(y, f, sub)
date(y, 'm', m, 'd', d)
	Constructs a new quarterly/monthly/weekly date from y (integer), frequency (string), and subperiod (integer). You may also construct a daily date with a similar syntax.
	
	Note: you may also use date(x), where x can be a value or a string, and Gekko will try to convert the argument into a date.
	
	Returns: date
		`%d = date(2020, 'q', 2);  //2020q2`
		`%d = date(2020, 'w', 42);  //2020w42`
		`%d = date(2020, 'm', 12, 'd', 24);  //2020m12d24`

fromExcelDate(v)	Converts an Excel date (the val `v`, counting the number of days since January 1, 1900) to a date with daily frequency. Returns: date (daily)	See examples regarding the toExcelDate() function.

getFreq(d)	Extracts the frequency of a date
	Returns: string
		`%d = 2020q2;`
		`prt %d.getfreq();  //'q'`

getDay(d)	Extracts the day number from a date. Will fail if the date is not daily.
	Returns: value (integer)
		`%d = 2020m3d25;`
		`prt %d.getday(); //25`

getSpecialDay(year, name)	Finds a special day by year and name, and returns its daily date. Mostly used for holidays. If you input a wrong name, Gekko will provide a link showing all possible special days. Current possible holiday names (English, and equivalent Danish names):
	
	New_Years_Day Nytaarsdag
		`prt 2021.getSpecialDay('Easter_Sunday'); //2021m4d4`
		`prt 2021.getSpecialDay('Paaskedag');      //same day`
		
		`for(val %year = 2021 to 2025);`
		`  %day = %year.getSpecialDay('Christmas_Eve');`

	Leap_Day Skuddag
	Maundy_Thursday Skaertorsdag
	Good_Friday Langfredag
	Easter_Sunday Paaskedag
	Easter_Monday Anden_paaskedag
	Labour_Day Foerste_maj
	General_Prayer_Day Store_bededag
	Ascension_Day Kristi_himmelfartsdag
	Whit_Sunday Pinsedag
	Whit_Monday Anden_pinsedag
	Constitution_Day Grundlovsdag
	Christmas_Eve Juleaften
	Christmas_Day Foerste_juledag
	Boxing_Day Anden_juledag
	New_Years_Eve Nytaarsaften
	
	Note that `getSpecialDay(%year, 'Leap_Day')` may return a null value, else it returns February 29 in leap years). The leap day is not a holiday, but is nevertheless kept in this list of special days. See example.
		`  %s = %day.getWeekday('en');`
		`  tell 'In {%year}, Christmas Eve is on a {%s}';`
		`end;`
		`//Result:`
		`//In 2021, Christmas Eve is on a Friday`
		`//In 2022, Christmas Eve is on a Saturday`
		`//In 2023, Christmas Eve is on a Sunday`
		`//In 2024, Christmas Eve is on a Tuesday`
		`//In 2025, Christmas Eve is on a Wednesday`
		
		`prt 2020.getSpecialDay('Leap_Day');`
		`prt 2021.getSpecialDay('Leap_Day');`
		
		`%y = 2020;`
		`if(not %y.getSpecialDay('Leap_Day').isNull());`
		`  //code to handle if %y is a leap year`
		`end;`

getMonth(d)
getMonth(d, lang)
	Extracts the month number from a date. More specific than getSubPer(), and will fail if the date is not monthly or daily.
	
	You may input a language (`'en'` = English, `'da'` = Danish), in which case a
		`%d = 2020m2;`
		`prt %d.getmonth();  //2`
		`prt %d.getmonth('en');  //'February'`
		`prt %d.getmonth('en').lower()[1..3];  //'feb'`

	string is returned. Returns: value (integer) or string	

getQuarter(d)	Extracts the quarter number from a date. More specific than getSubPer(), and will fail if the date is not quarterly.
	Returns: value (integer)
		`%d = 2020q2;`
		`prt %d.getquarter();   //2`

getSubPer(d)	Extracts the sub-period from a date (1 if annual or undated, the quarter if quarterly, the month if monthly or daily, and the week if weekly).
	Returns: value (integer)
		`%d = 2020q2;`
		`prt %d.getsubper();   //2`

getWeek(d)	Get the week number from a date of weekly frequency. Does not accept daily frequency as argument, but the conversion `date('w')` can be used as intermediary for this (see examples). Beware that the year may change when converting from daily to weekly frequency.
	Returns: value (integer)
	[New in 3.1.13]
		`%d = 2020w20;`
		`p %d.getWeek(); //20`
		`p %d.getYear(); //2020`
		
		`%d = 2019m12d31;`
		`p %d.getWeek(); //error!`
		`p %d.date('w').getWeek(); //1`
		`p %`
		`d`
		`.date('w').getYear(); //2020!`
		
		`%d = 2021m1d1;`
		`p %`
		`d.date('w').getWeek(); //53`
		`p %`
		`d`
		`.date('w').getYear(); //2020!`

getWeekday(d)
getWeekday(d, lang)
	Extracts the weekday number from a date. Will fail if the date is not daily. The numbers are as follows: Monday = 1, Tuesday = 2, Wednesday = 3, Thursday = 4, Friday = 5, Saturday = 6, Sunday = 7.
	
	You may input a language (`'en'` = English, `'da'` = Danish), in which case a string is returned.
		`%d = 2020m3d25;`
		`prt %d.getweekday();     //3`
		`prt %`
		`d`
		`.getweekday`
		`('en'); //Wednesday`
		`prt %`
		`d.getweekday('en').lower()`
		`[1..3]; //wed`
		`prt %`
		`d.getweekday('da'); //Onsdag`

	Returns: value (integer) or string. [New in 3.1.12]	

getYear(d)	Extracts the year from a date.
	Returns: value (integer)
		`%d = 2020q2;`
		`prt %d.getyear();  //2020`

max(d1, d2, ...)	Finds the largest of any number of dates. (`max()` can also be used with value arguments). Returns: date [New in 3.1.6]	`prt max(2002q1, 2001q4);`
min(d1, d2, ...)	Finds the smallest of any number of dates. (`min()` can also be used with value arguments). Returns: date [New in 3.1.6]	`prt min(2002q1, 2001q4);`

observations (d1, d2)	Counts the number of observations (periods), with both start and end date included. The date difference using `-` is always the same as the number of observations minus 1.	`prt observations(2020q2, 2023q3); //14`
		`prt 2023q3 - 2020q2;             //13`

toExcelDate(d)	Converts a daily date into an Excel date (counting the number of days since January 1, 1900). See also fromExcelDate(). Excel dates can be subtracted to obtain day spans. [New in 3.0.7]
	
	Returns: value.
		`%v1 = toExcelDate(2019m11d12);`
		`%v2 = toExcelDate(2019m12d3);`
		`prt %v1, %v2; //43781 and 43802`
		`prt %v2 - %v1; //21 days (span)`
		`%d = fromExcelDate(%v1 + 100);`
		`prt %d; //100 days from 2019m11d12`

truncate(d1, d2)	Finds overlap between two different time periods. The period d1 to d2 is compared with the global time period (if no local period is indicated), or with the local	`time 2010 2020;`
		`#m = truncate(2000, 2030);  //(2010, 2020)`
		`#m = truncate(2000, 2015);  //(2010, 2015)`
		`#m = truncate(2000, 2005);  //(null, null)`

	time period (if such a period is indicated in the <...> fields). Use a local time period if you need to find the overlap between two arbitrary time windows.
	
	Returns: a list of two elements, start and end date of the resulting period. If the two elements are both null, there is no overlap.
	
	[New in 3.1.6]
		`time 1980 1990;`
		`#m = truncate(<2010 2020>, 2000, 2030);  //(2010, 2020)`
		`#m = truncate(<2010 2020>, 2000, 2015);  //(2010, 2015)`
		`#m = truncate(<2010 2020>, 2000, 2005);  //(null, null)`
		`//`
		`// The overlapping period (z) can be visualized as`
		`// the overlap of these time windows/periods:`
		`//`
		
		`// period1     . x x x x x . . .`
		`// period2     . . . y y y y y .`
		`// truncate()  . . . z z z . . .`

Examples

Note that you may use expressions in the option field, when referring to dates. For instance (where %per1 and %per2 are two dates):

```
prt <%per1-2 %per1+1> fY;
```

You may wish to use dates to control the flow of your system of program files, centralizing the assignment of dates in one place.

```
global:%per1 = 2012;  //will actually become a value, not a date
global:%per2 = 2040;
read bank2;
<%per1 %per1> x2 += 1000;  //only 1 year
sim <%per1 %per2>;
mulprt <%per1-1 %per2> y2;
```

Note here the use of the Global databank for storing the two dates. The Global databank is unaffected by READ statements, and is practical for storage of general settings like such dates. Conversions are possible:

```
%s1 = '2010';  //string
%v1 = 2015;  //value
%d1 = date(%s1);
```

```
%d2 = date(%v1);
time %d1 %d2;
```

Note that in order to convert the string `%s1`, you need an explicit conversion with the date() function (on the contrary, the conversion from the value `%v1` is automatic). The conversion will fail if not possible, for instance the string '201x' or the val 2015.4).

You may convert a date into a val like this:

```
create data; //only necessary in sim-mode
for date %d = 1990 to 2012;
  data[%d] = val(%d) - 2000;
end;
```

This will not work without the val() function. The result is this (for the last three years):

```
                  data
  2010         10.0000
  2011         11.0000
  2012         12.0000
```

You may subtract two dates or add/subtract an integer to a date (for daily dates, possible February 29 leap days are included):

```
prt 2020m2d28 - 1, 2020m2d28 + 1;         //2020m2d27, 2020m2d29
prt 2020m3d1 - 2020m2d27;                 //3
prt observations(2020m2d27, 2020m3d1);    //4
```

The date difference using `-` is always the same as the number of observations minus 1.

Note

See the page with syntax diagrams if the basics of names, expressions, etc. is confusing.

If you need to convert a VAL or STRING scalar to a DATE type, use the date() conversion function.

Regarding variable types and the Gekko type system, see the VAR section. In this appendix, variable assignment rules, including variable types, is explained in more detail.

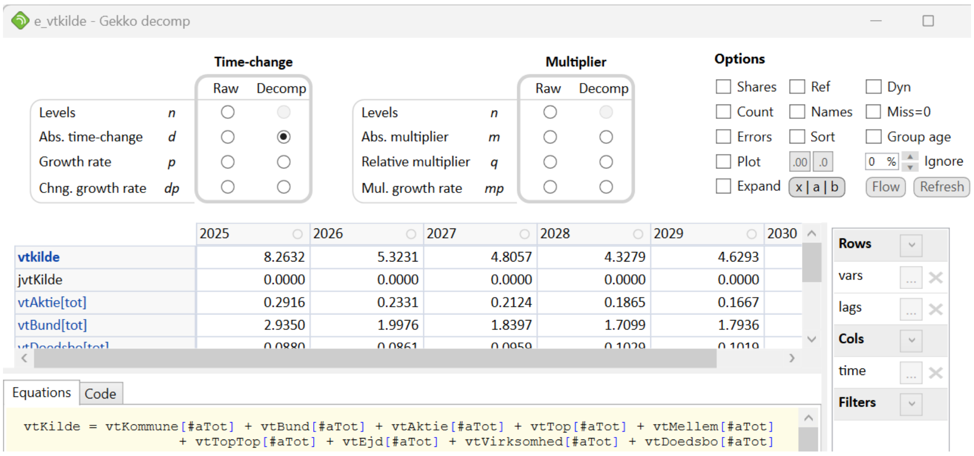
Related statements

STRING, VAL, FOR, IF, TIME

4.3.19 DECOMP

The DECOMP statement analyzes model equations, trying to figure out why a particular variable changes from period to period, or from a reference databank to a "multiplier" databank. In essence, DECOMP uses is a linearization of the model equations, in order to calculate the contributions. Consider the model equation `y = p1 * x1 + p2 * x2` where `y` could be a current prices aggregate of the quantities `x1` and `x2`, with prices `p1` and `p2`. If `y` changes by 2%, how do `p1`, `p2`, `x1` and `x2` contribute to that change? Gekko can compute these contributions/explanations, and also link up with other equations if needed. If for instance `x1 = c1 * x` and `x2 = c2 * x`, Gekko can merge these `x1` and `x2` equations into the `y` equation, essentially transforming it into `y = (p1 * c1 + p2 * c2) * x`, following which changes in `y` can now be explained by changes in `p1`, `p2`, `c1`, `c2`, and `x`.

DECOMP works both with normal Gekko models and GAMS-type dimensional models (cf. descriptions under the MODEL statement). Regarding GAMS-type models, a so-called scalar representation of the model is needed. For Danish language speakers, there are these detailed examples (decomposing the particular GAMS and Gekko models MAKRO and ADAM). These examples are self-contained (containing both models and data) and do not presuppose any GAMS installation.



Time-change

		Raw	Decomp
Levels	n	○	○
Abs. time-change	d	○	●
Growth rate	p	○	○
Chng. growth rate	dp	○	○

Multiplier

		Raw	Decomp
Levels	n	○	○
Abs. multiplier	m	○	○
Relative multiplier	q	○	○
Mul. growth rate	mp	○	○

Options

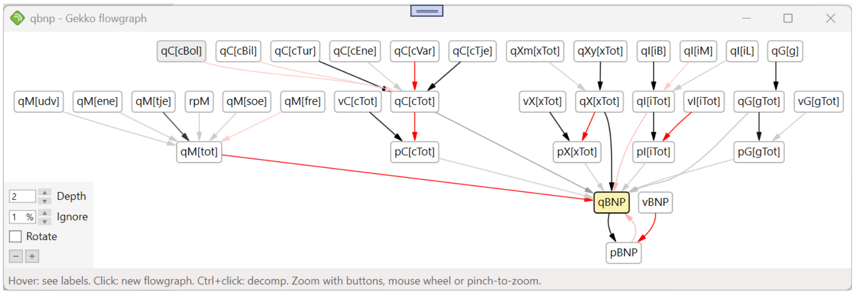
□ Shares □ Ref □ Dyn
□ Count □ Names □ Miss=0
□ Errors □ Sort □ Group age
□ Plot .00 .0 0 % Ignore
□ Expand x | a | b

	2025	2026	2027	2028	2029	2030
vtkilde	8.2632	5.3231	4.8057	4.3279	4.6293	
jvtKilde	0.0000	0.0000	0.0000	0.0000	0.0000	
vtAktie[tot]	0.2916	0.2331	0.2124	0.1865	0.1667	
vtBund[tot]	2.9350	1.9976	1.8397	1.7099	1.7936	
vtDoedsbo[tot]	0.0880	0.0861	0.0959	0.1029	0.1019	

Rows
vars ...
lags ...
Cols
time ...
Filters

```
vtKilde = vtKommune[#aTot] + vtBund[#aTot] + vtAktie[#aTot] + vtTop[#aTot] + vtMellem[#aTot]
          + vtTopTop[#aTot] + vtEjd[#aTot] + vtVirksomhed[#aTot] + vtDoedsbo[#aTot]
```

Gekko has a flowgraph functionality, activated by pressing the "Flow" button. This will show a flowgraph containing recursive decomposition to depth = 2 (default is 2, can be changed in the flowgraph window). In the flowgraph window, you may also change the "Ignore" percentage in order to remove irrelevant arrows and boxes, the flowgraph may be rotated, and the [-] and [+] buttons are for zooming.



2 Depth
1 % Ignore
□ Rotate

Hover: see labels. Click: new flowgraph. Ctrl+click: decomp. Zoom with buttons, mouse wheel or pinch-to-zoom.

When hovering the boxes, variable labels are shown, while clicking a box opens up a new flowchart window. With Ctrl+click, a new DECOMP window is opened instead. The current help page is pretty basic right now, just describing the syntax.

Mouse use

- You can use mouse-hovering.
- Clicking a variable in a Gekko `.frm` model opens a new DECOMP window (use Ctrl-click to open up a FIND window instead).
- Clicking a variable in a GAMS scalar model opens up a new FIND window.
- Double-clicking is not used for DECOMP or FIND windows.

Syntax

```
decomp < period operator COUNT DYN ERRORS IGNORE= MISSING=  NAMES SHARES
SORT PLOT >  variable FROM equations ENDO variables ROWS names  COLS
names;

decomp < period operator COUNT DYN ERRORS IGNORE= MISSING=  NAMES SHARES
 SORT PLOT >  variable;
```

period	(Optional). Local period, for instance `2010 2020`, `2010q1 2020q4` or `%per1 %per2+1`.
operator	(Optional). Operator, choose between `n`, `d`, `p`, `dp`, `rn`, `rd`, `rp`, `rdp`, `m`, `q`, `mp`. To show non-decomposed ('Raw') values, prefix with `x`, for instance `xd` or `xm`.
COUNT	(Optional). Show the number of elements hidden/aggregated inside the particular table cell.
DYN	(Optional). Resolve dynamic lagged/leaded variables. For instance, without DYN an equation like `y = 0.5 * y[-1] + x;` will show contributions from `y[-1]` and `x`, when `y` is decomposed. When using DYN, the equations is resolved over time, in effect unrolling it into `y = x + 0.5 * x[-1] + 0.25 * x[-2] + ... ;` and hence showing effects from `x`, `x[-1]`, `x[-2]`, etc. Note that DYN strongly depends upon the particular time period, so try using DYN on a sample containing a few periods first (note: DYN has no effects on a 1-period sample). Using DYN on a long period may be very time-consuming, because of matrix inversion.

ERRORS	(Optional). Show residuals and non-linear errors. Residuals are data errors in the equations (differences between left- and right-hand sides). • Data residuals should normally be = 0 for simulated values. Note that when equations are linked, residuals for each equation show up (`Residual1`, `Residual2`, etc.). • Errors originate from possible non-linearities in the equation(s) (for a linear equation, these errors should be = 0). If the variables are shown on rows, the error value is computed so that the first row equals the sum of the rest of the rows. With this option, the yellow/orange/red circled markers are generally absent, because the contributions sum up to the dependent variable per definition (the Error row/column makes sure of that).
IGNORE=	(Optional). Ignores small values. For instance, IGNORE=1 will filter out contributions amounting to less than 1% contribution to the change in the dependent variable. When ignoring, yellow/orange/red circled markers may show up, notifying that the shown contributions do not sum up to the dependent variable.
MISSING=	(Optional). Choose between `m` or `zero`. With `missing=zero`, missing values are treated as zeroes. TODO: This does not work properly yet.
NAMES	(Optional). Show the names of the elements "hidden" inside the particular table cell.
SHARES	(Optional). Show values as shares of the chosen ('dependent') *variable*. If variables are shown on rows, the rows are normalized so that the first row equals 100%.
SORT	(Optional). Sort the values by numerical size.
PLOT	(Optional). Shows a plot instead of a table. Like PLOT, the plot is produced by gnuplot, internally using .svg files. You may hover over lines to see variable names. You may zoom the plot size using `option decomp plot zoom = ... ;` (default = 100).
variable	The (dependent) variable that is to be decomposed. This variable is always shown first in the rows/columns.

FROM	Indicate a list of equation names, containing the equations that are to be used (linked).
ENDO	Indicate the endogenous variables. If there is only one equation, ENDO contains the name of the (dependent) *variable* that is being decomposed. If there are more than one equation, the (dependent) *variable* will be part of the list, and ENDO tells Gekko how to link up the equations. These variables are "substituted out" and do not show up as contributions (except for the dependent variable).
ROWS	Fields to show on rows, for instance: `vars, lags`.
COLS	Fields to show on columns, for instance: `time`.

- If no period is given inside the <...> angle brackets, the global period is used (cf. TIME).

Details

If you want to decompose a particular variable but do not know which equation it is determined in, you can use DECOMP without the FROM and ENDO parts, for instance `decomp x1;` to find all equations containing `x1`.

For 'Raw' operators (left part of operator columns).
- 'Shares' cannot be selected.
- 'Dyn' has no effect.
- 'Errors' will only show residuals (shown last).
- 'Sort' will sort according to levels. Residuals will be sorted, too.
- 'Ignore' will be ignored.

For 'Decomp' operators (right part of operator columns).
- 'Shares' will show relative values (relative to the chosen/dependent variable).
- 'Errors' shows residuals, ignored and errors. Residuals are shown after other variables, unless 'Sort' is chosen. Ignored are always shown second-last (when active), and errors are always shown last. When 'Errors' is not selected, a yellow/orange/red circle shown at the time period label will indicate that 'Errors' contain significant contributions.
- 'Sort' will sort according to relative values (relative to the chosen/dependent variable). Residuals will be sorted, too.
- 'Ignore' will gather all contributions with relative values smaller than the chosen value. The 'Ignore' component may include residuals (but will always exclude errors).

Examples

Decompose the variable `y` from the equation `e1`:

```
decomp <2010 2020 m> y from e1 endo y;
```

Note that when decomposing 1 equation, the `endo` variable must always be the same as the selected variable.

```
decomp y from e1, e2 endo y, c;
```

Decompose `y` using the two equations `e1` and `e2`, with the variables `y` and `c` set as endogenous variables.

```
decomp x[y] from e1, e2 endo x[y], x[c];
```

Same as before, this time using array-series.

```
decomp x[y];
```

This shows a list of equations containing `x[y]`.

Note

To quickly close all DECOMP windows, you may use the CUT statement or the "Close all PLOT and DECOMP windows" button in the Gekko main window. When using this button, and if the Gekko main window is out of focus, you may have to click the button two times (the first time brings the Gekko main window back in focus).

When a DECOMP statement is issued, Gekko takes a copy (data snapshot) of the first-position and reference databanks, for use with subsequent DECOMP windows. If you issue a new DECOMP statement while there are still old unclosed DECOMP windows floating around, these windows will also be linked to the new data snapshot. Use the "Refresh" button to load data from databanks into the data snapshot.

Missing values are shown as `M`, and missing variables are shown as `N`.

You may copy-paste decomp cells to a spreadsheet (mark cell areas or use Ctrl+A, followed by 'Copy' or Ctrl+C). When pasting into for instance Excel, make sure that `option interface clipboard decimalseparator` corresponds to you Excel settings, and also make sure that your decimal separator is different from your thousands separator (if these are both `.` or both `,` numbers may become garbled). In Excel, see File --> Options --> Advanced --> Editing options.

In a Gekko model, when clicking on a variable `x` to find the equations it is part of, only equations where `x` appears non-lagged/leaded are listed. This restriction will be relieved.

When looking for series and array-series, autocomplete (`[Tab]` or `[Ctrl+Space]`) can be practical as a supplement to the INDEX statement.

Related options

option interface clipboard decimalseparator; [period, comma]
option decomp flowgraph depth = 2;
option decomp plot zoom = 100;

Related statements

FIND, READ, CLONE, MULPRT, DISP, CUT

4.3.20 DELETE

DELETE is used to remove variables from databanks.

When deleting timeseries, remember that default DELETE only deletes series of the currently active frequency from the first-position databank. You may use `:` and `!` to augment, for instance `delete *:*!*;` deletes all timeseries of all frequencies from all databanks.

Syntax

```
delete variables;
delete < NONMODEL > ;
```

NONMODEL	Removes superfluous timeseries in the first-position and reference databanks (provided a model has been defined with MODEL). The removal is only done for series of the same frequency as the global frequency setting. For instance, you might have a databank and model variable `y` for income. Now, imagine that the definition and contents of the variable is changed to `y2` in both the databank and model. If the old variable `y` still resides in the databank, this may create confusion, and the NONMODEL option removes such non-model variables. Cf. also the Gekko menu 'Utilities' --> 'Compare model/databank/varlist...'.

- If a variable is stated without databank, the databank is assumed to be the first-position databank.
- Note that 'naked' wildcards are allowed in this statement, so you may for instance use `a*b?c` as wildcard instead of the more cumbersome `{'a*b?c'}`.
- Looping: with a list like for instance `#m = a, b;`, you may use `delete x{#m};` to delete `xa` and `xb`.

Examples

Delete a series `x`, a string `%x`, and a list `#x`:

```
x = 100;
%x = 'a';
#x = a, b;
delete x, %x, #x;
```

If, instead, you want to delete the series corresponding to the contents of `%x` and `#x`, use {}-curlies:

```
a = 100;
b = 200;
c = 300;
%x = 'a';
#x = b, c;  //or: #x = ('b', 'c')
delete {%x}, {#x};  //deletes the series a, b, c
delete %x, #x;  //deletes the string %x and the list #x
```

You may use wildcards like in COPY, INDEX, RENAME, etc.:

```
delete **;
```

This will delete all variables from the first-position databank. Alternatively (and better):

```
clear first;    //or clear work, if Work is the first-position
databank
```

Another example:

```
delete x*!q;
```

This will delete all quarterly series starting with `x`. You may also delete a variable from a particular databank (provided that bank is opened with OPEN<edit> or unlocked with UNLOCK), for instance:

```
delete bank2:x1!q;
```

Remove non-model variables with this special option:

```
delete <nonmodel>;
```

Note

To clear the entire workspace, including databanks, list, scalars, models, etc., see RESTART or RESET. To delete the contents of databanks, see CLEAR.

Related statements

CLEAR, CLOSEALL, RESTART, RESET

4.3.21 DISP

The statement is primarily used to print series or array-series, showing precedents and dependents if a model is loaded, and showing meta-information (cf. DOC). If a variable list is contained in the model file (`.frm` file) or as an external `varlist.dat` file (cf. MODEL), this information is shown, too.

When displaying timeseries, remember that default DISP only shows series of the currently active frequency from the first-position databank. You can use `:` and `!` to augment, for instance `disp *:*!*;` displays all timeseries of all frequencies from all databanks. Wildcard logic, including double and triple stars etc., is explained more generally on the wildcards page.

If a model is loaded, the DISP statement starts the equation browser. This means that linked variables can be clicked, and that you may browse forwards and backwards by means of the arrow buttons in the user interface. The 'home' button will browse back to the first DISP that started the equation browser.

For GAMS models, there are special options regarding how to identify which variable a given equation determines.

When displaying an array-series, the dimensions, possible domains, etc. are shown. DISP of other variable types than series works like PRINT.

Syntax

```
disp < period  INFO >  variables ;
disp 'search string' ;
```

period	(Optional). Local period, for instance `2010 2020`, `2010q1 2020q4` or `%per1 %per2+1`.
INFO	(Optional). Used to print out right-hand side variables for a given endogenous variable. Mostly used when a SIM breaks down, together with `option solve failsafe = yes`.
variables	Variables or lists (wildcards and bank indicators may be used), and items may be separated by commas.
'search string'	A string in single quotes to search for in all labels. Gekko will search for the string in both the variable list (if such a list is loaded with the model), and in the labels of each timeseries (cf. DOC).

- If no period is given inside the <...> angle brackets, the global period is used (cf. TIME).

- If a variable is stated without databank, the databank is assumed to be the first-position databank.
- Note that 'naked' wildcards are allowed in this statement, so you may for instance use `a*b?c` as wildcard instead of the more cumbersome `{'a*b?c'}`.

Example

DISP the volume of GDP and private consumption for the (local) period 2000-2010:

```
disp <2000 2010> fy, fcp;
```

If a model is loaded, you will be able to see which variables the given variable affect (dependents). You will also see the equation (if the variable is endogenous), and hence the variable’s precedents. These variables are clickable, so the DISP statement functions as an entrance to the equation browser.

If a variable list is put after a 'VARLIST;' or 'VARLIST$' in the model file (or is located in an external varlist.dat file), this meta-information is shown. You may search these labels in the following way:

```
disp 'import';
```

This will list all variables with a label containing this search string.

Wildcards can be used:

```
disp bank2:x*!q;
```

Displays quarterly series starting with 'x', from `bank2`.

Per default, only 3 lines of data is written when DISP'ing a variable. However, you can click the link ('show') to see any hidden periods. This limitation is intended for easier use of DISP as an equation browser.

Note

Regarding DISP of GAMS equations, see the description of the `model <dep = ...>` local option under MODEL.

You can use a TIMEFILTER to omit periods for a more readable output. (If a TIMEFILTER is set, the `print disp maxlines = 3` option is overruled, so that all non-filtered periods are shown even if there are more than 3 of these).

The DISP<info> statement can be used to print out right-hand side variables for a given endogenous variable. It can only be used for a one-period time period. It is called automatically if failsafe mode solving is set (`option solve failsafe yes`) and the simulation fails.

DISP shows meta-information in the following way:

- First, if there is a variable list contained in the model file (`.frm` file) or as an external `varlist.dat` file, this information is shown first, and the lines are shown in raw form, line by line exactly as they appear in the variable list.
- Next, if the timeseries itself contains meta-information (cf. `DOC`), this is shown, too. A label field is shown as it is, a source field is shown with `"Source:"` prefix, and a units field is shown with `"Units:"` prefix.

Related options

OPTION model type = default; //default | gams
OPTION print disp maxlines = 3;
OPTION model gams dep current = no;
OPTION model gams dep method = lhs; // lhs | eqname

Related statements

PRT, MULPRT, PLOT, DECOMP, TELL, FIND

4.3.22 DOC

The statement is used to manually change meta information fields in a timeseries object. This information is stored when a databank is written to a `.gbk` file. The meta information is shown in the DISP statement (when doing modeling: together with any information stored inside a model .frm file or in an external `varlist.dat` file).

The DOC statement may alternatively be used to produce a stand-alone html browser of a model's equations.

Syntax

```
doc  variables  LABEL=...  SOURCE=... UNITS=... STAMP=... ;
doc <varlist> FILE=... ;
doc <browser>;
```

variables	Variablename(s) or list(s) (wild-cards are allowed). You may prepend a databank name as bank:variable.
LABEL=	(Optional). Changes the label of the timeseries. You may use LABEL='' to clear.
SOURCE=	(Optional). Changes the source of the timeseries. You may use SOURCE='' to clear.
UNITS=	(Optional). Changes the units of the timeseries. You may use UNITS='' to clear.
STAMP=	(Optional). Changes the stamp of the timeseries. You may use STAMP='' to clear.
VARLIST	(Optional). Get/import labels from Gekko model or from file.
FILE=	(Optional). Possible file name for <varlist>.
BROWSER	(Optional). Generate html browser.

- If a variable is stated without databank, the databank is assumed to be the first-position databank.

`doc <varlist>` imports meta-information from a loaded MODEL (See the section "Gekko models: variable list"). With `doc <varlist> file=... ;` you may indicate a file containing the meta-information (for instance a file named `varlist.dat`).

`doc <browser>` produces a stand-alone equation browser in html, which can, for instance, be put on a web server. The produces system is independent of Gekko and shows variables, formulas, labels, graphs, estimation output, data, etc. In essence, the system replicates how DISP can show equations etc. from inside Gekko.

Examples

To change label, source and stamp on the timeseries `fY`, use:

```
doc fY label='Gdp' source='Statistics Denmark'  stamp='11-01-2024';
```

To clear the label, use an empty string:

```
doc fy label='';
```

You may delete all label/source/units meta information in the first-position databank like this:

```
doc * label='' source='' units='' stamp='';
```

If you would like to store meta information in a spreadsheet `varlist.xlsx`, you can get this meta information into `databank.gbk` with something akin to the following (depending upon how you design the spreadsheet). Here, the text information in the spreadsheet is designed with one row per timeserie:, column 1 is the series names, column 2 is the labels, column 3 is the sources, and column 4 us the units (the first row of the spreadsheet is skipped). See LIST regarding list indexing like `[.., 1]` etc.

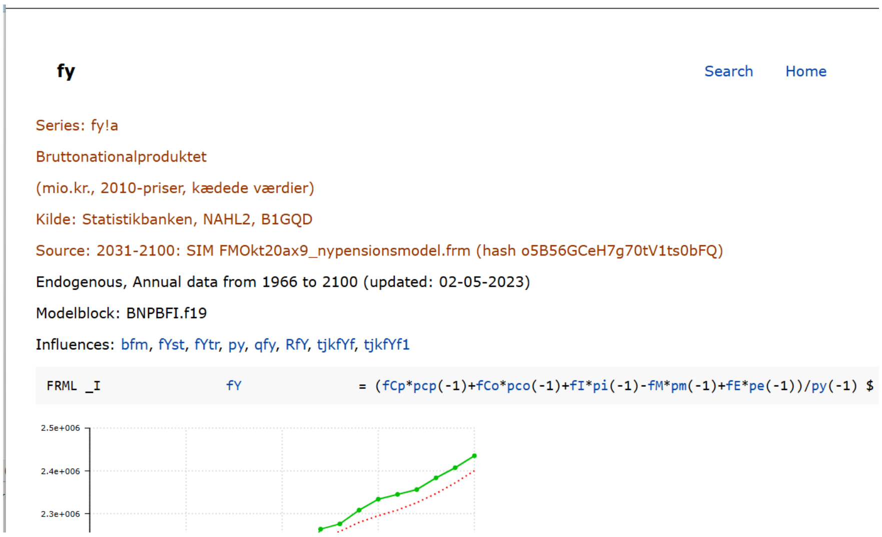
```
SHEET <import list> #v file = varlist.xlsx;
#varlist = #v[2..]; //skips first row with column explanation
#variables = #varlist[.., 1];
#label     = #varlist[.., 2];
#source    = #varlist[.., 3];
#units     = #varlist[.., 4];
open <edit> databank;
for val %v = 1 to length(#variables);
  %varname = {#variables[%v]}.getname(); //omits any bank or
frequency indicators
  doc {%varname} label = #label[%v] source = #source[%v] units =
#units[%v];
end;
close databank;
```

Html browser

If you first load a model with `model` (preferably the model `.frm` file includes a variable list, or a variable list is supplied from a `varlist.dat` external file) and read one or two databanks with `read`, you may issue a `doc<browser>` afterwards:

```
reset;
time 2010 2025;
model mymodel;
read mybank;
read<ref> myrefbank;
doc<browser>;
```

The `doc<browser>` statement will produce a html browser in a `\browser` sub-folder to your current working folder. Open the folder and start up `index.html` in a web browser to proceed. The html browser is stand-alone and hence independent of Gekko, model files or databank files, and plots and prints are shown for the given period (in the above example: 2010-2025). The variable pages look like the following:



fy

Series: fy!a

Bruttonationalproduktet

(mio.kr., 2010-priser, kædede værdier)

Kilde: Statistikbanken, NAHL2, B1GQD

Source: 2031-2100: SIM FMOkt20ax9_nypensionsmodel.frm (hash o5B56GCeH7g70tV1ts0bFQ)

Endogenous, Annual data from 1966 to 2100 (updated: 02-05-2023)

Modelblock: BNPBFI.f19

Influences: bfm, fYst, fYtr, py, qfy, RfY, tjkfYf, tjkfYf1

```
FRML _I          fY          = (fCp*pcp(-1)+fCo*pco(-1)+fI*pi(-1)-fM*pm(-1)+fE*pe(-1))/py(-1) $
```

If you want to tweak the layout, you may put a `browser.json` file in your working folder. The settings are as follows (with example values):

```
{
  "index_filename": "ADAMdok.html",     //Name of main html page
  "list_filename": "list.html",         //Filename for for list
  "find_filename": "find.html",         //Filename for the search
page
  "css_filename": "styles.css",         //Styles used in the html
pages
  "dok_filename": "supdok.lst",         //Possible extra output
  "est_filename": "est.lst",            //Possible estimation output
  "icon_filename": "figen.ico",         //Icon file, used in
index_filename
```

```
  "vars_foldername": "vars",           //Sub-folder name (also used
in index_filename)
  "commands": "<gekko commands>",      //Gekko commands to set up
model and databanks
  "plot_start": "2010",                //start for plots
  "plot_end": "2030",                  //end for plots
  "plot_line": "2022",                 //vertical line for plots
  "print_start": "2020",               //start for print
  "print_end": "2030",                 //end for print
  "show_source": false,                //show source information
(labels), true|false
  "include_p_type": "no",              //include P equations
("predict-equations")? "yes"|"no"
  "ekstrafiler": [ "okt20.frm"]        //Files used by
index_filename
}
```

If you do not want to show graphs/prints of variables from a reference databank, you may use `read<first>mybank;` so that there will be put no variables in the reference databank (omitting the `<ref>` makes Gekko produce a reference databank as a clone or the first-position databank).

Note

Meta information like this is read from and written to .gbk or .tsd files.

Regarding meta-information on timeseries, you may set these directly when defining the series, for instance `<label = 'Value added tax'> vat = 0.25;`.

Related statements

READ, IMPORT, WRITE, EXPORT, DISP

4.3.23 DOWNLOAD

At the moment, the statement is used to interface to a particular Danish databank containing among other things timeseries data. The downloaded file is in 'px' format, that is, PC-Axis. This is a format widely used by statistical offices.

It is the intention to augment the DOWNLOAD statement regarding other online databanks (jobindsats.dk for instance). Note that you can import a px file with IMPORT<px> or IMPORT<px array>.

The data is downloaded into the first-position databank. If you can, it is advised to put the time element/code last in the .json definition file. If this is not done, Gekko will proceed anyway, but loading the data will take a bit more time.

Syntax

```
download < ARRAY > url  filename   DUMP=...;
```

ARRAY	(Optional). If this is set, and DUMP is not used, Gekko will put the data into array-timeseries rather than normal timeseries. If DUMP is used, you may use IMPORT <px array> afterwards.
url	Url (web address) to the databank. Note: the web address should be in quotes.
filename	Filename of the .json file defining what data to download. Remember to put the time dimension as the last dimension in your .json file.
DUMP=	(Optional). Name of the file in which to store the contents of the download (in this case, a px-file). Beware that when importing the px file afterwards, you should consider using `import<px variablecode>` if you want the dimension codes to correspond to what you get with a normal DOWNLOAD (that is: shorter i length).

Examples

Example:

```
reset;
option freq m;
```

```
time 2000 2016;
download  'https://api.statbank.dk/v1/data'  statbank.json;
plot {'*'};
```

This imports data from api.statbank.dk, with the file statbank.json file describing what data to download.

```
----------------------- statbank.json
---------------------------------
{
   "table": "pris6",
   "format": "px",
   "valuePresentation": "Value",
   "variables": [
      {
         "code": "VAREGR",
         "values": ["011200", "011100"]
      },
      {
         "code": "enhed",
         "values": ["100"]
      },
      {
         "code": "tid",
         "values": ["*"]
      }
   ]
}
--------------------------------------------------------------------
------
```

You may use `["*"]` to get all values of the field. The resulting series are called `pris6_VAREGR_011200_enhed_100` and `pris6_VAREGR_011100_enhed_100`.

After the DOWNLOAD statement, these two timeseries are available in the first-position (Work) databank. The above provedure can be split into two parts (first dumping the download as data.px, and then importing that file):

```
reset;
option freq m;
time 2000 2016;
download 'https://api.statbank.dk/v1/data'  statbank.json   dump =
data;
import <px variablecode> data;  //variablecode option to get the
same dimension names as normal DOWNLOAD
plot {'*'};
```

If you prefer to use array-series, you may use that <array> option:

```
reset;
option freq m;
time 2000 2016;
download <array> 'https://api.statbank.dk/v1/data'  statbank.json;
plot {'*'};
```

or in two steps:

```
reset;
option freq m;
time 2000 2016;
download 'https://api.statbank.dk/v1/data'  statbank.json   dump =
data;
import <px array> data;
plot {'*'};
```

This produces array-series `pris6['011200', '100']` and `pris6['011100', '100']`. Because of the leading zero of the first element, you cannot use for instance `pris6[011200, 100]` to refer to the first array-series (it will be understood as `pris6[11200, 100]`).

Setting up the .json file

For users of statistikbanken.dk, you may construct the .json file interactively as follows:

Go to https://api.statbank.dk/console

- Under (1) choose "Retreive data".
- Under (2), type the table name under "Tabel id" (for instance `pris6` or `folk2`). You may find the table name by browsing statistikbanken.dk. Under "Format" choose "PX". Click "Variable- and value codes" and choose the fields you want to obtain as dimensions of the array-series (it is preferred to choose time as the last dimension). You may choose particular values ("Value-ids") for the fields, or simply `*` for all values. Click "Download the result as file".
- Under (3), click "Execute".

Now you should download a `data.px` file that can be read into Gekko with `import <px array> data;`. Try `disp pris6;` if you have been downloading the `pris6` table, in order to see the dimensions. Next, copy-paste the .json code under (3) into a new file `statbank.json`, and put the file into the Gekko working folder. After this, you should be able to use `download 'https://api.statbank.dk/v1/data' statbank.json;` to download the data directly into Gekko.

Reading the px format

The PC-Axis px format is a flexible data format well suited for multidimensional data. The format is used by many statistical offices in different countries to let their users retrieve statistics. Gekko does not use all of the contents of a px file. The way Gekko reads it is the following:

For instance, the timeseries name `PROD01_saesonkorrigering_EJSAESON_brancheDB07_BC` may be composed from the px file (and the timeseries may get the following label (metadata): "Ikke sæsonkorrigeret, BC Råstofindvinding og industri"). The timeseries names and data are extracted as follows:

- `MATRIX=` . Gets the table name from this (used in the timeseries names), for instance `PROD01`.
- `CODES("tid")`. Decodes the time periods used. The alternative `CODES("time")` is allowed.
- `CODES(...)`. Gets dimension names and dimension elements from these `CODES`-fields, for use in the timeseries names. For instance, the name part `brancheDB07_BC`, where the first part (`brancheDB07`) is the dimension name, and the last part (`BC`) is the dimension element. Read more about abbreviated codes in the following section.
- `VALUES(...)`. Only used for metadata in the timeseries (timeseries labels).
- `VARIABLECODE(...)`. These fields provide shorter names for the variables, for instance `VAREGR` instead of `varegruppe`. This is automatically activated regarding the DOWNLOAD statement, but for `IMPORT<px>` you have to activate these shorter names with `IMPORT<px variablecode>`.
- `DATA=` . Read the data from here. If, for instance, there is one dimension with 3 elements, and another with 4 elements, Gekko expects 12 numbers in all. Gekko will not accept if a number is split between lines, and numbers should preferably always be followed by a blank also at the end of the line (this is recommended in the px definition). Gekko will count the numbers, and a warning is issued if there are too few numbers compared to the span of the dimensions. In that case, the data may be scrambled/misaligned in Gekko, so take care! If there are too many numbers, Gekko will fail with an error.
- `STUB=` . Is not used!

Note that some sources of px files provide very long single lines of data (thousands of characters). If such a file is opened in a text editor and saved afterwards, the editor may insert line breaks that may render the file unreadable in Gekko (because numbers become split between lines).

Abbreviated codes

The codes inside the px file are ofte very verbose, so if possible, the often shorter dimension names and dimension elements from the .json file are used instead. For instance, in the .json file in the above example, the dimension `VAREGR` is queried, asking for the elements `011200` and `011100` (this is done in .json with `"code":`

"VAREGR", "values": ["011200", "011100"]). But in the returned px file, we get the following line: CODES("varegruppe")="011200","011100";. Here, the element names are unchanged, but the dimension name is changed from VAREGR to varegruppe, the latter being more verbose. The former name is used if you download normally with the DOWNLOAD statement, whereas the latter name is used if you for instance first download a px file (with dump=... option) and then read the px file with import<px>. To get Gekko to use the shorter codes when importing, you may use the variablecode option: IMPORT<px variablecode>.

For non-array downloading, the dimension elements (CODES elements) may be adjusted compared to the .px file. Any characters æ, ø, å, Æ, Ø, Å are changed into ae, oe, aa, AE, OE, AA, and subsequencly any character that is not a...z or 0...9 is removed (including hyphens - and underscores _). But normally, there are not too many "illegal" characters in px CODES.

Note

There is currently (March 2024) a limit of 1.000.000 observations when downloading, cf. changelog. Users often download data over a large period, and the issue is that data is often changed only in a few periods at the end of the sample (still, the users often download the whole period to guard against changes in old "historical" data). DST has said that it would be possible to implement a feature that indicates which periods of a given sample contain changed data since a given previous download date. With such a feature, a lot of redundant downloading could be avoided, and the new downloaded data could simply be merged with the older data from the existing Gekko databank (Gekko has features to do such merging easily). This principle is known from incremental backups, and from time to time the full dataset could be downloaded (for safety).

For more advanced px reading, you may take a look at the pxr package in R.

Related statements

IMPORT, READ, OPEN

4.3.24 EDIT

The EDIT statement uses Notepad to open up the designated file. The statement is practical for editing Gekko program files (.gcm), file lists, table or menu files, data files like .csv, .prn, etc. See also XEDIT for xml files.

You may use remote control for Gekko program files, cf. `option interface remote = yes|no;`.

Syntax

`edit filename ;`

filename	Filenames may contain an absolute path like `c:\projects\gekko\bank.gbk`, or a relative path `\gekko\bank.gbk`. Filenames containing blanks and special characters should be put inside quotes. Regarding reading of files, files in libraries can be referred to with colon (for instance `lib1:bank.gbk`), and "zip paths" are allowed too (for instance `c:\projects\data.zip\bank.gbk`). See more on filenames here. The extension .gcm is automatically added, if it is missing. If the filename is set to '*', you will be asked to choose the file in Windows Explorer.

Examples

You may use this to open up the file forecst1.gcm from the working folder:

```
edit forecst1;
```

The .gcm extension is automatically inserted. You may select .gcm files like this:

```
edit *;
```

This will open up a file dialog with .gcm files to choose from.

Note

Most people use a "real" external text editor like for instance Kedit or Sublime Text. There are kex files available for Kedit, and for Sublime Text, see these open-source modules for Sublime and Gekko integration (including remote control).

Related options

OPTION interface remote = no; [yes|no]

Related statements

SYS, XEDIT

4.3.25 ELSE

An ELSE statement is used in conjunction with an IF condition and an END statement.

Related statements

IF, ELSEIF, END, FOR

4.3.26 ELSEIF

An ELSEIF statement can be used to filter through different conditions and is used in conjunction with an IF condition and an END statement. The syntax for this is simpler than using ELSE and IF instead, cf. the examples.

Examples

ELSEIF can be used to sift through possibilities, like this:

```
%x = 1;
if (%x == 1);
  //do something when 1
elseif (%x == 2);
  //do something when 2
elseif (%x == 3);
  //do something when 3
else;
  //do something when not 1, 2 or 3
end;
```

The same can be done without using ELSEIF, but in that case a number of END's need to be added:

```
%x = 1;
if (%x == 1);
  //do something when 1
else; if (%x == 2);
  //do something when 2
else; if (%x == 3);
  //do something when 3
else;
  //do something when not 1, 2 or 3
end; end; end;
```

These END's are inconvenient and stems from the fact that the above code can be reorganized into the following structure:

```
%x = 1;
if (%x == 1);
  //do something when 1
else;
  if (%x == 2);
    //do something when 2
  else;
    if (%x == 3);
      //do something when 3
    else;
```

```
      //do something when not 1, 2 or 3
    end;
  end;
end;
```

Here it is more clear why there are three ending END's. Using ELSEIF avoids that.

Related statements

IF, ELSE, END, FOR

4.3.27 END

An END statement concludes a BLOCK, FOR (loop), IF/ELSEIF/ELSE (condition) or FUNCTION/PROCEDURE statement. Gekko will fail if the END statement is missing.

To execute for instance a loop in the statement window, it is often convenient to use `[Ctrl+Enter]` for newlines, and then execute all the lines as a unified block by means of marking all the relevant lines and hitting `[Enter]`. In that case, the lines are executed in the same way as using a Gekko program file.

Related statements

BLOCK, FOR, IF, FUNCTION, PROCEDURE

4.3.28 ENDO

ENDO and EXO are used for *fixing*, that is, setting variables to some values (goal), and asking the system to solve this by means of other variables (means). The ENDO statement works differently depending upon `option model type` (if a model is loaded with the MODEL statement, this option is set automatically).

- With `option model type = default`, ENDO endogenizes a list of variables (without date settings). A model must be defined beforehand.
- With `option model type = gams`, ENDO produces array-series with names starting with 'endo_'. These array-series can subsequently be used to tell e.g. GAMS which variables are fixed and non-fixed. ENDO must indicate dates.

Use UNFIX to remove previously set ENDO or EXO variables.

Syntax

```
default type:     endo variable1, variable2, ... ;
gams type:        endo <period0> variable1 <period1>, variable2
<period2>, ... ;
```

variables	Default type: The variables are simple series names, or lists of these, for instance `x2`, or `{#m}`. Gams type: The variables are series or array-series names, for instance `x2` or `x2[a, b]`. For array-series, lists may be used, for instance `x2[a, #i, #j]`, where `#i` and `#j` are lists of strings.
period0	A period is a Gekko time interval like <2020 2030> or <2020q1 2030q4>. The general period can be set in the period0 field, and this period will be be used for the variables, unless specific periods are given in the period1, period2, etc.
period1, period2, ...	A period is a Gekko time interval like <2020 2030> or <2020q1 2030q4>. These specific periods will overrule the general period (period0).

Examples

Gekko type

If you need to exogenize a variable `fy`, and endogenize a variable `tg`, use this:

```
model forecst;  //Gekko model, will set "option model type =
default"
exo fy;         //a list of strings can be used, for instance {#m}
endo tg;
sim <fix>;      //option <fix> must be used to enforce the
goals/means.
```

Gams type

The following example exogenizes variables `x1[a, k1]` and `y1`, and endogenizes `x2[a, k1]` and `y2`.

```
option model type = gams; //may omit if a MODEL<gms> statement is
used first
exo x1[a, k1] <2022 2024>, y1 <2024 2026>;       //or: x1['a', 'k1']
endo <2023 2025> x2[a, k1] <2021 2023>, y2;      //or: x2['a', 'k1']
prt <2020 2027 width=20 n> exo_x1, exo_y1, endo_x2, endo_y2;
```

The resulting variables are as follows (note that these variables are overwritten if they exist beforehand):

```

             exo_x1[a, k1]              exo_y1      endo_x2[a, k1]
            endo_y2
  2020                   M                   M                   M
                  M
  2021                   M                   M              1.0000
                  M
  2022              1.0000                   M              1.0000
                  M
  2023              1.0000                   M              1.0000
             1.0000
  2024              1.0000              1.0000                   M
             1.0000
  2025                   M              1.0000                   M
             1.0000
  2026                   M              1.0000                   M
                  M
  2027                   M                   M                   M
                  M
```

Instead of individual elements, you may use lists:

```
option model type = gams; //may omit if a MODEL<gms> statement is
used first
#a = a1, a2;
#k = k1, k2;
exo <2022 2024> x1[#a, #k];
endo <2021 2023> x2[#a, #k];
prt <2020 2025 width=20 split n> exo_x1, endo_x2;
```

The two lists are automatically unfolded into 2 x 2 = 4 elements (sub-series) regarding `exo_x1` and `endo_x2`:

```
           exo_x1[a1, k1]        exo_x1[a1, k2]        exo_x1[a2, k1]
    exo_x1[a2, k2]
2020                     M                     M                     M
                  M
2021                     M                     M                     M
                  M
2022                1.0000                1.0000                1.0000
             1.0000
2023                1.0000                1.0000                1.0000
             1.0000
2024                1.0000                1.0000                1.0000
             1.0000
2025                     M                     M                     M
                  M

           endo_x2[a1, k1]       endo_x2[a1, k2]       endo_x2[a2, k1]
    endo_x2[a2, k2]
2020                     M                     M                     M
                  M
2021                1.0000                1.0000                1.0000
             1.0000
2022                1.0000                1.0000                1.0000
             1.0000
2023                1.0000                1.0000                1.0000
             1.0000
2024                     M                     M                     M
                  M
2025                     M                     M                     M
                  M
```

Note

With a Gekko model, the ENDO and EXO statements are non-cumulative, so all endogenized/exogenized variables should be present in the same ENDO/EXO statement.

With a GAMS model, the ENDO and EXO statements are cumulative in the sense that ENDO or EXO do not delete existing endo_... and exo_... array-series.

Related options

OPTION model type = default; //default | gams

Related statements

EXO, SIM, UNFIX

4.3.29 EXIT

The statement EXIT terminates the application (without any warning, so use it carefully). It is often used in order to run Gekko sessions from batch (.bat) files.

From the user interface, you may exit by means of 'File' --> 'Exit', or Alt+F4. To stop/abort a program while it is running, you can use the red stop button in the user interface.

Syntax

exit ;

Related statements

STOP, RETURN

4.3.30 EXO

ENDO and EXO are used for *fixing*, that is, setting variables to some values (goal), and asking the system to solve this by means of other variables (means). The EXO statement works differently depending upon `option model type` (if a model is loaded with the MODEL statement, this option is set automatically).

- With `option model type = default`, EXO exogenizes a list of variables (without date settings). A model must be defined beforehand.
- With `option model type = gams`, EXO produces array-series with names starting with 'exo_'. These array-series can subsequently be used to tell e.g. GAMS which variables are fixed and non-fixed. EXO must indicate dates.

Use UNFIX to remove previously set ENDO or EXO variables.

Regarding syntax, examples, etc., see the ENDO statement.

Related options

OPTION model type = default; //default | gams

Related statements

ENDO, SIM, UNFIX

4.3.31 EXPORT

The statement writes the first-position databank or specific variables to a non-gbk file in a particular format. Use WRITE to write to a .gbk file. Please note that the EXPORT formats currently only supports series (or a matrix), not other variable types (you may use WRITE to store these in .gbk files).

> **Compatibility note**: If a time period is not indicated in the <>-option field, Gekko 3.0 will only export data inside the global time period. Before Gekko 3.0, all data would have been exported. To emulate previous behavior, you can use EXPORT<all>. Alternatively, you may set "OPTION bugfix import export = yes;". If the option is set, IMPORT and EXPORT will work as in pre-3.0 versions. The option will be removed at some point, so it is better to change occurrences of date-less EXPORT to EXPORT<all> in old Gekko program files.
>
> **Excel note**: When exporting xlsx files, if you encounter "dates" with integer numbers larger than 20000, this may be because Excel shows the dates as numbers rather than dates. You may try to change the format of the date cells: right-click, "Format cells", "Date".

There is the following equivalence between EXPORT and WRITE: EXPORT = WRITE<respect>, and the inverse: WRITE = EXPORT<all>. If a local period is used, like `export <1980 2020>`, EXPORT and WRITE behave in the same way.

Syntax

```
export < period format  ALL  CAPS=...  COLS DATEFORMAT=...  DATETYPE=...
OP=... >  filename ;
export < period format  ALL  CAPS=...  COLS DATEFORMAT=...  DATETYPE=...
OP=... >  variables  TO  variables FILE=filename ;
```

period	(Optional). Without a time period indicated, Gekko will write all the data for all observations. When a period is indicated, the written data(bank) is truncated.
format	File format. Choose between ARROW, CSV, FLAT, GCM, GDX, GNUPLOT, PRN, PYTHON, R, TSD, TSP, XLS/XLSX (regarding gbk, see the WRITE statement). • ARROW: The Apache Arrow format, see the examples under R_RUN or PYTHON_RUN. IMPORT<arrow> does not work yet, but will soon. • CSV: Only frequencies matching the current frequency setting will be written. • FLAT: This is a special Gekko text-based format with lines that resemble series statements. See more details in the IMPORT section.

- GCM: This will export series as Gekko SERIES statements. You can use operators `n`, `d`, `p`, `m` or `q`, for instance `export<gcm op=p> {#vars} file=data;` to put the percentage change in the `#vars` timeseries into the file data.gcm. Alternatively, you may use `^=`, `%=`, `+=` or `*=` operators, for instance `export<gcm op='%='> {#vars} file=data;`. With the latter operators, you must enclose them in single quotes ('). You may use `export<gcm>` to export in levels (corresponding to operator `n`). A .gcm file is imported simply with RUN. See the FLAT format for a faster version of this format.
- GDX: A binary GAMS-database. Note "OPTION gams exe folder = ..." where it is possible to point to the exact GAMS folder (otherwise the system will try to auto-locate GAMS). It seems necessary to use a 32-bit version of GAMS, since the current version of Gekko is 32-bit. Please note that only array-timeseries (see SERIES) are written to the .gdx file, and that Gekko does not (at the moment) export timeless timeseries. GAMS can be freely downloaded as a demo, and the demo will work fine regarding Gekko EXPORT.
- GNUPLOT: Gekko writes a prn-like format suitable for gnuplot. If no period is set, Gekko will write all years occurring in the first-position databank. (Note: PLOT also implicitly produces such a data file, see the temporary files folder, under \gnuplot. Location is given with Help --> About... in the main Gekko window).
- PRN: Same behaviour as for the CSV type. Note that a similar looking style of output can be accomplished with PRT (for instance `prt <n> x1, x2 file=output.txt;`, and PRT has many formatting options regarding decimals, etc.).
- PYTHON: Exports matrices as a Python script file. For instance like this: `#m1 = [1, 2; 3, 4]; #m2 = [11, 12; 13, 14]; export <python> #m1, #m2 file=matrix.py;` Only matrices can be exported. For running Python more interactively, see PYTHON_RUN.
- R: Exports matrices as a R script file. For instance like this: `#m1 = [1, 2; 3, 4]; #m2 = [11, 12; 13, 14]; export<r> #m1, #m2 file=matrix.r;` Only matrices can be exported. For running R more interactively, see R_RUN.
- TSD: For interchange with AREMOS and others. With option 'CAPS=no', all .tsd variable names are written as they are (otherwise they will be written as all caps). When exporting daily frequency data, all days of the week (Monday-Sunday) are designated "active".
- TSP: Gekko will write TSP records (load statements). Works for annual frequencies only.
- XLS or XLSX: Gekko will try to write the data to an Excel workbook. Only frequencies matching the current frequency setting will be written. If no period is set, global time will be used. Cf. also the SHEET statement. The engine used for Excel writing can be changed with "OPTION sheet engine = ...;". You can also export a matrix to xlsx format.

ALL	(Optional). With this option, all observations are exported, regardless of the global time period. This corresponds to pre-3.0 Gekko behavior.
CAPS=	When exporting a tsd file, the default is now to write the variable names with all caps. This is because AREMOS fails if this is not done. To avoid the caps, you may use option <tsd caps=no>.
COLS	(Optional). For .csv, .prn or Excel files, this indicates whether the timeseries are running downwards in columns.
OP=	(Optional). For .gcm files, this value indicates the operator used for the SERIES statements.
DATEFORMAT= DATETYPE=	(Optional). These options control the date format for .xlsx and .csv files, and for the use of SHEET in Gekcel. `DATEFORMAT` can be either `'gekko'` (default, for instance `2020q3` or `2020m11`) or a format string like `'yyyy-mm-dd'`, and the latter may contain a `first` or `last` indicator, for instance `'yyyy-mm-dd last'`, which indicates for quarterly, monthly or weekly data that the *last* day of the quarter/month/week is used. `DATETYPE` can be either `'text'` or `'excel'`. In the former case, the dates are understood as text strings (for instance `'2020q3'` or `'2020-09-30'` for a quarterly date), and in the latter case (not relevant for .csv files), the date is understood as an Excel date, which basically counts the days since January 1, 1900. This number would correspond to 44104 for the date 2020-09-31, and can be shown in Excel in different ways depending upon date format settings, language settings, etc., but the internal number itself is unambiguous. When using SHEET in Gekcel, using `DATETYPE='excel'` is advised for consistency. [New in 3.0.5].
variables	Variables or lists (wildcards and bank indicators may be used), and items may be separated by commas. If no variables are given, the full first-position databank is written.
TO	You may use TO to rename variables before they are written, for instance `export <csv> x* to *_old file = test;`, where Gekko will look for variables starting with `x`, and the found variables will acquire a `_old` suffix. This logic is similar to the `COPY` and `RENAME` statements.
filename	Filenames may contain an absolute path like `c:\projects\gekko\bank.gbk`, or a relative path `\gekko\bank.gbk`. Filenames containing blanks and special characters should be put inside quotes. Regarding reading of files, files in libraries can be

	referred to with colon (for instance `lib1:bank.gbk`), and "zip paths" are allowed too (for instance `c:\projects\data.zip\bank.gbk`). See more on filenames here.

- If no period is given inside the <...> angle brackets, the global period is used (cf. TIME).
- If a variable is stated without databank, the databank is assumed to be the first-position databank.

Examples

You may export the contents of the first-position databank into a spreadsheet like this:

```
export <xlsx all> data;
```

This produces the file data.xlsx. The <all> option makes sure that all observations are exported: if omitted, only observations inside the global time period are exported. If you only want subset of the variables or a subset of the time period, you may write for instance:

```
export <2040 2050 xlsx> fy, fe, fm  FILE=sim4050;
```

This produces the file sim4050.xlsx, containing the three variables over the period 2040-50. You may also use lists or wild-card lists regarding the variables:

```
export <xlsx> fX* file=fxfile;
```

This writes all variables in the first-position databank starting with 'fX' to the file fxfile.xlsx.

```
export <2015 2020 gcm op=p> px* file=px;
```

This writes all variables in the first-position databank starting with 'pX' to the Gekko program file px.gcm. The variables are written as percentage growth SERIES statements (the data can be imported afterwards with RUN).

```
export <gdx> ats file=gamsdata;
```

This will export the array-timeseries `ats` to gamsdata.gdx.

Export of a matrix `#m` to Excel (`matrix.xlsx`):

```
export <xlsx> #m file = matrix.xlsx;
```

Note

You may use SHEET if you need to put expressions into an Excel sheet, or into particular cells.

If `option folder = yes`, and `option folder bank` is set, the EXPORT statement tries to write to that particular folder instead of the working folder.

See the Gekko menu 'Options' --> 'Program dependency tracking' or use `option global dependency tracking = ...;` to activate dependency tracking, so that the use of external files (for instance program files, read/written databanks etc.) are shown as a list at the end of a Gekko session.

Related options

OPTION folder bank = [empty];
OPTION interface csv decimalseparator = period; [period|comma]
OPTION interface csv delimiter = 'semicolon'; [semicolon|comma|tab]
OPTION interface csv ignoremissing = no; [yes|no]
OPTION interface csv ndec = 100;
OPTION interface excel ignoremissing = no; [yes|no]
OPTION interface excel language = danish; [danish|english]
OPTION interface prn decimalseparator = period; [period|comma]
OPTION interface prn ndec = 100;
OPTION interface prn delimiter = blank; [blank|semicolon|comma|tab]

Related statements

IMPORT, READ, WRITE, SHEET

4.3.32 FINDMISSINGDATA

This statement finds timeseries with missing values (only the timeseries with frequency matching the global frequency setting).

Please note that the statement is not intended to be put inside a large loop. In such cases, using the iif() function is better, see the end of the examples section.

Syntax

```
findmissingdata < period REPLACE=... > variables ;
```

period	(Optional). Local period, for instance `2010 2020`, `2010q1 2020q4` or `%per1 %per2+1`.
REPLACE=	You may use for instance <REPLACE = 0> to replace any missing values with 0 (or any other value). When using the REPLACE options, lists are not generated.
variables	List of variable(s) to check, array-series can be stated. If omitted, all series from the first-position databank are investigated.

- If no period is given inside the <...> angle brackets, the global period is used (cf. TIME).
- If no variables are given, all variables in the first-position databank will be investigated

Examples

For instance, the statement

```
findmissingdata <2008 2010>;
```

looks for all series (including array-subseries) with any missing values in the period 2008-2010. You may restrict it like this:

```
findmissingdata <2008 2010> {#vars};
```

where the list #vars contains the names of the relevant variables you want to check. Gekko outputs a number of lists from the investigation: for instance the list #missingdata contains all variables with missing data in the first-position databank, whereas the lists #missingdata_all, #missingdata_endo etc. are subsets of that list, and correspond to the Gekko-defined lists #all, #endo etc. (i.e., all model variables, all endogenous model variables, etc.).

```
findmissingdata <2008 2010 replace = 0> {#vars};
```

This does not produce any lists, but replaces any missing values with 0.

You may use wild-card lists if preferable:

```
findmissingdata <2008 2010> fX*, fYf*;
```

This will check all variables starting with 'fX' or 'fYf'. If a period is not given, the global time setting is used.

If you need to change missing values to something else, using the iif() function is often much more speedy. For instance:

```
reset; mode data; option freq m;
time 2017m7 2017m10;
x = 100, 200, m(), 400;
y = 110, 210, 310, 410;
z = iif(x, '==', m(), y, x);
prt <n> x, y, z;
```

The result is:

```

                 x             y             z
  2017
    m7       100.0000      110.0000      100.0000
    m8       200.0000      210.0000      200.0000
    m9              M      310.0000      310.0000
    m10      400.0000      410.0000      400.0000
```

The x series has a hole in it (2017m9), and the iif() function checks (for each of the four periods) if x has a missing value, and if so it uses the y value, else the x value. So the resulting z series has the hole filled with the 2017m9 observation from y. The m() function inside the iif() function just returns a missing value. A dollar conditional ($) works similar to iif(), and the replace() function can also be used.

Note

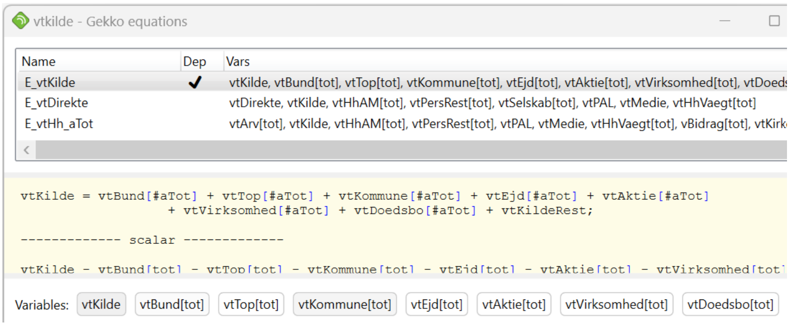
The statement is convenient when developing new models or changing existing models.

Related statements

COMPARE, DELETE, see also 'Utilities' --> 'Compare two databanks' (same as COMPARE)

4.3.33 FIND

The FIND statement is typically used together with DECOMP. For a given variable `x`, the FIND statement finds the equations that contain `x`. In addition, FIND can find the shortest path between two variables (traversing through model equations).



Name	Dep	Vars
E_vtKilde	✓	vtKilde, vtBund[tot], vtTop[tot], vtKommune[tot], vtEjd[tot], vtAktie[tot], vtVirksomhed[tot], vtDoeds
E_vtDirekte		vtDirekte, vtKilde, vtHhAM[tot], vtPersRest[tot], vtSelskab[tot], vtPAL, vtMedie, vtHhVaegt[tot]
E_vtHh_aTot		vtArv[tot], vtKilde, vtHhAM[tot], vtPersRest[tot], vtPAL, vtMedie, vtHhVaegt[tot], vBidrag[tot], vtKirk

```
vtKilde = vtBund[#aTot] + vtTop[#aTot] + vtKommune[#aTot] + vtEjd[#aTot] + vtAktie[#aTot]
                    + vtVirksomhed[#aTot] + vtDoedsbo[#aTot] + vtKildeRest;

------------- scalar -------------

vtKilde - vtBund[tot] - vtTop[tot] - vtKommune[tot] - vtEjd[tot] - vtAktie[tot] - vtVirksomhed[tot]
```

Mouse use

- You can use mouse-hovering on equations and on the variables at the bottom (click to select/retain a variable).
- Clicking an equation opens a DECOMP window.
- Ctrl-click selects/retains an equation, to look at it further in the lower part of the FIND window.
- Double-clicking is not used for the FIND window in Gekko 3.1.16+.

Syntax

```
find variable ;
find variable TO variable ;
```

TO	When using `... TO ...` , Gekko will find the shortest path between to variables in a model.

- If no period is given inside the <...> angle brackets, the global period is used (cf. TIME).

Examples

The following statement finds those equations that contain the variable `x`.

```
find x;
```

The statement

```
find x[a] to y;
```

finds the shortest path between `x[a]` and `y`, traversing through model equations.

Related statements

DECOMP, DISP

4.3.34 FOR

The FOR statement initiates a loop over strings, dates or values. Parallel loops (tandem) over lists are also possible. Like the procedure and function definitions, Gekko demands that the variable type is stated explicitly.

Guide: loops
For an easier introductory guide on Gekko loops, see this page.

Loop over elements

This elements loop loops through the list of elements on the right-hand side of '='. Indicating the type here is mandatory in Gekko 3.0.

```
for [type] %x = items ;
  statements... ;
end ;
```

[type]	The type must be indicated
`%x`	The loop variable `%x`
items	Any list of items. For a simple list of strings, you may use the naked list `a, b, c` instead of `('a', 'b', 'c')`, similar to how a list may be defined using short form. You may also use for instance lists (`#mylist`) or wildcards (e.g. `fx*`). You may also use a list of values, for instance `(1, 2, 3)`, or a list of dates, for instance `(2001q1, 2001q2, 2001q3)`.
Note that you may use parentheses, for instance `for([type] %x = items)`, like the IF statement.	

Parallel loop over elements

This parallel string loop loops through the items in parallel/tandem. So in the i'th iteration, `%s1` is equal to the i'th item in *items1*, `%s2` is equal to the i'th item in items2, etc. The number of items must be the same in all the lists on the right-hand sides of the '='. And the names on the left-hand sides of the '=' must be different. The type must be stated.

```
for type1 %s1=items1  type2 %s2=items2  type3 %s3=items3 ... ;
  statements... ;
end ;
```

%s1, %s2, ...	The loop variables (for instance: strings).
items1, *items2, ...*	Any list of items.
Note that you may use parentheses `for( %s1=items1  %s2=items2 ... )`, like the IF statement.	

Date loop, FOR ... TO

A date loop loops through dates from a start date to an end date, with an optional stepsize. (To use logical conditions on individual observations inside timeseries, see the iif() function)

```
for date %d = date1  TO  date2  BY  step ;
  statements... ;
end ;
```

%d	The loop variable d (of date type).
date1	Start date (inclusive), can be expression (including integer value).
date2	End date (inclusive), can be expression (including integer value).
step	(Optional). An optional stepsize (default step: 1). Must be integer, and may be negative. You may omit `BY step` if not needed. You may use STEP instead of BY if preferred.
TO	You may use '..' (range) instead: for instance `for date %d = 2015q1 .. 2020q4;`
Note that you may use parentheses `for(date %d = date1  TO  date2  BY step)`, like the IF statement. If one or both of `date1` and `date2` are positive integers, they will be interpreted as annual dates. You cannot combine parallel looping with FOR ... TO syntax, but in that case consider using the `seq()` function (cf. "Examples (parallel loops)").	

Value loop, FOR ... TO

A value loop loops through values from a start value to an end value, with an optional stepsize. If the stepsize is negative, the values will decrement.

```
for val %v = val1  TO  val2  BY  step ;
  statements... ;
end ;
```

v	The loop variable %v.
val1	Start value. Can be any number or expression.
val2	End value. Can be any number or expression.
step	An optional stepsize (default step: 1). Can be any number or expression, and may be negative. Omit `BY step` if not needed. You may use STEP instead of BY if preferred.
TO	You may use '..' (range) instead: for instance `for val %v = 1 .. 100;`
Note that you may use parentheses `for(val %v = val1  TO  val2 BY step)`, like the IF statement. You cannot combine parallel looping with FOR ... TO syntax, but in that case consider using the `seq()` function (cf. "Examples (parallel loops)").	

Examples (list)

You may wish to use some sector codes to print out production values easily:

```
for string %i = nf, nz, qz, o;  //or: ('nf', 'nz', 'qz', 'o')
  prt fX{%i};
end;
```

This will print out the variables `fXnf, fXnz, fXqz, fXo` (one by one). You may use a pre-defined list after the '=' in the for statement `for string %i = #mylist;`, or a wild-card list (`for string %i = ['fx*'];`), or combinations of these.

Nested loop:

```
for string %i = a, b, c;  //or: ('a', 'b', 'c')
  for string %j = x, y, z;
    prt var{%i}o{%j};
  end;
end;
```

The loop prints 9 variables beginning with `varaox, varaoy, varaoz, varbox, varboy,` ... etc. Note that you can easily pre- and suffix list items, cf. the LIST statement.

Gekko can also loop over a list of values or dates, for instance:

```
option freq q;
for date %d = (2020q1, 2020q3);  //omitting the parenthesis will
not work for dates
  time %d %d+1;
end;
```

This will set the period 2020q1-2020q2, and afterwards 2020q3-2020q4. Note the parenthesis in the first line. Without it, the list will be understood as ('2020q1', '2020q3'), that is, two strings and not two dates.

You may loop two or more lists in parallel, cf. the examples in the section "Examples (parallel loops)".

Examples (dates range)

To compute the largest number of the variable `fX{%i}`, for the sectors `a, b, nf, qf`, over the period `%d1` to `%d2`:

```
%d1 = 1990;
%d2 = 2015;
#vars = a, b, nf, qf;
%max = 0;  //initialize
for string %i = #vars;
  for date %d = %d1 to %d2;
    if (fx{%i}[%d] > %max);
      %max = fX{%i}[%d];
      %dmax = %d;
      %imax = %i;
    end;
  end;
end;
tell 'Largest value in sector {%imax}, period {%dmax}, value = {%
max}.';
```

After this loop, the string `%imax` will contain the sector name with the highest number, the date `%dmax` will contain the period containing that number, and the value `%max` will contain the max number. It is assumed that the values are all positive, so that `%max` can safely start out with value 0.

This example sets the timeseries `y`, depending upon two timeseries `x1` and `x2`, over the period 2001-2003. For the observations where `x1 > x2`, `y` is set to `x0`, else to `%v` (a scalar).

```
for (date %d = 2001 to 2003);
  if (x1[%d] > x2[%d]);
    y[%d] = x0[%d];
  else;
    y[%d] = %v;
  end;
end;
```

Note that such conditional setting of values via time-looping can be done much easier with the iif() function:

```
<2001 2003> y = iif(x1, '>', x2, x0, %v);
```

You may loop over frequencies like this:

```
for string %i = a, q, m;
  option freq = {%i};
  xx = 100;
end;
```

After this, there will be series `xx!a`, `xx!q` and `xx!m`, corresponding to each of the frequencies a, q and m. Looping over days includes possible leap days (February 29):

```
for date %d = 2020m2d27 to 2020m3d2;
  tell %d.string();
end;
//prints out 2020m2d27, 2020m2d28, 2020m2d29, 2020m3d1, 2020m3d2
```

Examples (values range)

A value loop is similar to date loops

```
for val %v = 10 to 0 by -2.5;
  tell 'Value: {%v}';
end;
```

This will print out the numbers 10, 7.5, 5, 2.5 and 0.

```
for (val %v = 10 to 0 by -2.5)
  tell 'Value: {%v}';
end;
```

Equivalently, using parentheses (the semicolon in the first line may be omitted in this case). This is just to avoid an error if the user assumes the same syntax as the IF statement (which has mandatory parentheses).

Examples (parallel loops)

You may use parallel lists like this:

```
#m1 = a, b, c;  //or: ('a', 'b', 'c')
#m2 = x, y, z;
for string %i = #m1  string %j = #m2;      //or: for string %i = a,
b, c   string %j = x, y, z;
  tell '{%i}, {%j}';
end;
//Result:
//a, x
//b, y
//c, z
```

In contrast to the nested loop above (that ran the PRT statement 3*3 = 9 times), this loop only runs the TELL statement 3 times in all. The parallel loops is an easy way to loop two (or more) lists in tandem. It is easier to use than doing the same loop 'manually', like the code below (produces the same output):

```
#m1 = a, b, c;
#m2 = x, y, z;
for val %v = 1 to #m1.length();
  %i = #m1[%v];
  %j = #m2[%v];
  tell '{%i}, {%j}';
end;
```

Parallel looping only works over two or more lists of same length, and parallel loops do not allow the FOR ... TO syntax. However, for sequences, you may consider using the `seq()` function to easily create the lists:

```
for val %i = seq(0, 2)  date %j = seq(2020q3, 2021q1);
  tell '{%i}, {%j}';
end;
//result:
//0, 2020q3
//1, 2020q4
//2, 2021q1
```

If you need to use fractions or other values than integers, you could just define for instance `%i2 = %i/100` inside the loop to get 0.00, 0.01, 0.02.

Note

You may sometimes need to use an explicit type conversion from one scalar variable type to another. In that case, use the conversion functions val(), date() or string().

Related statements

END, STRING, DATE, VAL, IF

4.3.35 FUNCTION

FUNCTION is used to define user-defined functions. Such user functions may return a variable (if you need to return multiple variables, consider returning a map). For a function that does not return anything (a stand-alone function), you may consider using a procedure instead. A procedure is essentially the same as a user function with no return value.

Note that all Gekko functions (both in-built and user-defined) implement so-called UFCS so that a function like for instance `f(x, y)` can be written as `x.f(y)`, and `f(x, y, z)` can be written as `x.f(y, z)`.

You may decorate a user function with a <>-option field containing an optional time period. User-defined functions allow optional parameters with default values, and the function may prompt (ask) the user about these parameters, if `f?(...)` is used instead of `f(...)`, where `f` is the name of the function. See the LOCAL keyword if you need variables defined inside the function to be inaccessible/nonexisting outside the function.

Libraries
Beginning with Gekko 3.1.12, it is possible to organize user-defined functions and procedures in libraries. These are basically just .zip-files that are loaded with the LIBRARY statement.

Syntax

```
function type funcname(<date t1, date t2>, type1 var1 label1 = default1,
type2 var2 label2 = default2, ...);
  expressions... ;
end;
```

Unless the return type is `void`, the function body must contain at least one RETURN statement, returning a variable.

t1, t2	(Optional). You may state optional time period parameters inside <>-brackets, for instance `function series f(<date %t1, date %t2>, series x);` after which `%t1` and `%t2` are assigned to for instance 2020 and 2030 in the call `f(<2020 2030>, z)`. If the function is called without <>-brackets, for instance `f(z)` , the parameters `%t1` and `%t2` are assigned to the local/global time period instead. Using a <>-brackets in a function call does not in itself change the local time period inside the function: use for instance the BLOCK structure to do that. See examples.

type1, ...	Types of incoming and outgoing variables: `series`, `val`, `date`, `string`, `list`, `map`, `matrix`. You may also use the special `name` type for parameters, which behaves 100% as a `string` inside the function, but where the single quotes are omitted when calling the function from outside (the shorter call `f(y)` is used instead of `f('y')`). If the function does not return anything, `void` can be used as type.
var1, ...	The parameters/variables/expressions
label1, ...	(Optional). A label for the parameter, used if the function is promting (called with `f?(...)`). See more about prompting below.
default1, ...	(Optional). A default value for the parameter. If the parameter is omitted, the default value is used. If the function is asked to prompt (called with `f?(...)`) and the parameter is omitted, the default value is shown in the dialog box. In the dialog box, `Enter` or `Escape` will return the default value, and fire up the next dialog box (for the next optional parameter). If a `;` is entered in the dialog box, all the remaining parameters attain their default values, and no more dialog boxes are shown. For string input, the use of quotes (`'`) in the input box is optional. At the moment, only `val`, `date` and `string` types can be used for prompting input boxes.
funcname	The function name
body	The function body, that is, the statements to be performed. Use RETURN to return a variable. If several variables need to be returned, use a map or list to bundle them.

Tip: if you need to stop execution at a specific point/line to inspect variables etc., try inserting a STOP statement. This will abort from all called program files/procedures/functions, without executing anything more after the STOP (this is not the case regarding RETURN). Therefore, STOP can be practical for debugging, etc.

Example

Value examples, including multiple return values

The function square() below returns the input VAL squared.

```
function val sq(val %x);
  return %x*%x;
end;
```

```
//---------------
%y = sq(4);
%z = sq(sq(4));
```

So the VAL statement will produce a scalar value `%y = 16`, whereas `%z = 256` (the function calls may be nested).

Multiple variables may be returned, using a collection like for instance a map:

```
function map f(val %x, val %y);
  return (%sum = %x + %y, %product = %x * %y); //see definition of
a map
end;
//-----------------------------
#m = f(3, 7);
prt #m.%sum, #m.%product;  //10, 21
```

Date example

```
function date add3(date %d);
  return %d + 3;
end;
//-------------------------------
%d3 = add3(2000q3);
prt %d3;  //2001q2
```

String example

```
function string f(string %x);
  return %x + 'shine';
end;
//-----------------------
%y = f('sun');
prt %y;  //'sunshine'
```

If you prefer to omit the quotes when calling the function (that is, `f(sun)` instead of `f('sun')`, you may use the name type:

```
function string f(name %x);
  return %x + 'shine';  //%x behaves completely like a string
end;
//-----------------------
%y = f(sun);
prt %y;  //'sunshine'
```

List example

```
function val ncommon(list #x, list #y);
  #temp = intersect(#x, #y);
  return #temp.len();
end;
//----------------------
#m1 = x1, x2, x3, x4;
#m2 = x2, x4, x5, x6;
%v = ncommon(#m1, #m2);
prt %v;  //2 common elements
```

Series example

```
function series idx(series x, date %d);
  return x/x[%d];
end;
//--------------------------------------
time 2000 2010;
create x1; x1 = 10, 11, 12, 13, 11, 14, 16, 17, 15, 19, 20;
prt x1, idx(x1, 2002), idx(x1, 2008);  //index 2002=1 and 2008=1
```

The function idx() provides indexed values.

Stand-alone function call example (void type)

```
function void f(val %x1, val %x2, val %x3);
  tell 'Sum is: ' + (%x1 + %x2 + %x3);
end;
//-----------------------------------
f(1, -2, 3);  //result: 'Sum is: 2'. This is similar to a procedure
call
              //                     like "f 1 -2 3;" but without
the
              //                     problem of this being
interpreted as
              //                     "f (1-2) 3;", or "f -1 3;".
```

Combined example

```
function void load(string %n, string %label, date %d1, date %d2,
val %v);
  create {%n};  //must use {...}-braces to use as name.
  doc {%n} label = %label;
  {%n} <%d1 %d2> = %v;
  return;
end;
```

```
//-----------------------------------
load('extra1', 'Helper variable', 1980, 2020, 100);
load('vat', 'VAT rate', 1980, 2020, 0.25);
disp extra1, vat;
```

The load() function will create the two timeseries `extra1` and `vat`, both with labels, and values 100 and 120, respectively. Since the function does not return any variable, you may use a procedure instead.

Local period example

```
function series f(<date %t1, date %t2>);
  block time %t1 %t2;
    y = 100;
  end;
  return y; //return statement after the block ends
end;

time 2001 2001;
z1 <2002 2002> = f();             //z1 will be 100 in 2002
z2 <2002 2002> = f(<2003 2003>);  //z2 will be 100 in 2003
p <2001 2003 n> z1, z2;

// Result:
//                   z1             z2
//  2001              M              M
//  2002       100.0000              M
//  2003              M       100.0000
```

In the `z1` statement, it is seen how the local period <2002 2002> is used inside the f() function, by means of a BLOCK using the arguments `%t1` and `%t2`. The f() function itself is not called with time period, and since the time period is absent in the function call, `%t1` and `%t2` are assigned to the local period set outside of the f() function.

The `z2` statement illustrates a call of f() where a time period is present inside the f() function. This overrules the local time period 2002-2002.

Promt and default values example

Gekko user-defined functions allow default values, and prompting regarding these.

```
function val f(val %x1, val %x2 'parameter 2' = 1, val %x3
'parameter 3' = 2);
  return 10000 * %x1 + 100 * %x2 + %x3;
end;
%y1 = f(9, 3, 4); //--> 90304
%y2 = f(9, 3);    //--> 90302
%y3 = f(9);       //--> 90102
%y4 = f?(9, 3);   //enter 5 into the dialog box --> 90305
```

```
%y5 = f?(9);      //enter 6 and 7 into the dialog boxes --> 90607
%y6 = f?(9);      //enter 6 and ';' into the dialog box --> 90602
mem;
```

Beware that `f()` or `f?()` will fail with an error, since the first parameter is required. As shown regarding the last function call, you may terminate a sequence of input boxes with `;`, which means the default values are used for the current and following parameters. Pressing `Enter` or `Escape` returns the default value, and opens up the next input box. For prompt input, only the variable types val, date, string and name are supported at the moment (for name type, use for instance `... , name %x2 'parameter 2' = 'x', ...`).

Note

See also PROCEDURE. A procedure can be thought of as a function without return values. Procedures and user functions do not live in databanks, and are hence not affected by CLEAR, CLOSE, READ, etc., but are removed with RESTART or RESET.

If a function is defined without <>-brackets to indicate time, it may still be called with <>-brackets. In that case, the time period inside the brackets is just ignored.

Note that in Gekko 3.0, multiple return values are handled with maps. In Gekko's before 3.0, so-called tuples were used to the same effect (such tuples do not work anymore).

You can at most use 14 arguments, else use maps to bundle incoming arguments. Per default, all arguments are passed by value, not by reference (cf. `option system clone`). This means that functions cannot have so-called side-effects on the incoming arguments. Maps can be practical for bundling output variables.

It is planned to introduce the type `namelist` in addition to the `name` type, so that an argument like `(a, b, c)` can mean `('a', 'b', 'c')` internally.

Related statements

LIBRARY, LOCAL, PROCEDURE, RUN

4.3.36 GEKKO

Starting from Gekko 3.1.18, the GEKKO statement can be used to check if the currently running Gekko version is suitable regarding a particular Gekko program (`.gcm` file). For instance, you may want to ensure that the running Gekko version is `>= 3.1.18` to make sure that it is compatible with some new functionality introduced in Gekko 3.1.18. Alternatively, you may want to ensure that the running Gekko version is `< 4.0`, if we presume that a future Gekko 4.0 introduces some non-backwards-compatible changes.

You may state a range of Gekko versions, and you may use Gekko version dates, too (the date corresponding to `gekko.exe` of the running Gekko version). Checking the version date is mostly for beta-testers in between official Gekko (development) versions. If both a Gekko version and a Gekko date is indicated, the version number always trumps the date.

If the version check is ok, a brief message is printed, but if not ok, Gekko will abort with an error.

Syntax

```
GEKKO VERSION  operator  arg  arg ;
GEKKO VERSION  operator  arg  arg  andOr  VERSION  operator  arg  arg ;
```

operator	Choose between `<`, `<=`, `==`, `>=`, `>`, or `<>`. Note that non-equality is `<>`, not `!=`.
arg	Arguments: either string (for a Gekko version number) or daily date (for a Gekko date).
andOr	(Optional). Can be either `and` or `or`.

Examples

To check that the running Gekko version is `>= 3.1.18`, use the following in your Gekko program:

```
gekko version >= '3.1.18';
```

Note that the Gekko version number is stated as a string, and `gekko version >= 3.1.18;` will fail with a syntax error. A range of versions could be stated like this (note: Gekko 4.0 is not even planned as of March 2024):

```
gekko version >= '3.1.18' and version < '4.0';  //or just '4'
```

To check that the running Gekko version number is `>= 3.1.18` and has a `gekko.exe` with a date from at least `2024m4d4`, use this:

```
gekko version >= '3.1.18' 2024m4d4;
```

The official Gekko 3.1.18 was from `2024m4d4`. Testing for dates is mostly for beta testers in between official versions. In general, it is best to include Gekko version numbers and not only use dates, because file systems, version control systems, etc. may tamper with file dates (like for instance the date of `gekko.exe`).

Note

If a `gekko version` check is invalid, Gekko will stop with an error. If you remove or alter such checks, beware that even if your program now can run without syntax- or run-time errors, non-backwards-compatible changes in Gekko may change the results (just an example: in Gekko 2.x.x, the statement `import <csv> data.csv;` gets all data from a .csv file, whereas in Gekko 3.x.x, the same statement `import <csv> data.csv;` respects/limits the data according to the global time period).

For Gekko versions `<= 2.5.2` or in the range `[3.0 to 3.1.16]`, a `gekko version` statement will fail with a syntax error (but this syntax error should be pretty telling still). Gekko 2.5.3 understands a `gekko version` statement without date indication, and Gekko 3.1.18+ understands `gekko version` statements with both version numbers and dates. Gekko 3.1.17 had a bug regarding `gekko version` checks and is therefore not available for download anymore.

Related functions

gekkoVersion(), gekkoVersionDate()

Related statements

HELP, SIGN

4.3.37 GLOBAL

The GLOBAL statement is used to designate variable names that are to be located in the Global databank. Following a `global x;` statement, any subsequent use of `x` (without databank designation) will be understood as `global:x`.

After Gekko leaves the program file, function or procedure, these global variables live on in the Global databank. Therefore, using GLOBAL or `global:x = ...` can be practical regarding permanent storage of variables, for instance settings, without polluting the 'normal' databanks.

Use `global<all>;` to render all variables global. After a `global<all>`, you can still search for a bankless variable `x` outside of the Global databank by means of the special `all:` designation (for instance `y = all:x;`).

See the description of the OPEN statement regarding different types of databanks in Gekko.

See also the similar LOCAL statement, for local variables.

Syntax

```
global varnames;
global <all>;
```

varnames	Comma-separated list of variables
ALL	(Optional). With this option, all following (in the rest of the program/function/procedure) left-hand side variables without explicit databank designation are located in the Global databank. For a variable `x` that you would like to keep in another databank despite using a `global<all>`, you may use `first:x` or another bank designation to circumvent `global<all>`.

Examples

```
global x, %y, #z;
```

After this, any use of `x`, `%y`, or `#z` (in the present program file, function or procedure) will be interpreted as `global:x`, `global:%y`, and `global:#z`.

The Global databank is searched last, if databank searching is active (that is, data- or mixed mode), cf. databank search.

Variables in the Global databank survive for instance READ and CLEAR statements, and the Global databank is practical for storing long-term variables like setting etc. For instance:

```
global:%per1 = 2010;
global:%per2 = 2050;
global:%path = 'm:\data\scenario2';
global:%unit = 1000;
```

As long as Global is not cleared explicitly (or a RESET or RESTART is issued), `%per1`, `%per2`, `%path`, and `%unit` would be available. In data- or mixed mode, you can just refer to for instance `%per1`, provided that there is no `%per1` located in other open databanks. If you want to be absolutely sure that the variable is taken from Global, you can use `global:%per1` to refer to the variable.

To avoid all the `global:` indicators, you may consider this alternative, using a procedure for the global settings:

```
reset;
procedure globals;
  global<all>;
  %per1 = 2010;
  %per2 = 2050;
  %path = 'm:\data\scenario2';
  %unit = 1000;
end;
globals; //call the procedure
//
// the rest of the program here
//
```

Note

You are not forced to use the GLOBAL keyword, when operating with global variables. Defining `global:%per1 = 2010;` first, and referring to `global:%per1` later on is possible, too. In that sense, the GLOBAL keyword is just for convenience, especially if `%per1` is used several times.

Variables in the Global databank are practical for settings, etc. These variables survive READ, CLEAR, etc., and do not 'pollute' the first-position databank if this is later on written to file.

Note that the Local or Global databanks are always searchable, independent on MODE etc.

Related statements

LOCAL

4.3.38 GOTO

GOTO can be used to transfer execution to some other point (TARGET) in the program.

You should mostly use this statement to jump out of loops (cf. the example below). It is not intended for jumping around in plain non-looping code, where the presence of GOTO/TARGET may render the programs slow-running and hard to read.

Syntax

goto name;

The label must be name-like, that is, alphanumeric characters including underscore (and not starting with a digit). You can not use scalars or expressions etc. as labels.

Examples

```
%sum = 0;
for val %i = 1 to 5;
  if(%i == 4);
    goto lbl1;
  end;
  %sum += %i;
end;
target lbl1;
```

This example skips the iterations before the fourth iteration is about to be executed. The value of `%sum` will be 6 (= 1+2+3, not 1+2+3+4+5).

The example below is NOT what the statement is intended for:

```
tell 'a'; goto x1;
target x2; tell 'c'; goto x3;
target x1; tell 'b'; goto x2;
target x3; tell 'd';
```

This prints 'a', 'b', 'c', 'd', but please use other means to organize the flow of your gcm-file!

Note

Target names cannot be duplicated. An error will be issued.

The program will also fail with an error, if the label does not exist. But 'orphaned' labels are accepted (a TARGET without a corresponding GOTO).

You cannot call a target inside a loop, from outside the same loop. For instance, the following will fail, and an error will be issued:

```
%sum = 0;
goto lbl1;
for val %i = 1 to 5;
  target lbl1;
  %sum += %i;
end;
```

Eternal loops may be accidently created, for instance the line `target lbl1; goto lbl1;` will run forever. This example is easy to spot, but such problems may arise if the GOTO structure is misused. It has been proven that the GOTO statement is technically superflous, and it can lead to so-called spaghetti code (cf. Dijkstra: "Go To Statement Considered Harmful").

At a later point, BREAK and CONTINUE might be added to Gekko loops, too.

Related statements

TARGET

4.3.39 HDG

HDG (heading) will put the heading into a databank file. This only works for .gbk files.

Syntax

hdg *heading* ;

heading	A string

Examples

Putting a heading on a databank can be useful:

```
hdg 'Bank for multiplier analysis, simulated 2010-2050';
write mulbank ;
read mulbank ;
```

When reading the .gbk databank, info like this is printed on the screen:

```
 Info     : Bank for multiplier analysis, simulated 2010-2050
 Date     : 26-10-2011 11:13:31
```

The heading will also be shown in the databank list (F2 button).

Databank files in .gbk format can contain meta-information like headings and date and time when written.

Related statements

WRITE, READ

4.3.40 HELP

The HELP statement (or F1) provides access to Gekko help system. Through the HELP statement it is possible to get quick help on a statement and its syntax, examples, etc.

The help system opens up in a separate window and is of the type "Compiled HTML Help" (stored in a .chm file). The help system is browsable/searchable. It is typically not possible to open the .chm from a network drive. Per default Gekko copies the .chm file to a temporary folder on the user's hard disk, in order to avoid this problem.

The help files are also available online here. The online version is not updated as often as the inbuilt version (.chm).

Syntax

```
help  statement;
```

statement	(Optional). The statement on which help is needed. If the statement does not exist, the help system will indicate that the file is missing. Opening up with just `help;` is possible, but in that case pressing F1 is easier. Using the menu: 'Help' --> 'Gekko help file' is also possible.

Examples

If, for instance, you are in doubt about the syntax regarding the SERIES statement, you may look directly for this topic in the help file:

```
help series;
```

If you cannot remember that the name of the relevant statement is SERIES (for instance), you may write

```
help;
```

In the section "Gekko statements", there is a page called "Statement overview" where the statements are grouped by categories. Else, the .chm help system is also searchable, cf. the "Search" tab.

Related options

OPTION folder help = [empty];
OPTION interface help copylocal = [yes|no];

Related statements

OPTION

4.3.41 IF

The IF statement is used for conditional execution of different blocks of statements. The IF statement works with strings, dates, values (or for instance single timeseries observations like `x[2010]`). In addition, the IF statement also supports comparing two timeseries.

Guide: conditional statements
For an easier introductory guide on conditional statements in Gekko, see this page.

You may sometimes need to explicitly convert the variables in order to compare them (by means of the functions val(), date() or string()).

The IF-statements work with operators like for instance `if (%x == 100); ...; end;`, testing if `%x` has the value 100. But IF-statements also works with single values, like `if (%x); ...; end;`. In that case, the statements are not executed if `%x` has the value 0, and are executed otherwise.

Two series may be compared using IF, for instance `if (x > y); ...; end;`. This IF will be true if -- for all observations in the global time period -- all of the observations of x are larger than y. Statements involving values are also allowed, for instance `IF (x > 0); ...; end;` which is true if all observations of x (inside the global time period) are positive. If you are comparing two values or series that contain missing values, special rules apply: see this section.

If you need to perform IF-like operations *inside* a SERIES, you may use $-conditionals on expressions, or the iif() function. See examples in the SERIES section.

Syntax

```
if (expression1);
  statements1... ;
elseif (expression2);
  statements2... ;
else;
  statements3... ;
end ;
```

expression	A logical expression involving strings, dates or values (or single timeseries observations like `x[2010]`), in addition to the logical operators `AND`, `OR`, `NOT`, `<`, `<=`, `==`, `>=`, `>`, `<>`. Please note that logical equivalence uses `==` (and not `=` which is assignment) , and that you must use `<>` for logical difference, not for instance `!=`.

	If the expression is a scalar value, the statements are not executed if the scalar value is 0, and are executed otherwise.
statements1	Gekko statements to be executed if *expression1* is true.
statements2	Gekko statements to be executed if *expression1* is false and *expression2* is true. (You may use as many ELSEIF's as you please).
statements2	(Optional). Gekko statements to be executed if *expression1* and *expression2* are both false.

Example

See more examples involving IF in loops in the FOR help file. A very simple example using the string scalar variable.

```
%write = 'yes';
if (%write == 'yes');      //note the use of '==', using '=' here
will fail
  tell 'Yes chosen';
else;
  tell 'No chosen';
end;
```

To sift through more choices, you may use IF, ELSEIF, ELSE and END like this:

```
%windows = '<type something...>';
if (%windows == 'xp');
  tell'Windows XP was released in 2001';
elseif (%windows == 'vista');
  tell'Windows Vista was released in 2007';
elseif (%windows == '7');
  tell'Windows 7 was released in 2009';
elseif (%windows == '8');
  tell'Windows 8 was released in 2012';
elseif (%windows == '10');
  tell'Windows 10 was released in 2015';
elseif (%windows == '11');
  tell'Windows 11 was released in 2021';
else;
  tell 'Could not recognize the Windows version "{%windows}"';
end;
```

Inside the IF or ELSEIF parentheses you may use logical conditions like AND or OR:

```
%v = 2000;
%s = '2000';
date %d = 2000;  //without 'date' it becomes a value
if (%v == val(%s) AND %v == val(%d) AND date(%s) == %d)
  tell 'Ok';
else;
  tell 'Not ok';
end;
```

This will print 'Ok'. Note that you have to explicitly convert the variables to be able to compare them, otherwise you will get an error. In this case, the conversion functions val() and date() are used. The conversions may seem obvious and superfluous here, but consider this example:

```
%s = '100';
%s1 = '33';
%v = 133;
if (%v == %s + %s1)   //Will give type mismatch error
  tell 'Ok';
end;
```

This gives an error, because Gekko does not know how to compare the value 133 with the string '10033' (the sum of the two strings taken literally). So

```
if (%v == val(%s + %s1))        //will be false, comparing 133 and
10033
```

or this:

```
if (%v == val(%s) + val(%s1))   //will be true, comparing 133 and
133
```

To avoid ambiguities, the type system in Gekko is quite strict. To access individual observations from a series in the first-position databank, use the `variable[period]` syntax:

```
time 2010 2012;
//create data; //use this in sim-mode
data = 1, 2, 3;
date %d = 2011; %v = 2;
if(data[%d] == %v) tell 'Ok'; end;
```

This will print 'Ok'. To compare and transform timeseries depending upon individual observations, see the iif() function.

The $-conditional can often be used instead of IF, for instance:

```
if(x[2015] == 100);
  y = 2;
else;
  y = 0;
end;
```

This can be stated in the following simpler way, using the $-conditional. You may also use the similar iif() function.

```
y = 2 $ (x[2015] == 100);
```

To compare two series, or a series and a value, use the following:

```
time 2001 2003;
x1 = 2, 3, 4;
x2 = 1, 2, 3;
//the condition must be true for *all* observations in the global time period
//for the IF statement as such to be true
if (x1 > x2); tell 'larger'; else; tell 'not larger'; end;
if (x1 > 1); tell 'larger than 1'; else; tell 'not larger than 1'; end;
//try changing x1 to x1 = 1, 3, 4 and run the two IF statements again
```

Note

- Note the use of two equal signs (`==`) in IF-statements, and `<>` for logical difference.
- There is also an `exist(x)` function that returns 0 or 1 depending upon whether the series `x` exists or not.
- See the GAMS-inspired $-condition or the iif() function for logical conditions inside timeseries observations.
- You may ask a list if it has a particular member (will return 1 for true, and 0 for false). For instance `'a' in #m` is true if `#m` contains `'a'`. Therefore, you may for instance use if(`'a' in #m`) without logical operator. The alternative syntax `#m.contains('a')` is equivalent.

Related statements

ELSEIF, ELSE, END, FOR

4.3.42 IMPORT

The IMPORT statement merges data (typically series) from an external file into the first-position databank. The IMPORT statement is primarily for non-.gbk files, and it should be noted that IMPORT without options restricts data to the global time period, it only puts data into the first-position databank, and it merges data with any pre-existing data in the first-position databank.

Import supports collapsing (aggregating) data points of high frequency into annual, quartery, monthly, weekly or daily series, cf. the <collapse> option.

Compatibility note: If a time period is not indicated in the <>-option field, Gekko 3.0 will only import data inside the global time period. Before Gekko 3.0, all data would have been imported. To emulate previous behavior, you can use IMPORT<all>. Alternatively, you may set "OPTION bugfix import export = yes;". If the option is set, IMPORT and EXPORT will work as in pre-3.0 versions. The option will be removed at some point, so it is better to change occurrences of date-less IMPORT to IMPORT<all> in old Gekko program files.

Tabular formats note: When using IMPORT with xlsx, csv or prn files, it is advised to first set the global frequency (`option freq = ...`) to the frequency of the data file (temporary frequency change can be done with `block freq ...; import ... ; end;`). With DATEFORMAT and DATETYPE at their default values, dates like `1990a1`, `1990y` or `90` are treated as annual `1990`. A date like `199003` is treated as `1990q3` or `1990m3`, if global frequency is set to `q` or `m`. A date like `19900325` is always understood as daily `1990m3d25`. To import data with undated (`u`) frequency, the global frequency should be set to `u` first.

IMPORT is intended for non-.gbk files, and can be thought of as a soft version of READ. In contrast to READ, IMPORT does not clear the first-position databank (instead it merges data), it only imports data corresponding to the global time period (unless a time period or <all> is used), and it does not alter the Ref databank. There are the following equivalences: IMPORT = READ<first merge respect>, and the inverse: READ = CLEAR<first> + IMPORT<all> + CLONE.

Because Gekko uses a so-called cache for databank files, the same file will load faster the next time it is imported.

Syntax

```
import < period  format  ALL  COLS  REF  SHEET=... CELL=...
 NAMECELL=...  DATECELL=...  COLLAPSE=...  METHOD=...  DATEFORMAT=...
DATETYPE=...  ARRAY  VARIABLECODE >  filename TO bankname;
```

period	(Optional). Without a time period indicated, Gekko will import all

	the data for all observations. When a period is indicated, the databank is time-truncated.
format	(Optional). Choose between AREMOS, CSV, FLAT, GDX, PCIM, PRN, PX, TSD, TSP, XLS, XLSX. • AREMOS: If you need more precision (significant digits) from AREMOS than the TSD format can provide, you may use AREMOS dump files. See example. • CSV: Comma-separated file. Tabular format with rows/cols consisting of names and dates. The global frequency (OPTION freq) must correspond to the frequency in the file. • FLAT: A Gekko-specific text-based format with lines that resemble Gekko series statements. The format is: variable name + start date + end date + numbers. These items are separated by blanks, for instance "x 2020 2022 1.5 -2.5 3.5". This corresponds to `x <2020 2022> = 1.5, -2.5, 3.5;`. If only one number is given, it will be used for the full time period. You can use 'm' or 'm()' to indicate a missing value. Blank lines and lines beginning with '//' are ignored. This format reads much faster than 'real' series statements (which have to be parsed and compiled before the values are extracted). • GCM: Gekko can export timeseries as a .gcm file, using `export<gcm>`. To "import" such a .gcm file, just RUN it (or copy-paste the contents into your preferred .gcm file). Note that .flat files are similar, but load much faster. • GDX: A binary GAMS-database. Note `option gams exe folder = ...` where it is possible to point to the exact GAMS folder (otherwise the system will try to auto-locate GAMS). Even without the option set, the auto-locator seems pretty good at locating a 32-bit GAMS for a 32-bit Gekko, and a 64-bit GAMS for a 64-bit Gekko version. Please note that the data is read as array-timeseries (see SERIES), and that Gekko only reads variables, parameters, sets (as Gekko lists) and domains. GAMS can be freely downloaded as a demo, and the demo will work fine regarding Gekko IMPORT. Default options are `option gams time set = 't'; option gams time prefix = ''; option gams time offset = 0; option gams time detect auto = no;`. This corresponds to time having the set name 't', with natural values, for instance 2020, 2021, etc. Default GAMS read is using a fast reader (low-level API). If this poses problems, try the more robust normal API by setting `option gams fast = no;`. See more under OPTION. • PCIM: A binary PCIM databank. • PRN: The first item in the prn format must be either 'date' or 'name' to indicate the orientation. The global frequency (`option freq = ...`) must correspond to the frequency in the file. • PX: Imports a PC-Axis file. See also the <array> option. See more info regarding the px format and how Gekko reads it under

	the DOWNLOAD statement. Note that the time element of the .px file (CODES("tid") or CODES("time")) should preferably be stated last (this is to keep the .px data file organised in the time dimension, which makes it load faster in Gekko). See also the `array` and `variablecode` options: specifically it is advised to use the `variablecode` option in order to get shorter timeseries names in the non-array case (these will correspond to what you get with direct DOWNLOAD). • SDF: Imports 'sdf' format. The format is a tab-separated 'table' without header row, with period in the second-last column, and value (with decimal comma) in the last column. • TSD: For interchange with AREMOS and others. • TSP: Imports TSP records. • XLS and XLSX: Tabular format with rows/cols consisting of names and dates. See SHEET<import> to see how to load a spreadsheet as a nested list of "cells" and scrape data from these. The global frequency (`option freq = ...`) must correspond to the frequency in the file. The engine used for Excel reading can be changed with `option sheet engine = ...;`.
filename	Filenames may contain an absolute path like `c:\projects\gekko\bank.gbk`, or a relative path `\gekko\bank.gbk`. Filenames containing blanks and special characters should be put inside quotes. Regarding reading of files, files in libraries can be referred to with colon (for instance `lib1:bank.gbk`), and "zip paths" are allowed too (for instance `c:\projects\data.zip\bank.gbk`). See more on filenames here. If the filename is set to '*', you will be asked to choose the file in Windows Explorer. The extension .gbk is automatically added, if it is missing.
ALL	(Optional). With this option, all observations are imported, regardless of the global time period. This corresponds to pre-3.0 Gekko behavior.
REF	(Optional). Reads the file into the reference databank (shown as REF on the F2 window list). Note that the Ref/reference databanks does not show up in the F2 window if it is empty.
COLS	(Optional). For .csv or Excel files, this indicates whether the timeseries are running downwards in columns. Note that for .prn files, you indicate this in the first 'cell' (date/name).
TO	(Optional). If "TO *bankname*" is indicated, Gekko will put the data into a seperate 'named' databank alongside the Work and Ref databanks. For instance, after `import <xlsx> adambk TO a;`, you

	may refer to the variables by means of colon, for instance `prt a:var1;`. If you use `import <xlsx> adambk TO *;`, the bankname will be the same as the file name. It should be noted that the databank will be read-only (non-editable) when opened like this (this functionality is a subset of the OPEN statement)
ARRAY	(Optional). Regarding the PX format, if this option is set, Gekko will put the data into array-timeseries rather than normal timeseries (for the GDX format, Gekko always puts into array-timeseries per default). The `variablecode` option will not matter if `array` option is used.
VARIABLECODE	(Optional). Regarding the PX format, if this option is set, Gekko will use shorter variable names codes, for instance `VAREGR` instead of `varegruppe`. Using this option is recommended, and the DOWNLOAD statement uses it implicitly. The `variablecode` option will not matter if `array` option is used.
SHEET=	(Optional). The name of the sheet for your data, for instance 'Data1'
CELL=	(Optional). For Excel files: the first cell of the data section. Defaults to 'B2'. This allows for rows/columns between the dates, names and data cells (cf. `DATECELL` and `NAMECELL`).
DATECELL=	(Optional). For Excel files: the first cell of the dates labels. Calculated from `CELL` location if not provided. This allows for rows/columns between the dates, names and data cells (cf. `CELL` and `NAMECELL`).
NAMECELL =	(Optional). For Excel files: the first cell of the names labels. Calculated from `CELL` location if not provided. This allows for rows/columns between the dates, names and data cells (cf. `CELL` and `DATECELL`).
COLLAPSE=	(Optional). For Excel files with Excel-dates that are going to be collapsed, this option can be set to either `a`, `q`, `m`, `w` or `d`, and indicates the frequency that the data points are being collapsed into. A data point is an Excel date and a corresponding value, for instance 24-Dec-2010 with the value 123.45. In your Excel version, this date might be shown as 24/12/2010 (British English) or 12/24/2010 (US English) or in other formats, but internally there is no confusion, since the Excel dates are stored as values (technically the number of days since January 1, 1900). See the collapse example below.

METHOD=	(Optional: default = total). For Excel files using COLLAPSE, the METHOD option sets how the collapse (aggregation) is performed. Choose between total, avg, first, last, count, cf. also the COLLAPSE statement. Use "method=count" to check that data is being collapsed as desired, and note that "method=avg" amounts to "method=total" divided by "method=count". After collapsing into monthly or quarterly timeseries, X12A may be used for seasonal adjustment.
DATEFORMAT= DATETYPE=	(Optional). These options control the date format for .xlsx and .csv files, and for the use of SHEET in Gekcel. `DATEFORMAT` can be either `'gekko'` (default, for instance `2020q3` or `2020m11`) or a format string like `'yyyy-mm-dd'`, and the latter may contain a `first` or `last` indicator, for instance `'yyyy-mm-dd last'`, which indicates for quarterly, monthly or weekly data that the *last* day of the quarter/month/week is used. `DATETYPE` can be either `'text'` or `'excel'`. In the former case, the dates are understood as text strings (for instance `'2020q3'` or `'2020-09-30'` for a quarterly date), and in the latter case (not relevant for .csv files), the date is understood as an Excel date, which basically counts the days since January 1, 1900. This number would correspond to 44104 for the date 2020-09-31, and can be shown in Excel in different ways depending upon date format settings, language settings, etc., but the internal number itself is unambiguous. When using SHEET in Gekcel, using `DATETYPE='excel'` is advised for consistency. [New in 3.0.5].

- If no period is given inside the <...> angle brackets, the global period is used (cf. TIME).

Examples

Reading data from the file data.xlsx (spreadsheet) can be done with:

```
import <xlsx> data;
```

or by the following:

```
import <xlsx> *;
```

and then selecting the file. You can use paths etc.:

```
import <tsd> otherbanks\adam3;
```

This will look for adam3.tsd in the subfolder 'otherbanks', relative the the Gekko working folder.

Use the TO keyword like this:

```
import <xlsx> forecst2 TO f2;
```

This reads forecst2.xlsx into the named databank `f2`. After this, you may use for instance `prt f2:gdp;` to print out the timeseries gdp from this databank. You may use `import <xlsx> forecst2 TO *;` if you wish to use the filename as databank name. It is possible to use for instance `import <xlsx> * TO *;`.

Using IMPORT for csv or Excel files is only implemented for 'well-formed' spreadsheets. That is, with data starting in the first column and first row, and with either timeseries running left-to-right (normal for .csv files) or downwards (less normal). You may use `import<csv cols>` or `import<xlsx cols>`, if the timeseries are running downwards. If you need to pick out data from Excel cells more arbitrarily, see the SHEET<import> statement.

Example, collapse

Excel data may be collapsed from higher frequencies than daily (for instance hourly observations), if the dates are represented as 'Date' types in Excel. Example:

```
import <xlsx sheet='oil' collapse=m> highfreq.xlsx;
```

The Excel sheet might look like this:

	19-1-2011	20-1-2011	21-1-2011	24-01-2011	25-01-2011
oil_crude	1	2	3	4	5
oil_refined	2	4	6	8	10

Using this, it is expected that the timeseries run row-wise, with data starting at cell B2. Hence, the first date should be found at B1, and the dates should continue at cells B2, B3, etc. The first name should be at cell A2, and the names should continue at cells A3, A4, etc. In the example, the timeseries will be collapsed into monthly frequency, in the following way. For each date in the dates row, the month of this particular date is found, and the data is put into that particular month for each timeseries. Since the default method is 'total', the data is being summed for each month. You may use "collapse = q" or "collapse = a" to collapse directly into quarterly or annual data (this is better and simpler than using the COLLAPSE statement on the resulting monthly timeseries). For daily data, you may alternatively read these directly with normal IMPORT, and later use COLLAPSE to collapse them into monthly timeseries.

If needed, the X12A statement can then be used for seasonal adjustments. If you only want to obtain parts of the timeseries, you may restrict with a time period, for instance:

```
import <2000m1 2018m5 xlsx sheet='oil' collapse = m> highfreq.xlsx;
```

If you prefer averages, use "method=avg":

```
import <xlsx sheet='oil' collapse=m method=avg> highfreq.xlsx;
```

To check that there is a similar number of data points for each month, you may use "method=count" and print/plot the resulting series to check this (particularly relevant regarding the start and end of the range of Excel dates). As noted above, "method=avg" amounts to "method=total" divided by "method=count".

The data does not need to start at cell B2. If, for instance, the first data cell is at G10, you may use:

```
import <xlsx sheet='oil' cell='g10' collapse=m> highfreq.xlsx;
```

Here, Gekko will expect the first date cell to be at G9, and the first name cell to be at F10. If there are rows/cols between the dates/names and the data cells, you may indicate the precise location of the dates/names:

```
import <xlsx sheet='oil' cell='g10' datecell='g1'  namecell='a10'
 collapse=m> highfreq.xlsx;
```

In this particular case, the dates/names are located in the first row and column of the spreadsheet. If the timeseries run downwards in columns, you may use <cols> for transposed importing:

```
import <xlsx cols sheet='oil' collapse=m> highfreq.xlsx;
```

Note: when using this functionality, you may 'collapse' for instance monthly data into its own frequency. In that case, using <method=count> should produce timeseries with value 1, indicating that there is only 1 observation for each month (otherwise something is wrong regarding the Excel sheet). Note also that dates in Excel are represented as the number of days since January 1, 1900. These dates may contain fractions, so 1 hour is represented by 1/24, etc. Keep this in mind if you are using <datecell=...>. If this points to a sequence of numbers that are not dates, these numbers may be erroneously interpreted as dates!

Example, AREMOS dump files

You may use a setup like the following to transfer timeseries with high precision from AREMOS (the tsd format has rather limited precision).

```
!In AREMOS

procedure dump1 list m;                    !names without .bnk
extension
  set report columns 35 decimals 20;
  set databank limit 25000;                !make sure there are no
'Search truncated by limit' warnings
  for i = #m;
    close *; clear;
    open #i;
    dump * #i|.dmp;
    close *; clear;
  end;
end;

list banker = bank1, bank2, bank3;
dump1 #banker;
```

This will produce the three files bank1.dmp, bank2.dmp, and bank3.dmp.

```
//In Gekko

reset;
global:#i = bank1, bank2, bank3;
time 1900 2500;
for string %i = #i;
  clear work;
  import<aremos>{%i}.dmp;
  write {%i};
end;
```

This will produce the three files bank1.gbk, bank2.gbk, and bank3.gbk. Only timeseries can be transferred like this.

Note

To convert a .tsd file or other formats into a .gbk file, just import it with IMPORT<tsd>, and WRITE it. Please note that a .tsd file operates with 8 significant digits (or less), so there will typically be a loss of precision compared to a .gbk file (which is in double-precision).

The option `copylocal` below copies the targeted file to a temporary file on the user's local hard disk before reading. This copying is typically very fast, and afterwards reading the temporary file is faster and more reliable, if the targeted file is located on a network drive. In general, this is a recommended option that alleviates some potential network problems.

The `cache = nongbk` option is recommended, too. It does the following: (a) it finds the data file, (b) it optionally copies it to the user's local hard disk, and (c) it calculates a MD5 hash code from the file. If this hash code is already known, Gekko uses a cache file representing the databank data, which in most cases is much faster to read (it is in protobuf format: the same format that is used for .gbk files). So there is a small speed penalty for calculating the hash, and a large speed gain for using the protobuf/cache file.

See the Gekko menu 'Options' --> 'Program dependency tracking' or use `option global dependency tracking = ...;` to activate dependency tracking, so that the use of external files (for instance program files, read/written databanks etc.) are shown as a list at the end of a Gekko session.

Related options

OPTION databank file cache = nongbk; [all|nongbk|none]
OPTION databank file copylocal = yes;
OPTION folder bank = [empty];
OPTION folder bank1 = [empty];
OPTION folder bank2 = [empty];
OPTION gams exe folder = [empty];
OPTION gams fast = yes;
OPTION gams time set = 't';
OPTION gams time prefix = '';
OPTION gams time offset = 0;
OPTION gams time detect auto = no;
OPTION gams trim = 0; //can ignore variables/parameters with few elements.
OPTION sheet engine = internal; //use 'excel' for the older .xls format

Related statements

READ, WRITE, EXPORT, OPEN, CLONE, DOWNLOAD, COLLAPSE

4.3.43 INDEX

The statement is used to search for variables in databanks, using wildcards or ranges. The result of the search may be put into a list. When indexing timeseries, remember that default INDEX only finds series of the currently active frequency from the first-position databank. You may use `:` and `!` to augment, for instance `index *:*!*;` to find all timeseries of all frequencies from all databanks.

Note that 'naked' wildcards are allowed in this statement, so you may for instance use `a*b?c` as wildcard instead of the more cumbersome `{'a*b?c'}`.

If one or more of the databanks contains many variables, this output may become voluminous. Use INDEX<mute> or COUNT if you prefer to avoid the output. Wildcard logic, including double and triple stars etc. (`index **;` or `index ***;`), is explained more generally on the wildcards page.

The INDEX statement works for array-subseries too, for instance `index x[a*b?c];` to get the matching subseries of the 1-dimensional array-series `x`. Multidimensional indexing follows the same logic, for instance `index y[b*, *z];`. To match all subseries regardless of the number of dimensions, you may use double stars: `index y[**];`.

For convenient searching of variable names you may use the keyboard keys `[Tab]` or `[Ctrl+Space]` to get autocompletion on timeseries names (cf. here). For instance you may type `prt a*b?c` and hit `[Tab]` or `[Ctrl+Space]` to select from a list of matching series names.

Syntax

```
index <MUTE  BANK=...  SHOWBANK=...  SHOWFREQ=... > type  wildcards  TO
 listname ;
```

MUTE	(Optional). If set, Gekko will not print the list of found items on the screen. The COUNT statement is essentially an INDEX<mute>.
BANK=	(Optional). A databank name indicating where the variables are to be located.
SHOWBANK =	(yes|no|all), default = 'yes'. If this option is 'no', banknames are not included in the items. If the option is 'all', banknames are always included in the items. If the option is 'yes' (default), banknames are included in the items, except if the bankname is the same as the first-position databank.

SHOWFREQ =	(yes|no|all), default = 'yes'. If this option is 'no', frequencies are not included in the items. If the option is 'all', frequencies are always included in the items. If the option is 'yes' (default), frequencies are included in the items, except if the frequency is the same as the current frequency.
type	(Optional). Restrict the type of variables.
wildcard	The variables to be searched for. You may use banknames to indicate a particular bank, and you may separate the wildcards with commas. In general, wildcards are of the form `a*x` to find all variables starting with `a` and ending with `x`, or `a?x` to match only one character. You may use syntax like `x[a*, *b]` for array-series.
listname	(Optional). The list name where the result is stored. The listname may be for instance `#m`, or `#(listfile m)`. The list is always a list of strings (names of variables), not the objects themselves.

- If a variable is stated without databank, the databank is assumed to be the first-position databank.

The following provides a list of all variables in all databanks, including banknames and frequencies (beware, this output may be voluminous if the databanks are large):

```
index <showbank=all showfreq=all> ***;                 //all
variables in all banks
index <showbank=all showfreq=all> *:**;                //the same
index <showbank=all showfreq=all> *:%*, *:#*, *:*!*;  //the same
```

A string (or list of strings) representing variable names may be manipulated by means of Gekko's inbuilt functions to handle these. Variable names here include bank, frequency, indexes, etc., and examples of such functions could be setBank(), removeBank(), replaceBank(), setFreq(), removeFreq(), setNamePrefix(), etc. There are many more of such functions, see the functions section, under ‘Bank/name/frequency/index manipulations’.

For instance, if you have a list `#m = ('x', 'y');`, you may use `prt {#m};` to print out `x` and `y`, `prt {#m.setBank('b')}`; to print out `b:x` and `b:y`, or `prt {#m.setFreq('q')};` to print out `x!q` and `y!q` (here, `prt b:{#m};` and `prt {#m}!q;` will work, too).

Examples

The following INDEX statement will look for timeseries beginning with 'f' in the first-position databank (and with the current frequency), and put the result into `#m`.

```
reset;
fa = 1; fb = 2; fc = 3;
index f* TO #m;      //result: 'fa', 'fb', 'fc'
prt #m;              //prints the three strings
prt {#m};            //prints the three series
```

INDEX will print the list of found variables, unless the <mute> option is used. To look for the same pattern/wildcard in the Ref databank:

```
reset;
ref:fa = 1; ref:fb = 2; ref:fc = 3;
index ref:f* TO #m;      //result: 'ref:fa', 'ref:fb', 'ref:fc'
prt #m;                  //prints the three strings
prt {#m};                //prints the three series
```

Here, the bankname is included, since Ref is not the first-position databank. You may search in all banks like this:

```
reset;
fa = 1; fb = 2; fc = 3;
clone;
index *:f* TO #m;        //result: 'fa', 'fb', 'fc', 'ref:fa',
'ref:fb', 'ref:fc'
prt #m;                  //prints the six strings
prt {#m};                //prints the six series
```

Instead of the above, you could alternatively use this:

```
reset;
fa = 1; fb = 2; fc = 3;
clone;
#m = ['*:f*'];           //result: 'fa', 'fb', 'fc', 'ref:fa',
'ref:fb', 'ref:fc'
prt ['*:f*'];            //prints the six strings
prt {'*:f*'};            //prints the six series themselves
disp *:f*;               //also prints them: DISP does not need
{'...'}-syntax for wildcards.
```

In the light of the above example, the reader may ask: why use the INDEX statement at all? The answer is three-fold:

- Often, one would just like to see the result of a wildcard search, without putting the result into any list. To that end, for instance `index *:f*;` is practical.
- In addition to the above, for INDEX, the `['....']` part of the wildcard can often be dropped, using the shorter `index *:f*;` instead of using `{'*:f*'}` or `['*:f*']`. The shorter notation does not work generally for all statement types, for instance

`#m = *:f*;` or `prt *:f*;` would fail with an error. Do not expect "naked" wildcards to work in statements that accept mathematical expressions.
- The INDEX statement provides options regarding the search. For instance, `index string %*;` only matches string scalars. You may also indicate the bank in which the variables are searched, and you may indicate how you want bank and frequency information shown in the resulting list.

For instance:

```
reset;
fa = 1; fb = 2; fc = 3;
clone;                         //copies the series into the Ref
databank
index {'*:f*'};                //just prints 'fa', 'fb', 'fc',
'Ref:fa', 'Ref:fb', 'Ref:fc'
index *:f*;                    //same, shorter
index <bank=ref> f*;           //same as ref:f*, result = 'Ref:fa',
'Ref:fb', 'Ref:fc'
index <showbank=no> *:f*;      //omits banknames, result = 'fa',
'fb', 'fc', 'fa', 'fb', 'fc' (note dublets)
index <showbank=all> *:f*;     //all banknames, result = 'Work:fa',
'Work:fb', 'Work:fc', 'Ref:fa', 'Ref:fb', 'Ref:fc'
index <showfreq=all> *:f*;     //all freqs, result = 'fa!a', 'fb!a',
'fc!a', 'Ref:fa!a', 'Ref:fb!a', 'Ref:fc!a'
index string {'%*'};           //indicate type of variable, will
find all strings in first-position bank
```

Matching rules: the wildcard `*` does not just match any variable. The wildcard `*` does not match starting characters `%` or `#` at the start of the variable name, nor does it match any frequency indicators at the end of the variable name (for instance `!a` or `!q`). So the following rules apply:

- `*` matches all series of the current frequency in the first-position databank
- `*!*` matches all series of all frequencies in the first-position databank
- `%*` matches all scalars in the first-position databank
- `#*` matches all collections in the first-position databank
- `**` matches all variables in the first-position databank

The last two-star wildcard is special, and can be understood as `**` = `*!*` + `%*` + `#*`. If you need to perform the same search in a particular databank, or in all databanks, just add bankname and colon:

- `*:*` matches all series of the current frequency in all databanks
- `*:*!*` matches all series of all frequencies in all databanks
- `*:%*` matches all scalars in all databanks
- `*:#*` matches all collections in all databanks
- `*:**` matches all variables in all databanks

You may of course indicate a particular databank, for instance `b2:*`. To match everything, use the following three-star wildcard:

- `***` matches all variables in all databanks, that is, 'everything'.

Array-series

You can match sub-series inside an array-series like this:

```
reset;
x = series(1);
x[ia] = 1; x[ib] = 2; x[ja] = 3; x[jb] = 4;
index x[i*]; //x[ia], x[ib]
y = series(2);
y[i, a] = 1; y[i, b] = 2; y[j, a] = 3; y[j, b] = 4;
index y[i, *]; //y[i, a], y[i, b]
index y[**]; //y[i, a], y[i, b], y[j, a], y[j, b]
index y[*, b] to #m; //y[i, b], y[j, b]
prt #m; //list: ('y[i, b]', 'y[j, b]')
prt {#m}; //y[i, b], y[j, b]
```

If you want to use sub-series wildcards when for instance printing or in general expressions, you cannot use the 'naked' form of the wildcards (otherwise the meaning of `*` would become ambiguous). Instead, elements containing wildcards must be enclosed in single quotes (`'`), for instance:

```
//continued...
prt y['*', b]; //y[i, b], y[j, b]
```

Note

If you use variable names without wildcards or ranges, a kind of existence check is performed. The variable name will be kept in the resulting list only if the variable exists.

You may also try `series ?;`.

See also the wildcards page regarding wildcards, syntax, etc.

Related statements

LIST, COUNT

4.3.44 INI

The INI statement runs any gekko.ini file, if this file is present in the program folder (where gekko.exe is located) and/or working folder.

See concrete examples of INI files in the RESTART help file.

Syntax

ini ;

Note

The RESTART statement is in reality a RESET statement followed by an INI statement. If no gekko.ini files are present, RESTART and RESET are equivalent.

Note: With `option interface remote = yes;`, Gekko may be remote-controlled from a special remote.gcm program file in the working folder (cf. OPTION).

You maybe put a gekko.ini next to gekko.exe, so that every time Gekko is started up, you can be sure that this gekko.ini is run. This can be practical for very general settings, like MODE, initial time period, file folders etc. Alternatively, you can put a gekko.ini in your working folder, so that this gekko.ini will be run when Gekko starts up in that folder. You may put a gekko.ini in both locations, in which case the gekko.ini next to gekko.exe will be run first.

Settings etc. in a gekko.ini file can with advantage be put in the Global databank, for instance `global:%start_period = 1980;`. Variables in the Global databank survive READ, CLEAR, etc., and do not 'pollute' the first-position databank if this is later on written to file.

Related statements

RESTART, RESET

4.3.45 INTERPOLATE

INTERPOLATE transforms (disaggregates) a lower-frequency timeseries to a higher-frequency timeseries, for instance converting annual data to quarterly data, optionally using a higher-frequency indicator series (use COLLAPSE to perform the inverse transformation). The statement ignores the global time period, and a local time period cannot be set.

Instead of the INTERPOLATE statement, you may alternatively use the similar `interpolate()` function, for instance `x!a.interpolate();` (see under functions).

Syntax

interpolate < print > highfreq = lowfreq indicator method indicatormethod;

highfreq	Higher frequency timeseries. Frequency can be indicated with suffix `!a`, `!q`, `!m` or `!w`. Banknames may be used. Lists of names can be used, like for instance `{#m}`.
lowfreq	Lower-frequencey timeseries. Frequency can be indicated with suffix `!a`, `!q` or `!m`. Banknames may be used. Lists of names can be used, like for instance `{#m}`
indicator	(Optional). A high-frequency indicator series to aid the construction of the resulting timeseries, by providing the seasonal patterns. When using an indicator, choose between the `olsette`, `cholette` or `denton` methods.
method	(Optional). Choose between: • `total`: The sum of the high-freq values is equal to the low-freq value. Example: an annual value of 100 will be split into four quarterly values of 25 each. • `avg`: The average of the high-freq values is equal to the low-freq value. Example: an annual value of 100 will be split into four quarterly values of 100 each. • (`repeat`: Obsolete: use `avg`.) • (`prorate`: Obsolete: use `total`.) Note: if no method is indicated, default is `avg`. You can alter the default with `option interpolate method = ... ;` (cf. OPTION).
indicatormethod	(Optional). When using an indicator, choose between these metods, all of the "Denton family":

- `olsette`: Unless you have very few observations in the low-frequency series, this is the recommended method to use with an indicator series. The method is more foolproof than both the `cholette` and (especially) the `denton` methods. With `olsette`, the collapsed high-frequency indicator series is first collapsed into a low-frequency indicator, which is OLS-fitted on the low-frequency series (+ a trend and a constant term), following which an adjusted low-frequency indicator series is constructed. The adjusted indicator series is then used instead of the "real" indicator series, using the Cholette method. The Olsette method is invariant regarding the relative and absolute levels of the high-frequency indicator versus the low-frequency series. The method may fail with an OLS error if for instance the OLS algorithm has too few degrees of freedom (too few low-frequency observations), and also -- but much more rare -- if the indicator series collapses exactly into a constant series (in that case, OLS also fails). Use `option interpolate olsette trend = no;` to switch off the trend in the OLS estimation.
- `cholette`: The Cholette method is essentially an "improved Denton", fixing the Denton problem of spurious movements in the beginning of the resulting series, if the levels of the low-freq and indicator series do not match. The method tries to minimize the absolute changes in differences between the indicator series and the resulting series, while preserving total- or avg-aggregation. The Cholette method is invariant regarding absolute levels of the high-frequency indicator versus the low-frequency series (but it is not invariant regarding the relative levels of the high-frequency indicator versus the low-frequency series).
- `denton`: This is the original Denton method, cf. this paper (reproduces the delta(x-z) column on page 101). The Denton method is sensitive regarding the relative and absolute levels of the high-frequency indicator versus the low-frequency series. So in general, use `olsette` or `cholette` rather than `denton`, because with Denton, you risk spurious movements at the start of the resulting series. Denton should only be used if Olsette/Cholette uses too many computer ressources, which would only happen with extremely long timeseries.

When using an indicator, you must at the same time choose both a *method* and an *indicatormethod*, for instance `interpolate x!q = y!a indicator=z!q avg cholette;`.

Note that during calculations with indicators, no matter TIME settings or local `<...>` period, the method will always use the longest possible period where there is data for both the low-frequency and high-frequency input series. In the `interpolate()` function variant, TIME or `<...>` period only affect which periods are updated *after* the calculation is performed.)

print	(Optional). When using an indicator together with the `olsette` method, the OLS regression is printed on the screen for the user to inspect. With many INTERPOLATE statements, this option will slow down Gekko considerably due to the printing.

- If a variable without databank indication is not found in the first-position databank, Gekko will look for it in other open databanks if databank search is active (cf. MODE).
- Looping: with a list like for instance `#m = x1, x2;`, you may use `interpolate {#m}!q = {#m}!a;` to interpolate `x1!a` into `x1!q`, and `x2!a` into `x2!q`.

Examples

Use this to convert frequency:

```
reset; time 2018 2020;
x!a = 2, 3, 4;
interpolate x!q = x!a;
prt <n> x!a, x!q;
```

Since the method is `avg` as default, this will create the quarterly timeseries `x!q` where each quarterly observation in `x!q` is the same as the corresponding annual observation in `x!a`. Alternatively, you use `total` option (`interpolate x!q = x!a total;`). Now, the quarters will sum up to `x!a` instead of just being repeated.

Note that when interpolating from `a`, `q` or `m` frequency into `w` (weeks), the weeks at the start/end will often become missing values, when the weeks do not fit exactly inside the years, quarters or months. Consider this example:

```
reset; option freq m; time 2017m1 2017m2;
x!m = 1;
interpolate x!w = x!m total;
option freq w; time 2016w52 2017w9;
prt<n> x!w;
prt sumt(<2017w1 2017w8>, x!w);   //1.8963

//                      x!w
// 2016w52                M //6 missing days
// 2017w1          0.2258
// 2017w2          0.2258
// 2017w3          0.2258
// 2017w4          0.2258
// 2017w5          0.2431
// 2017w6          0.2500
// 2017w7          0.2500
// 2017w8          0.2500
// 2017w9                M  //5 missing days
```

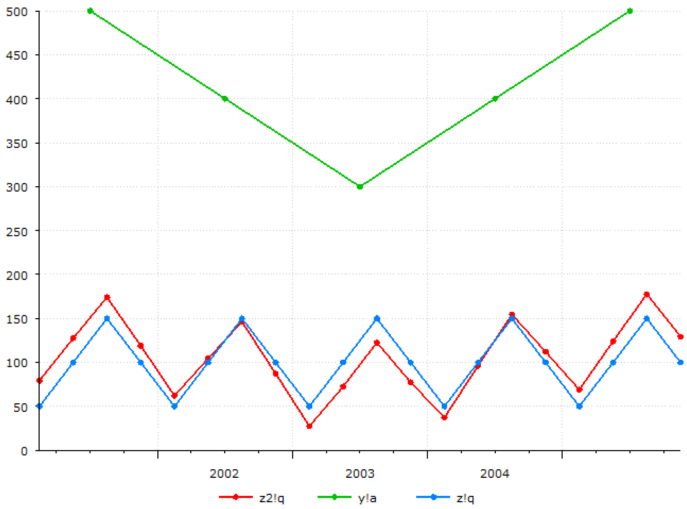
The data for the months `2017m1` and `2017m2` are interpolated into the weeks `2016w52`-`2017w9`. Week #52 of 2016 actually contains January 1, 2017 (a Sunday), but since the six previous days of that week are missing (December 26 to December 31, 2016), the interpolation sets 2016w52 to missing values. Similarly, week #9 of 2017 contains the days February 27 and February 28, but the five subsequent days are missing. The sum is not exactly 2, because one day of January and two days of February are "spilled" (1.8963 = 2 – 1/31 – 2/28). (Weeks are defined and numbered following the ISO 8601 standard, where days around New Year may belong to week 52, week 53 or week 1. Active 'workdays' will be implemented, omitting for instance weekends and holidays. For now, see the getSpecialDay() function).

Olsette/Cholette

The following is an example of using the Cholette method. (Note: changing the method to `olsette` here will result in an error, because the collapsed indicator is a constant annual series with value 400, which makes the OLS regression fail. Changing the method to `denton` gives almost the same result, because the series are well aligned level-wise).

```
//Cf. example Denton: Adjustment of Monthly or Quarterly Series to
Annual Totals: An Approach Based on Quadratic Minimization
(https://www.oecd.org/sdd/21779760.pdf). The results below differ a
tiny bit from
the values in the article, because the Cholette method is used
here. To reproduce exactly, use
"interpolate z2!q = y!a indicator=z!q total denton;"

reset;
time 2001 2005;
y!a = 500,                400,                300,
              400,                500;
z!q = 50, 100, 150, 100,  50, 100, 150, 100,  50, 100, 150, 100,
 50, 100, 150, 100,  50, 100, 150, 100;
interpolate z2!q = y!a indicator=z!q total cholette; //Or: z2!q =
interpolate(y!a, z!q, 'total-cholette');
p <n> z2!q.collapse(), y!a, z!q.collapse(); //Shows that z2!q sums
up to y!a, but z!q does not.
prt <n> z2!q; //z!q.collapse() should be comparable to y!a levels!

 --> result:
               z2!q
 2001
 q1          79.2980
 q2         127.5788
 q3         174.1404
 q4         118.9828

 2002
 q1          62.1060
 q2         104.5129
 q3         146.2034
 q4          87.1777
```

```
2003
q1          27.4355
q2          72.5645
q3         122.5645
q4          77.4355

2004
q1          37.1777
q2          96.2034
q3         154.5129
q4         112.1060

2005
q1          68.9828
q2         124.1404
q3         177.5788
q4         129.2980
```



The above can be exactly reproduced with the `tempdisagg` package in R:

```
library(tempdisagg)
d.q <- ts(rep(c(50, 100, 150, 100), 5), frequency = 4)
d.a <- ts(c(500, 400, 300, 400, 500))
a1 <- predict(td(d.a ~ 0 + d.q, method = "denton-cholette",
criterion = "additive", h = 1))
print(a1)

--> result:

       Qtr1       Qtr2       Qtr3       Qtr4
1  79.29799  127.57880  174.14040  118.98281
2  62.10602  104.51289  146.20344   87.17765
3  27.43553   72.56447  122.56447   77.43553
4  37.17765   96.20344  154.51289  112.10602
5  68.98281  124.14040  177.57880  129.29799
```

Note

If a frequency indicator is omitted, Gekko will use the current frequency.

Interpolate with `total` option and collapse with `total` option are in a sense mirror images, and for example for an annual timeseries `x!a`, `x!a.interpolate('total')` will disaggregate into quarters, and `x!a.interpolate('total').collapse('total')` will aggregate the quarters back to the original annual series (this identity does not work if you are interpolating to and collapsing from weekly data).

The `olsette` method is named both after OLS and Jes Asger Olsen.

Related options

OPTION interpolate method = repeat; [total|avg]
OPTION interpolate olsette trend = yes; [yes|no]

Related statements

COLLAPSE, SMOOTH

4.3.46 ITERSHOW

The statement ITERSHOW show details of previous Gauss-Seidel iterations (cf. SIM).

Syntax

```
itershow < period > variables;
```

period	(Optional). Local period, for instance `2010 2020`, `2010q1 2020q4` or `%per1 %per2+1`.
variables	Variable(s) or a list like `{#m}`.

Example

Use this syntax to show iterations for the variable `gdp` for the year 2010:

```
itershow <2010 2010> gdp;
```

Details

The statement produces output containing (for each iteration) the value of the endogenous variable before and after simulating the Gauss-Seidel loop, and differences, different criteria etc.

The output shows the iteration number, values before and after the iteration. The next column "Hist. var" is the historical variance/variability in the data, obtained by means of looking at lagged historical values of the endogenous variables.

The next column is the difference between the values before and after the iteration, and "Relative1" is this difference divided by the historical variance. This is the criterion Gekko uses for relative convergence per default.

The last column is the difference divided by the value before the iteration. "Relative2" is the criterion used in the software package PCIM. So "Relative1" corresponds to setting "OPTION solve gauss conv = conv1" (default), and "Relative2" corresponds to setting "OPTION solve gauss conv = conv2".

It is often the case that "Relative1" is larger than "Relative2", so using this criterion is stricter and would demand more iterations given the same relative criterion (0.0001 for instance). Sometimes the inverse is true, however, especially when the "Before" value is close to zero, whereas the historical variability is large. In that case, the PCIM-like criterion ("Relative2") would be stricter. But often this will just postpone the solution, in case the variable just happens to be have a solution close to zero in that particular year (but without being close to zero in general). Examples of this could be balances and flows, for instance the balance of payments, net investments, revaluation ("omvurderinger") etc. Close-to-zero solutions for such variables are different in kind from variables with levels generally close to zero (in ADAM, for example interest rates). Looking at historical variability as done in Gekko is actually a means to try distinguishing such classes of variables from each other regarding convergence. If, for instance, the balance of payments can change by an amount of around 20000 (million DKK) from year to year, and the true solution in a particular year just happens to be 1 million DKK, we are not interested in obtaining an extreme precision (i.e., many digits after the decimal point) regarding that particular value. If the variable normally can change by an amount of 20000 from year to year, a solution of 2 is nothing to worry about, even if the true solution is 1. Whereas if the true solution regarding an interest rate is 0.04, an alternative solution of 0.08 is worrying.

Note

In order for this statement to work, `option solve gauss dump` must first be set to `yes`, and a simulation performed. Beware that setting this option consumes a lot of RAM when simulating, and also slows simulations down. In case of a RAM error, try to limit narrow the time period.

Related statements

SIM

4.3.47 LIBRARY

The LIBRARY statement can be used to load and handle libraries of user-defined functions, procedures or external files. A Gekko library is basically a .zip-file containing user-defined functions/procedures inside Gekko program files (.gcm), and data files inside the `\data` subfolder of the zip file (data files can be for instance table or plot schemas, list files, databanks, etc.).

> **Don't be scared**
> Gekko libraries are designed to be as easy as possible to develop and use. If the user knows how to zip one or more program .gcm files into a zip file, that will be enough to create a library. There is no compiling, packaging, registering, uploading, installation, etc. Just zip the file(s), and the library is ready to use and share, and because of caching and so-called lazy loading, a LIBRARY statement normally executes very quickly, even if the library is large.

Libraries are organized in a hierarchy, similar to how the databanks hierarchy works. There is a special kind of library always available: the so-called `Local` library (empty when Gekko starts up). The `Local` library stores all "normal" functions/procedures, defined in a normal .gcm program file (or the statement prompt). With the LIBRARY statement, additional libraries can be loaded from library files, and when a function `f()` or a procedure `f` is called, Gekko searches for it in the hierarchical list of loaded libraries. If you need to be sure that a function/procedure is taken from a particular library, you may use colon (`:`) to indicate the library, for instance `lib1:f(x)`. The `Local` library is always searched first in the library hierarchy.

Inside the library .zip file, you may organize your .gcm files (containing functions/procedures) as you please, but note that libraries operate with the special subfolders `\data` and `\meta`. Gekko will not look for .gcm files with functions/procedures inside these folders, which are used for possible data files and/or meta information (descriptions etc.). If you put for instance a file `data.csv` in the subfolder `\data` of the library `lib1.zip`, you may subsequently use `read <csv> lib1:data.csv` (or just `read <csv> data.csv;`), and Gekko will find the file, provided that the library is loaded (such library files are read-only). Note that when referring to a library with colon (`:`), backslashes should never be used.

Syntax

```
library lib1, lib2, ... ;              //Load these files/libraries
library lib1 as l1, lib2 as l2, ... ; //Alias names
library <close> lib1, lib2, ... ;      //Close libraries (not possible
for the Local library).
library <close> *;                     //Close all libraries (except
Local library).
library <clear> local;                 //Clear the Local library.
library ?;                             //Show info about loaded
libraries.
```

To refer to a function/procedure from a particular library, use colon (`:`). For instance `lib1:f()` or `lib1:f`. When loading a library with LIBRARY, if no file path is provided, Gekko will look for library files in the working folder. If not found there, it will additionally look in the folders designated with `option folder command = ...`, `option folder command1 = ...`, and `option folder command2 = ...`.. This is where Gekko also looks for normal `.gcm` files, cf. RUN.

Examples

As noted above, Gekko libraries are simply a collection of .gcm files (and possibly other files) stored inside a .zip file. You may use any folder structure inside the .zip file: this structure has no significance (apart from the folder names `\data` and `\meta` being specially treated). Try creating the following two .gcm files (you may use EDIT to do that, or use some other text editor). After the files are created, pack them into individual zip files (you may alternatively download the two .zip files here and here).

lib1.zip, containing lib1.gcm

```
//lib1.gcm
function string f1();
  return 'lib1';
end;
```

lib2.zip, containing lib2.gcm

```
//lib2.gcm
function string f1();
  return 'lib2';
end;

function string f2();
  return f1();
end;
```

The two zip-files, `lib1.zip` and `lib2.zip`, should be located in the working folder. Now, we can try out the libraries:

```
reset;
library lib1;             //loads lib1.zip
prt f1();                 //prints 'lib1'
library <close> lib1;     //closes the library
prt f1();                 //error
library lib1;             //(re)load it
library lib2;             //load lib2.zip, too. Library list is now
Local, lib1, lib2
library ?;                //Info on libraries and their contents.
prt f1();                 //prints 'lib1' because lib1 is before
lib2.
```

```
prt lib2:f1();             //prints 'lib2' because lib2 is denoted
with colon.
prt f2();                  //prints 'lib2' because own library f1()
version has priority
function string f1(); return 'Local'; end;
prt f1();                  //prints 'Local', because local functions
have priority
prt f2();                  //still prints 'lib2' because own library
f1() version has priority
```

When both `lib1.zip` and `lib2.zip` are loaded, `lib1` is before `lib2` in the hierarchical list of loaded libraries. When `f1()` is called, Gekko first looks in `Local` (empty), and then in `lib1`, where it is found. Therefore, `prt f1();` prints `'lib1'`, because it is the version of `f1()` in `lib1` that gets called. When instead `lib2:f1()` is used to denote the library explicitly, Gekko locates `f1()` in `lib2` instead. In the last line above, `f2()` is called. This function only exists in `lib2`, and the function basically just calls `f1()`. When `f1()` is called from within `f2()`, Gekko first looks for `f1()` in the same library as the calling function resides in (namely `lib2`). Therefore, `prt f2();` prints `'lib2'`. As shown with the `function` statement above, if you define `f1()` locally, without putting it inside a library zip file, it will take priority over other versions of `f1()`.

The `library ?;` statement will print the following on screen:

```
---------------------------------------------------------------
Libraries:
---------------------------------------------------------------
'Local' with 0 functions and 0 procedures
'lib1' with 1 function and 0 procedures (more)
'lib2' with 2 functions and 0 procedures (more)
---------------------------------------------------------------
```

If you for instance click *more* regarding `lib2`, its functions are shown:

```
---------------------------------------------------------------
Library 'lib2':
---------------------------------------------------------------
2 functions: f1(), f2()
---------------------------------------------------------------
```

If you click a function, its contents (Gekko code) is shown in the output tab.

External files inside libraries

As mentioned above, you may also put other kinds of data files inside a library. Such external files must be placed inside a `\data` subfolder inside the .zip file. Such files will be read-only and can be for instance table or plot schemas (.gtb/.gpt), Gekko databanks, data files, etc. In general, when encountering a filename without a path, Gekko will first look for the file in the working folder, and next in folders stated with `option folder ... ;`. If not present there, and if libraries are loaded, Gekko will also look for the file inside the `\data` subfolder of the libraries.

For instance, consider the plot schema file (setting the plot style to show histograms):

pretty.gpt

```
<gekkoplot>
  <type>boxes</type>
</gekkoplot>
```

Try storing this as a file `pretty.gpt`, and put it inside a `\data` subfolder of a zip-file called `lib3.zip` (or download the zip file from here). Next, place the zip file in your working folder, and try the following:

```
reset;
time 2021 2025;
x1 = 2, 4, 3, 5, 4;
x2 = 3, 6, 5, 7, 8;
plot x1, x2;                     //normal lines
library lib3;                    //loads lib3.zip
plot <using=pretty.gpt> x1, x2;  //boxes/histograms
option plot using = pretty.gpt;  //set global plot schema
plot x1, x2;                     //boxes/histograms
```

If you want to be completely sure that a file originates from a particular library, you may prefix the library name like `lib3:pretty.gpt`. Inside the `\data` folder of `lib3.zip`, you may place `pretty.gpt` as you please. Examples of file types that can be stored in the `\data` subfolder: databanks, data files, listfiles, model files, program files, R/Python files, menu browser files, table files, plot files, download definitions (.json), etc.

Difference between the Local library and loaded libraries

As mentioned above, external libraries are loaded in a lazy fashion, that is, the functions/procedures inside the library are not actually parsed and compiled, until they are called. Therefore, syntax errors in a function or procedure definition do not show up immediately, in contrast to functions/procedures defined in a normal .gcm file. Normal functions/procedures end up in the Local library, which is always first in the hierarchical list of loaded libraries. The Local library cannot be closed, but it can be cleared with `library <clear> local;`. The Local library never contains external files, only functions and procedures.

Strategy when developing a library

If you are developing a library, you may proceed following these steps:

1. Define the function/procedure in a normal .gcm file (for instance called `lib.gcm`).

2. Run the .gcm file with `run lib.gcm;`, and fix any syntax errors. (If incomprehensible errors are encountered, you may try to out-comment the FUNCTION statement and corresponding END statement for better error messages).
3. When `lib.gcm` has no more syntax errors, try calling and testing the function/procedure from the statement prompt. This will reveal possible run-time errors.
4. If this works as expected, you may put the function/procedure into a library zip file: zip `lib.gcm` into the zip file `lib.zip`. Try issuing the statement `library lib;`, and try subsequently to call the function/library again.
5. Later on, you may need to change the function/procedure inside `lib.zip`. Debugging from within a library zip file may be a bit confusing, because syntax errors do not show up at once, but first when the function/procedure is called. But you may always unpack the .gcm file inside the .zip file and try it out directly with `run lib.gcm;`, essentially returning to step (2) above. When the file is error-free and tested, re-zip it back into `lib.zip`.

Private functions/procedures/files

If the library `lib1` contains a function `f()`, and this function calls a function `g()`, Gekko will first look for `g()` in `lib1` (similarly regarding procedures and external files). In this sense, a library has a concept of "locality".

If you need further encapsulation to guard against naming ambiguities, there are the following possibilities:

- Private names: After loading a library, per default all of its functions, procedures and external files will be available from outside of the library. If you do not want some helper functions/procedures/files to be accessible (exposed) for use from outside of the library itself, you may prefix the name of the function/procedure/file with `__` (double underscore) to ensure privateness.
- 'This' keyword: Inside a library function/procedure, you may use `this:` prefix on function/procedure/file names to refer to the library itself (guarding against bugs if the function/procedure/file does not exist in the library, but accidentally exists somewhere else).

How does this compare to R/Python packages?

A Gekko library is similar to a R or Python *package*. In R, packages are typically first *installed* with syntax like `install.packages("lib1")`, and in Python with syntax like `pip install "lib1"`. After installation, in R the package is *loaded* with `library(lib1)`, and in Python with `import lib1`. Gekko skips the installation part and only performs loading, so in Gekko, the statement `library lib1;` is comparable to `library(lib1)` in R or `import lib1` in Python. Gekko libraries are intentionally simpler to create than their package counterparts in R and Python. Anyone capable of putting a function or procedure into a .gcm file and zip this file should be able to create a Gekko library!

Notes and details

For user-defined functions and procedures, beware that it is not advised to use the same name as an in-built Gekko function or statement name (in that case, the in-built version always takes precedence, unless an explicit library name is designated with colon `:`). You may encounter the problem that an new Gekko version uses the same function or statement name as your user-defined function/procedure. In that case, it is advised to rename your user-defined function/procedure (Gekko will issue warnings in such cases).

The names of the .gcm files outside of the `\data` subfolder of the zip file have no significance. Some people prefer to store the function `f()` inside a file called `f.gcm`, whereas others store all functions/procedures inside the same large .gcm file. The .gcm files outside of the `\data` subfolder of a zip file may only contain functions or procedures, nothing else (and nested functions/procedures are not legal). When referring to functions/procedures/files inside a library file, the internal folder structure of the zip file has no significance (other than external files reside in the `\data` subfolder, and functions/procedures do not). You may define different versions of the same function or procedures, provided that they have different numbers of arguments (so-called overloads).

Libraries are loaded in a lazy fashion. So when referencing a library via a statement like `library lib1;`, the functions/procedures inside `lib1.zip` are not actually parsed and compiled, until a particular function/procedure from `lib1.zip` is called. In order for LIBRARY to read a library .zip file as fast as possible, the statement uses text heuristics to locate the functions/procedures inside the .gcm files of the .zip file. The heuristics are mostly a question of locating the FUNCTION or PROCEDURE statements and using these to cut the .gcm file into parts corresponding to different functions/procedures. If a library function/procedure fails mysteriously, try to RUN the unzipped .gcm file normally, too see if the library heuristics are to blame.

Library names must be of length > 1, because libraries can also be used to store external files. The length restriction is to avoid the possibility of storing for instance a `pretty.gpt` schema file for plots inside a `c.zip` file, and later refer to the schema file with `c:pretty.gpt`. This would be confusing, since `c:` could be a drive letter. You cannot use the following names as library names: `Local`, `Global`, `this`, `null`, or any name that begins with `Gekko`.

Beware that functions/procedures in a zip library `lib1` have a sense of "locality": if function `f()` from `lib1` calls function `g()`, Gekko will first look in `lib1` for `g()`. Similarly for files: if function `f()` from `lib1` uses a file `data.csv`, Gekko will first look in the `\data` folder of `lib1` for the file. You may use the special keyword `this` as library name to be completely sure that a library function or procedure only may use its own functions/procedures/files. For instance inside the function `f()`, you may call the function `this:g()` or use the file `this:data.csv`. Since per default a library function or procedures first looks in its own library for a function/procedure/file, `this:` is mostly used to guard against the following: consider a library function `f()` that uses another library function `g()` from the same library. This works fine, but then someone deletes `g()` from the library by accident. This would typically result in an error, but what if -- by coincidence -- `g()` is defined in the Local library or in some other library? Using `this:` guards against that possibility.

An even more restrictive option for library functions/procedures is to prefix the names with double underscore (`__`), which makes them private (non-accessible from outside of the library). In that case, `f()` of the library `lib1` could call the private function `__g()` (also from `lib1`), but this function is not callable from outside of `lib1`. Regarding external files in the `\data` subfolder of the library `lib1`, you may follow the same convention and prefix the file names with `__`. (these files will not be accessible from outside of the library `lib1`). Functions/procedures/files with prefix `__` are typically helper functions/files that the user does not want to expose for general use.

If your library `lib1` uses/depends upon the library `lib2`, it is safest to refer to the components of `lib2` with colon. So a function `f()` from `lib1` may load the library `lib2` (with `library lib2;`, possibly first checking if it is already open with `if(isLibraryLoaded('lib2'))`), and then the function `g()` from `lib2` can be called with `lib2:g()` to make sure that `g()` is really from that library.

Instead of storing Gekko statement lines in an external .gcm file inside the `\data` subfolder of a given library, consider storing the same lines as a procedure in the non-`\data` part of the library.

Note that Gekko supports "zip paths", so that in principle you may "manually" refer directly to a file inside a zip file. So using a file name like `lib1.zip\data\pretty.gpt` is possible and will work without loading `lib1.zip` as a library (using a zip path, Gekko will automatically extract `pretty.gpt` from inside of `lib1.zip`). Still, loading the .zip file as a library in such cases is typically more convenient.

Related options

```
OPTION folder command = [empty];
OPTION folder command1 = [empty];
OPTION folder command2 = [empty];
OPTION folder working = [empty];
```

Related statements

FUNCTION, PROCEDURE, RUN

4.3.48 LIST

A Gekko list contains sequentially ordered variables (elements) of any type. List names always start with the symbol `#`, like the other collection types map and matrix. You may refer to list elements by means of indexes, for instance `#m[1]` for the first element of `#m`. Listfiles can be used, using for instance `#(listfile m)` instead of `#m`, where the list elements are stored in the external file `m.lst` instead of in a databank.

Guide: lists
For an easier introductory guide on Gekko lists, see this page.

Upgrade note
Gekko 2.x uses syntax like `list m = a, #m1, d, #m2;`, which in Gekko 3.x would be `#m = a, {#m1}, d, {#m2};` (the string `'a'` plus the strings inside `#m1`, etc.).

Scrape data from xls(x) spreadsheets
See SHEET<import> to see how to load a spreadsheet as a nested list of "cells" and scrape data from these with loops.

A general list is defined with parentheses and commas, for instance `#m = ('a', 120, 2020q3);`. In that case, the list contains a string, a value, and a date. In the special case where all the list elements are simple alphanumeric words (including '_') without special characters, you may use for instance `#m = a, a38, 7z, 5, 007, 2001q1;` instead of the more cumbersome `#m = ('a', 'a38', '7z', '5', '007', '2001q1');`. The 'naked' version without parentheses and quotes is practical for lists of names etc. If all the list elements are values, like `#m = 1, 2, 3;`, a list of values is produced instead of a list of strings (among other things, this is practical regarding the series statement, for instance `y = 1, 2, 3;`). A naked list either returns a list of strings or a list of values, cf. the page on naked lists.

You may may use the operators `+=` and `-=` to add or subtract elements from a list (this works for naked lists, too: `#m = a, b; #m += c, d; #m -= c, d;`). If you need to use a naked list with one element, use a trailing comma, for instance `#m = a,;`. This way, you can easily add or remove a single element like this: `#m = a, b; #m += c,; #m -= c,;`. Single-item lists (so-called singletons) can alternatively be stated as `#m = list('a')` or `#m = ('a',)`. Using `#m = a` or `#m = ('a')` will fail with an error, since Gekko understands this as setting a list equal to a series or a string. An empty list can be created with `#m = list()`.

The first element of a list #m is `#m[1]`, and the last element is `#m[#m.length()]`. Slices/sublists can be cut out by means of ranges, for instance `#m[2..5]`. Non-naked lists may contain any other variable types, including lists, so `#m = (1, (2, 3));` is a nested/reursive list, where `#m[2][1]` would refer to 2 (because `#m[2]` refers to the sublist `(2, 3)`).

Syntax

```
#name = ( expr1 REP n1 , expr2 REP n2, ... ); //REP repeats the item
#name = name, name, ... ;                    //short form for 2 or more
string variables
#(listfile m1) = #(listfile m2);             //use of listfiles m1.lst
and m2.lst (for lists of scalars, or nested lists of scalars)
#name = list(...);                           //useful for singleton
lists or empty lists like #m = list('a') or #m = list().
list #name = ...;                            //list keyword may be
added, but is typically not necessary.
list ?;                                      //show/count lists in open
databanks
```

It is no longer legal to use for instance `list m = ... ;`, omitting the `#` on the left-hand side.

Referring:

```
{#m}                //use braces {...} to use strings as variable names.
```

You may pick out individual items from a list with the `[]`-brackets. For instance:

- `#x[%i]` = element number `%i` (a scalar)
- `#x[%i1..%i2]` = elements `%i1` to `%i2` (inclusive), returns a list (slice)
- `#x[%i1..%i2, %j1..%j2]` = matrix-like selection of nested list
- `#x[%i1..%i2][%j1..%j2]` = beware that this is different from above (nested indexing).
- `#x['fx*']` = returns a list of those string elements that start with `'fx'`
- `#x['f?a']` = strings that match the pattern `'f?a'`
- `#x['pxa'..'pxqz']` = strings in the range `'pxa'` to `'pxqz'` (both inclusive)
- `#x['a']` = check if the string `'a'` is an element in the non-nested list `#x`.
- `#x['a', 'b']` = check if the list `('a', 'b')` is a sublist in the nested list `#x`. ALso works for more dimensions.

Use `#x.length()` or `length(#x)` to get the number of elements in `#x`, `#x[0]` cannot be used for this anymore. In IF statements and $-conditions, you can use the syntax `if(#x['a'] == 1)` or `if('a' in #x)` to condition on `#x` containing some particular element.

Operators:

- `#y = #x1 + #x2;`. Same as `#y = #x1.extend(#x2);`. Adds the elements of `#x2` to `#x1`, use of `+=` is possible. Note that `#x1 + %s` is not legal (where for instance `#x1` is a list of strings and `%s` is a string): to append to each list item, use `#x1.suffix(%s)` instead.
- `#y = #x1 - #x2;`. Same as `#y = #x1.except(#x2);`. Subtracts the elements of `#x2` from `#x1`. Use of `-=` is possible.

- `#y = #x1 || #x2;`. Same as `#y = #x1.union(#x2);`. Union of two lists, dublets removed.
- `#y = #x1 && #x2;`. Same as `#y = #x1.intersect(#x2);`. Intersection of two lists.

In order to generate sequences you can use the function `seq()`, for instance `#a = seq(1, 100);` produces a list of values `1`, `2`, ... , `100`, and that `#a = seq(1, 100).strings();` produces a corresponding list of strings `'1'`, `'2'`, ... `'100'`.

The LIST keyword is optional and can typically be omitted. Use `rep` to repeat items. For the last item, when defining or calculating timeseries, you may use `rep *`. You may use a list definition with parentheses (strict version) everywhere a variable or expression is expected.

There are quite a lot of functions to deal with string lists, for instance sort() to sort elements, unique() to remove dublets, prefix()/suffix() to add prefixes/suffixes to elements, etc. You can also use the union(), except(), and intersect() functions for lists of strings, or equivalently via operators `||`, `-`, and `&&`. Please see the examples below, and the functions section. Regarding string lists of variable names, there are a lot of functions to handle banks, frequencies, indexes, etc., cf. the functions section, under 'Bank/name/frequency/index manipulations'.

A listname always starts with the symbol '#', like the other collection types map and matrix. If you prefer to refer to a list item by name instead of numbers/indexes, see the MAP collection. In that case, you could use for instance `#m['nairu']` or `#m.nairu` rather than for instance `#m[1]` to refer to a particular named object in the collection (this makes the programs easier to read).

String lists with variable names, optionally including banks, frequencies, indexes, etc. are often used, and to refer to the variables themselves, {}-curlies must be used. For instance: `#m = ('x', 'b2:y!m[a, u]', '%z');` contains the strings corresponding to `x` (series x without bank indication, and with default frequency), monthly array series `y` from bank `b2` with indexes `[a, u]`, and finally a scalar `%z`. If you want to refer to the variables corresponding to these strings, use {}-curlies, for instance `PRINT {#m};`. This is the same as `PRINT x, b2:y!m[a, u], %z;`.

Lists of values are often used, for instance to define series. A list of values could be `#m = 1, 2, 3;`, omitting the parentheses. If you need to use the sequential data in linear algebra, instead of `#m = 1, 2, 3;` or `#m = (1, 2, 3);` you may use a row or column vector instead, for instance `#m = [1, 2, 3];` or `#m = [1; 2; 3];`. The latter column vector is a bit more similar to the list in the sense that `#m[1]`, `#m[2]` is allowed regarding column vectors, where row vectors would have to use `#m[1, 1]`, `#m[1, 2]`, etc. If a list of values is given as mathematical expressions, you must use enclosing parentheses, for instance `#m = (1/7, 2 + %v, 3*%v);`.

All Gekko functions implement so-called UFCS so that a function like for instance `extend(#x, #y)` can generally be written as `#x.extend(#y)`. This makes chaining of such function calls more readable, for instance `#m = #m1.extend(#m2).except(#m3)`.

List functions:

Note that some of the functions assume that the lists are lists of strings. This will be fixed regarding values and dates.

Function name	*Description*	*Examples*
[x]-index	Index: picks out a single element. In contrast to R, this does not return a 1-element list containing the variable. If you need that, use for instance #m[3..3]. Returns: var	`#m[3];   //the third element`
[x1..x2]-index	Index: picks out a range of elements. You may omit x1 or x2. Returns: list	`#m[3..5];  //the third to fifth elements`

[x1, x2]-index	For a nested list of lists, `#m[3, 5]` will return the same element as `#m[3][5]`, so this is just convenience to make a nested list accessible like a matrix. See more here.
	Returns: variable
	
	[New in 3.0.6].
		`#m = ((1, 2), (3, 4));`
		`prt #m[2, 1], #m[2][1];  //same`

[x1..y1, x2..y2]-index
[x1..y1, x2]-index
[x1, x2..y2]-index
	For a nested list of lists, `#m[2..3, 2..4]` will select the given 'rows" and "columns", corresponding to selecting a submatrix from a matrix. Beware that in general, `#m[2..3, 2..4]` is completely different from `#m[2..3][2..4]`. See more here.
	Returns: list
	[New in 3.0.6].
		`//  1  2  3`
		`//  4  5  6`
		`//  7  8  9`
		`// 10 11 12`
		`#m = ((1, 2, 3), (4, 5, 6), (7, 8, 9), (10, 11, 12));`
		`prt #m[2, 2..3];`
		`prt #m[2][2..3]; //same as above`
		`prt #m[2..4, 2]; //matrix-like selection`
		`prt #m[2..4][2]; //different from above!`
		`prt #m[2..4, 2..3]; //matrix-like selection`
		`prt #m[2..4]`

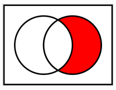
		[2..3]; //different from above!

append(x1, x2)
append(x1, i, x2)
	Adds variable x2 as it is at the end of list x1. Note that if x2 is a list of for instance 3 items, only 1 element is added (the list itself). If you need to add the 3 elements individually, use extend().
	If used with i argument, x2 is inserted at index i, instead of at the end. See also extend().
	To prepend, use append(x1, 1, x2).
	Returns: list
		#y = #x1.append(#x2); //or: append(#x1, #x2)
		#y = #x1.append(2, #x2); //insert at position 2

contains(x1, x2)	Checks if the list of strings x1 contains the string x2. Returns 1 if true, 0 otherwise. You may alternatively use `x2 in x1`, see the last example. See also the count() and index() functions. The comparisons are case-insensitive.
	Returns: val
		%v = #x1.contains(%s);
		if(#x1.contains(%s) == 1); tell 'yes'; end;
		if(%s in #x1); tell 'yes'; end;

count(x1, x2)	Counts the number of times the string x2 is present in the list of strings x1. See also the contains() and index() functions. Note: to obtain the number of elements in a list, use the length() function. The comparisons are case-insensitive. Returns: val	%v = #x1.count(%s); //or: count(#x1, %s)

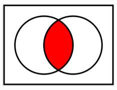
data(x)	Accepts a string of blank-separated values x and turns them into a list of values. This is handy for long sequences of blank-separated numbers, instead of manually setting the commas. Returns: list	`#m = data('1.0  2.0  1.5');`
dates(x)	Tries to convert each element of the list x to a date. Returns: list	`#y = dates(#x);`

except(x1, x2)

	The except() function subtracts x2 from x1. You may alternatively use the operator `-`. Only works for lists of strings. See also intersect() and union().
	Was called difference() in Gekko 2.0. See also extend().
	Returns: list
		`#y = #x1.except(#x2);  //or: except(#x1, #x2)`
		`#y = #x1 - #x2;         //same`
		`#y -= #x1;  //subtract from itself`

extend(x1, x2)
extend(x1, i, x2)
	The arguments x1 and x2 must be lists. The function inserts the elements of list x2 one by one at the end of (or at position i in) the list x1. The resulting list may contain dublets.
	For two lists x1 and x2, you may alternatively use the `+` operator. See also except() and append().
	To pre-extend, use extend(x1, 1, x2).
	Returns: list
		`#y = #x1.extend(#x2);  //or: extend(#x1, #x2)`
		`#y = #x1 + #x2; //same as above`
		`#y = #x1.extend(2, #x2);  //insert at position 2`
		`#y += #x1; //add to itself`

flatten(x)	For at list x, the function returns a flattened version	`#m1 = (1, (2, 3));`
		`#m2 = #m1.flatten(); //or: flatten(#m1).`

	of the list. For instance, the list `(1, (2, 3))` is transformed into a non-recursive list of non-list elements: `(1, 2, 3)`. Returns: list	
index(x1, x2)	Returns the index of the first occurrence of the string x2 in the list of strings x1. Returns 0 if x2 is not found in x1. See also the count() and contains() functions. The comparisons are case-insensitive. Returns: val	`%i = #x1.index(%s);  //or: index(#x1, %s)`

intersect(x1, x2)

	The intersect() function finds the common elements of the two list of strings x1 and x2. The resulting list will not contain dublets. You may alternatively use the operator `&&`. Only works for lists of strings. See also except() and union().
	Returns: list
		`#y = #x1.intersect(#x2);  //or: intersect(#x1, #x2)`
		`#y = #x1 && #x2;`

length(x)	Returns the number of elements in the list x. You may use len() instead of length().
	Returns: val
		`%v = #x.length();  //or: length(#x).`
		`%v = #x.len(); //the same`

list(x1, x2, ...)	Returns a list of the variables x1, x2, etc. The function is handy for lists with only 0 or 1 elements. See examples.
	Returns: list
		`#m = ();      //will fail`
		`#m = list();  //ok: empty list`
		`#m = (1, 2);  //easy`
		`#m = (1);     //will fail`
		`#m = (1,);    //is ok`
		`#m = list(1); //is ok`

lower(x)	Returns string elements in the list as lower-case. Returns: list	`#y = #x1.lower();  //or: lower(#x1)`

pop(x1, i)
pop(x1)
	Removes the element at position i in the list x1. Removes the last element if called with pop(x).
	Returns: list
		`#y = #x1.pop(2);  //or: pop(#x1, 2)`
		`#y = #x1.pop();  //last element`
		`#y = #x1.pop(1);  //first element`

preextend(x1, x2)	Same as extend(x1, 1, x2), putting the elements of x2 in the first position of x1.	`#y = #x1.preextend(#x2);  //insert at position 1`
prefix(x1, x2)	If x1 is a list of strings, each element has the string x2 prefixed (prepended) Returns: list	`#y = #x1.prefix(%s);  //or: prefix(#x1, %s);`
prepend(x1, x2)	Same as append(x1, 1, x2), putting x2 in the first position of x1.	`#y = #x1.prepend(#x2);  //insert at position 1`

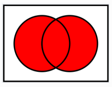
sort(x)
sort(x, 'natural')
	Returns a sorted list of strings, provided that x is a list of strings. Sorting is case-insensitive.
	When the `'natural'` argument is used, strings containing numbers will sort naturally, for instance returning `a8, a9, a10`, instead of `a10, a8, a9`.
	Returns: list
	Natural sort: [New in 3.1.10].
		`#y = #x.sort();  //or: sort(#x)`
		`#y = #x.sort('natural');`

remove(x1, x2)	Removes any string x2 from the list of strings x1. See also the except() function.
	Returns: list
		`#y = #x1.remove(%s);  //or: remove(#x1, %s);`
		`#y = #x1 - %s;        //also legal`

replace(x1, x2, x3)
replaceinside(x1, x2, x3)
replaceinside(x1, x2, x3, max)
	replace(): In the list of strings x1, if this string element is the same as x2, x3 is inserted instead.
	
	replaceinside(): the string element has any occurences of x2 inside the string replaced with x3. The
		`#y = #x1.replace(%x2, %x3); //or: replace(#x1, %x2, %x3)`
		
		`#y = #x1.replaceinside(%x2, %x3); //or: replace(#x1, %x2, %x3, 'inside')`

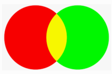
	replacements may be limited via the max argument. Returns: list	
reverse(x)	To be done	
split(x, s)	To be done	
strings(x)	Tries to convert each element of the list x to a string Returns: list	`#y = strings(#x);`
suffix(x1, x2)	If x1 is a list of strings, each element has the string x2 suffixed (appended) Returns: list	`#y = #x1.suffix(%s);  //or: suffix(#x1, %s);`

t(x)	For a nested list of lists, the t() function returns the transpose, similar to transposing a matrix. [New in 3.0.6].
	Returns: list (of lists)
		`#m = ((1, 2), (3, 4));`
		`p #m, t(#m);`

union(x1, x2)

	The union() function finds the union of the two lists. Alternatively use the operator `||`. The resulting list will not introduce dublets, in contrast to the similar `+` operator (simple concatenation). Only works for lists of strings. See also except() and intersect().
	Returns: list
		`#y = #x1.union(#x2);  //or: union(#x1, #x2)`
		`#y = #x1 || #x2;`

unique(x1)	Retains only those elements of list x1 that are unique (list of strings only). Returns: list	`#y = #x1.unique();  //or: unique(#x1)`
upper(x)	Returns string elements in the list as upper-case. Returns: list	`#y = #x1.upper();  //or: upper(#x1)`

vals(x)	Tries to convert each element of the list x to a value Returns: list	`#y = vals(#x);`

venn(x1, x2)	For two lists of strings, this function prints out lists corresponding to a Venn diagram.
	
	That is, intersection and two differences. For instance, `venn(#m1, #m2)` corresponds to printing out the following:
	• `#m1 && #m2`
	• `#m1 - #m2`
	• `#m2 - #m1`
	See also intercept(), except() and union().
	Returns: nothing.
		`#m1 = a, b, c, d, e;`
		`#m2 = c, d, e, f, g;`
		`venn(#m1, #m2);`
		`//Results:`
		`//The following 3 strings are in both lists:`
		`//'c', 'd', 'e'`
		`//The following 2 strings are in list #1, but not in list #2:`
		`//'a', 'b'`
		`//The following 2 strings are in list #2, but not in list #1:`
		`//'f', 'g'`

A string (or list of strings) representing variable names may be manipulated by means of Gekko's inbuilt functions to handle these. Variable names here include bank, frequency, indexes, etc., and examples of such functions could be setBank(), removeBank(), replaceBank(), setFreq(), removeFreq(), setNamePrefix(), etc. There are many more of such functions, see the functions section, under ‘Bank/name/frequency/index manipulations’.

For instance, if you have a list `#m = ('x', 'y');`, you may use `prt {#m};` to print out `x` and `y`, `prt {#m.setBank('b')}`; to print out `b:x` and `b:y`, or `prt {#m.setFreq('q')};` to print out `x!q` and `y!q` (here, `prt b:{#m};` and `prt {#m}!q;` will work, too).

Examples

Define a list:

```
#m = ('a', 'b', 'c');              //list of strings
```

```
#m = a, b, c;                    //naked list syntax, only allowed
for a list of simple names.
#m += d, e;                      //naked list adding two elements.
#m -= d, e;                      //naked list removing them again
#m += d,;                        //naked list adding one element,
note the comma
#m -= d,;                        //naked list removing it again,
note the comma
#i = x, y;
#m = x[a], x[#i];                //naked list allows indexes too
(#i is a list of strings)
#m += #m2;                       //adding another list
#m = 1, 3, 2;                    //list of simple numbers,
parentheses can be omitted
#m = (2020q1, 2020q4, 2021q1);   //list of dates
#m = (a, b, c);                  //NOTE: this is a list of series
objects, not strings!
#m = (('a', 'b'), (1, 3));       //list of lists
#m = seq(1, 3).strings();        //the strings '1', '2' and '3'
```

If instead of for instance `d`, `e`, etc. above you need to use strings like `%s1` or `%s2`, you may use for instance `#m += {%s1}, {%s2};` or `#m += {%s},;`.

As shown, for naked lists you may use indexes to state array-series, for instance `x[#i]` is unfolded into `x[a], x[b], x[c]`, if `#i = a, b, c`. In the last list, `#m[1]` refers to the list `('a', 'b')`, `#m[2]` refers to the list `(1, 3)`, and for instance `#m[1][2]` refers to `'b'`. To print a list, simply use `prt #m;`. If the list is a list of strings, and you want to print the variables corresponding to the strings, use {}-curlies:

```
#m = ('a', 'b', 'c');            //list of strings, or: #m = a, b,
c;
prt #m;                          //print the raw strings
prt {#m};                        //print the variables (series) a,
b, and c. Same as "prt a, b, c;".
```

There are two functions to add items to a list: append() and extend():

```
#m1 = (1, 2, 3);                 //or: #m = 1, 2, 3;
#m2 = (4, 5);
#m = #m1.append(#m2);            //result: (1, 2, 3, (4, 5)),
nested list
#m = #m1.extend(#m2);            //result: (1, 2, 3, 4, 5)
```

Gekko functions implement so-called UFCS, so the function `append(#m1, #m2)` may alternatively be written `#m1.append(#m2)`, moving the first argument of the original function out of the parentheses. This provides a more fluent interface, and enables easy chaining of list functions (see examples below). As it is seen, append() puts the list `#m` at position 4 in `#m1`, whereas extend() unpacks `#m2` and puts the two elements at positions 4 and 5 in `#m1`. Therefore, if you need to add a single value to a list, use append(), but if you need to add the elements of a list to another list, use extend():

```
#m1 = (1, 2, 3);
#m = #m1.append(4);               //result: (1, 2, 3, 4)
#m = #m1.extend(4);               //Fails with an error
```

Instead of `#m1.extend(#m2)`, the `+` operator can be used instead of extend() when dealing with two lists:

```
#m1 = (1, 2, 3);
#m2 = (4, 5);
#m = #m1 + #m2;                   //result: (1, 2, 3, 4, 5), same as
#m1.extend(#m2)
```

It should be noted that `#m1 + %s` does not append `%s` to the list (but will fail with an error). Use `#m1.append(%s)` or `#m1 + list(%s)` instead. In earlier Gekko versions, the expression `#m1 + %s` appended `%s` to each element of `#m1`, but this behavior is deprecated.

Lists in Gekko may contain dublets. Still, set operations are possible with the functions union(), except() and intersect():

```
#m1 = ('a', 'b', 'c');                //or: #m1 = a, b, c;
#m2 = ('b', 'd');
#m = #m1.union(#m2);                  //result: ('a', 'b', 'c', 'd')
#m = #m1.except(#m2);                 //result: ('a', 'c')
#m = #m1.intersect(#m2);              //result: ('b')
```

It is noted that union() avoids introducing dublets in the list, which is not the case regarding extend() and the `+` operator. Instead of these functions, you may use operators `||`, `-` and `&&`:

```
#m1 = ('a', 'b', 'c');                //or: #m1 = a, b, c;
#m2 = ('b', 'd');
#m = #m1 || #m2;                      //result: ('a', 'b', 'c', 'd')
#m = #m1 - #m2;                       //result: ('a', 'c')
#m = #m1 && #m2;                      //result: ('b')
```

If the list `#x1` contains none of the `#x2` elements, the expression `#x1 + #x2 - #x2` will be = `#x1`. However, Gekko lists may contain dublets, and you can use the unique() function to remove these, and the sort() function for sorting. For instance:

```
#m1 = ('c', 'b', 'a');                    //or: #m1 = c, b, a;
#m2 = ('b', 'c', 'd', 'e');
#m = #m1.extend(#m2);                     //result: ('c', 'b', 'a',
'b', 'c', 'd', 'e' )
#m = #m1.extend(#m2).unique();            //result: ('c', 'b', 'a',
'd', 'e')
```

```
#m = #m1.extend(#m2).unique().sort();    //result: ('a', 'b', 'c',
'd', 'e')
```

The above example also illustrates function chaining. The expression `#m1.extend(#m2).unique().sort()` is easier to understand than the equivalent but more backwards expression `sort(unique(extend(#m1, #m2))`.

To check if a list contains an item, use contains() or index():

```
#m = ('a', 'b', 'c');                    //or: #m = a, b, c;
%v = #m1.contains('b');                  //result: 1 (true)
%i = #m1.index('b');                     //result: 2 (the position)
```

You may replace elements in a list, for instance:

```
#m1 = ('ax', 'bx', 'xc');                    //or: #m1 = ax, bx,
xc;
#m = #m1.replace('bx', 'y')                  //result: ('ax', 'y',
'xc')
#m = #m1.replaceinside('x', 'z')             //result: ('az', 'bz',
'zc')
```

The last function, replaceinside(), replaces text inside the individual elements.

List elements may be of mixed types, for instance (1, 'two', 2003q3), and lists of values are often used to feed values into timeseries:

```
time 2020 2026;
x = 100, 101 rep 3, 102, 103 rep *;        //parentheses can be
omitted for simple values
prt x;                                     //result: 100, 101, 101,
101, 102, 103, 103
```

You can use `rep` to repeat values, and `rep *` is special when used to define a series, since it repeats the last item to make the list fit with the sample (here: 7 observations). If you have input data with (a lot of) blank-separated values, you may use the data() function instead of manually setting the commas:

```
time 2020 2026;
x = data('100 101 101 101 102 103 103');    //result: 100, 101,
101, 101, 102, 103, 103
```

Lists can be looped, for instance:

```
#m = ('a', 'b', 'c');                        //or: #m = a, b, c
%s = '';
```

```
for string %i = #m;
  %s += %i;
end;
prt %s;                                      //result: 'abc'
```

Lists can be nested, for instance an #alias list (cf. OPTION interface alias):

```
x = 100;
y = series(1); y[z] = 200;
option interface alias = yes;
global:#alias = (('a', 'x'), ('b', 'y[z]'));
prt a, b; //will be the same as prt x, y[z]
```

An `#alias` list can among other things be convenient as a bridge between the naming conventions of two different models.

Listfiles

Instead of storing lists in a databank, you may optionally use an external file instead. Using a listfile `animals.lst` (you may use `edit animals.lst;` to do this):

```
------------ animals.lst ------
//comments like these are allowed, and blank lines too
dog
cat
mouse
fish
------------------------------
```

You can use the listfile in the following way:

```
#animals = #(listfile animals);
```

Listfiles may contain strings or values, and for simple strings starting with a letter, you may omit single quotes. If you need to make sure that an element is interpreted as a string and not as a value, you can enclose the string in single quotes.

When Gekko reads an element, if it is enclosed in quotes, it always becomes a string. Otherwise, the interpretation rules are the same as for naked lists. When writing elements, if all elements are simple strings starting with a letter, Gekko will omit the quotes. Else it will write strings with single quotes, and dates are written as strings, too. To convert a list of strings into a list of dates, just use the dates() function. Like normal lists, listfiles may contain elements beginning with '-',for instance:

```
------------ items.lst -------
x1
-x2
```

```
x3
-------------------------------
```

But you cannot put complicated items like expressions into a listfile (use a normal list for that). You can create a listfile m.lst with `#(listfile m) = a, b, c;`. Nested listfiles are possible: just separate the elements with the ';' symbol:

```
------------ nested.lst ------
1
2; 3
mouse
2010q1; 2010q3
-------------------------------
```

All these items are converted into strings, because they are not all values. You could create this file with `#(listfile nested) = ('1', ('2', '3'), 'mouse', ('2010q1', '2010q3'))`. The last element can be followed by the ';' symbol, but it may also be omitted. The following list is similar to a matrix:

```
------------ matrix.lst ------
1; 2
3; 4
-------------------------------
```

If `#m = #(listfile matrix)`, for instance `#m[2][1] = 3`, corresponding to row 2, column 1. This is similar to the matrix `#m = [1, 2; 3, 4]`, where `#m[2, 1] = 3`. In general, the listfile format corresponds to .csv, and the reason ',' is not used to delimit elements is that csv files may allow numbers to be stored with decimal separator ',' instead of '.'.

Instead of using a two-dimensional csv-like listfile like `matrix.lst` above, you may instead store such data in an Excel spreadsheet, and load them with `sheet <import list> #m file = ... ;`.

Searching

You may search a list of strings using wildcards, in order to return certain patterns of string elements.

```
#m1 = ('abd', 'abcd', 'abcde');            //or: #m = abd, abcd,
abcde
#m2 = #m1['a*d'];                          //result: 'abd', 'abcd'
#m3 = #m1['a?d'];                          //result: 'abd'
#m4 = #m1.addbank('b1').addfreq('q');      //result: 'b1:abd!q',
'b1:abcd!q', 'b1:abcde!q'
```

As the last line shows, there are a lot of functions available, if you need to handle for instance banknames, frequency indicators, etc. on such lists of names. See under functions, bank/name/frequency/index manipulations. Note that if you need to for instance print the elements of `#m1` with a particular bank and frequency, you may use either `prt {#m1.addbank('b1').addfreq('q')};`, or the easier `prt b1:{#m1}!q;`.

You may use a 'naked' wildcard to obtain variable names from open databanks, for instance:

```
abd = 1; abcd = 2; abcde = 3;
#m2 = ['a*d'];                          //result: 'abcd', 'abd',
list is alphabetical
#m3 = ['a?d'];                          //result: 'abd'
```

The wildcards match variable names found in open databanks. Please keep in mind that the logic regarding such 'naked' wildcards is a bit different from searching inside a list of strings. A wildcard like `['a*d']` will look for series starting with 'a' and ending with 'd', but only in the current first-position databank, and only regarding variables with the same frequency as the current frequency. To get series names from all banks with all frequencies, starting with 'a' and ending with 'd', you would have to use `['*:a*d!*']`. And to get, for instance variables starting with `%` (scalars), and then followed by `'a'` and ending with `'d'`, you would have to use `['%a*d']`. See more on the wildcards page and on INDEX section.

To print out results of wildcards searches, {}-curlies must be used, for instance:

```
abd = 1; abcd = 2; abcde = 3;
prt ['a*d'];                            //prints the two strings
'abcd' and 'abd'
prt {['a*d']};                          //result: same as prt
abcd, abd;
prt {'a*d'};                            //shortcut: the inner []-
brackets may be omitted here
```

As seen above, you must use `{}`-curlies to print out the variables corresponding to a wildcard search. And as shown, you do not have to use the pattern `{['....']}`, but can omit the innermost `[]`-brackets and just use `{'....'}` to print wildcard variables.

Lists and array-series

As a last note, lists can be used to define domains for array-series. See the SERIES section, but the following example illustrates the use:

```
time 2020 2022;
x =
series
```

```
(
1);                                                                   //arra
y-series with 1 dimension
x[a] = 1; x[b] = 2; x[c] = 3;
#i = a, b, c;
p <n> x[#i], sum(#i, x[#i]), sum(#i, x[#i] $ (#i in
#i.remove('b')));
```

The result is the following:

```
                                                                         sum(#
i, x[#i]
                                                                      s    $
(#i in #i.
                 x[a]            x[b]            x[c]   um(#i, x[#i])
remove('b')))
  2020         1.0000          2.0000          3.0000          6.0000
  4.0000
  2021         1.0000          2.0000          3.0000          6.0000
  4.0000
  2022         1.0000          2.0000          3.0000          6.0000
  4.0000
```

So `x[#i]` prints out the three elements, `sum(#i, x[#i])` sums them up, and `sum(#i, x[#i] $ (#i in #i.remove('b')))` show the sum for the list `#i` except the element `'b'`.

You may assign a domain to array-series dimensions with the setdomains() function, and you may restrict which elements are printed/plotted via a special #default map. See more in the SERIES section.

Note

See the page with syntax diagrams if the basics of names, expressions, etc. is confusing.

You can remove `<direct>` in `list<direct>` from older Gekko 2.0/2.2/2.4 code.

You can use minus (`-`) before a list item in a naked list. This can for instance be used in the Laspeyres chain index function laspchain(), to indicate a variable that is to be deducted. For instance:

```
#p = p1, p2, p3, p4;   //same as: #p = ('p1', 'p2', 'p3', 'p4');
#q = x1, x2, -x3, x4;  //same as: #q = ('x1', 'x2', '-x3', 'x4');
#m = laspchain(#p, #q, 2010);
```

The function returns a map `#m` containing the aggregated `p` and `q` series as named elements, where series `x3` is to be deducted from the aggregation (and the price `p` is set to 1 in 2010). The resulting series and quantities can be accessed with `#m.p` and `#m.q`.

If you have input data with blank-separated values, you may use the data() function to avoid having to set the commas, for instance `#m = data('1.0  2.0  1.5');`. This function can be practical for long lists of numbers. Data from an Excel spreadsheet can be loaded into a nested list by means of `sheet <import list>`.

Note that an assignment like `#m1 = #m2;`, where `#m2` is a list, `#m1` will become a *copy* of `#m2`, not a reference to it. The same is the case regarding function arguments, where manipulating `#m1` inside the function body of f(`#m1`) will not affect `#m1` after the function has been left.

The lists `#all` (all model variables), `#endo` (all endogenous variables), and `#exo` (all exogenous variables) and some more are defined beforehand, if a model is loaded by the MODEL statement. These lists are located in the Global databank. See `list ?;`.

The reader may wonder why it is not possible to add a list `#m` and a scalar `%s` like this: `#m + %s`, appending `%s` to `#m`? This is tempting, but then what about `%s1 + %s2 + #m + %s3`? Should this mean a list with `%s1`, then `%s2`, then the elements of `#m`, and lastly `%s3`? But if `%s1` and `%s2` are two strings, these strings are added into another string. And if they are values, the values are added first. Hence, to avoid such confusion, using `+` between a list and a scalar is disallowed (which is also the case in Python and other languages).

Regarding variable types and the Gekko type system, see the VAR section. In this appendix, variable assignment rules, including variable types, is explained in more detail.

Beware that any `rep *` in a list definition will be ignored (`rep *` is designed for timeseries calculations).

A Gekko list relates to the following in different languages:

- Gekko: `#m = ('x', 2.2);`.
- R: list (holds different types of variables), `m <- list("x", 2.2)`. A R vector is similar, but holds only variables of the same type, for instance `m <- c("a","b","c")`. Note that `#m[%i]` in Gekko returns the object at position `%i`, not a 1-element list containing the object, like R. Use `#m[%i..%i]` to get such a R-like slice. In this sense, `#m[%i]` in Gekko corresponds to `m[[i]]` in R.
- Python: list (holds different types of variables), `m = ["x", 2.2]`. Python has a built-in set type, with union(), intersection() and difference() methods. Note that for union(), Python uses operator `|` where Gekko uses `||`, for intersection() Python uses operator `&` where Gekko uses `&&`, and for difference() both Python and Gekko use operator `-` (difference() is called except() in Gekko). Python sets do not allow dublets, but Python lists do not provide methods union(), intersection() and difference().
- Matlab: same type elements: array, else: cell array, `m = {"x", 2.2}`.

Related statements

MAP, MATRIX, FOR

4.3.49 LOCAL

The LOCAL statement is used to designate variable names that are to be located in the Local databank. Following a `local x;` statement, any subsequent use of `x` (without databank designation) will be understood as `local:x`.

After Gekko leaves the program file, function or procedure, these local variables do not live on, so the variables only live in the local context. Therefore, using LOCAL or `local:x = ...` can be practical regarding temporary variables that are not intended to live on, polluting the databanks (such variables are called "side-effects").

You may use `local<all>;` to render all bankless variables local inside a function or procedure. This is useful regarding encapsulation: ensuring that a function only uses its own arguments and local variables (unless a variable is stated with an explicit databank reference). After a `local<all>`, you can still search for a bankless variable `x` outside of the Local databank by means of the special `all:` designation (for instance `y = all:x;`).

See also the similar GLOBAL statement for global variables, and the BLOCK structure for temporary settings.

Syntax

```
local varnames;
local <all>;
```

varnames	Comma-separated list of variables
ALL	(Optional). With this option, all following (in the rest of the program/function/procedure) left-hand side variables without explicit databank designation are located in the Local databank. This may practical for functions/procedures where no temporary variables are supposed to exist after the function/procedure has been left. For a variable `x` that you would like to keep despite using a `local<all>`, you may use `first:x` or another bank designation to circumvent `local<all>`.

Example

```
local x, %y, #z;
```

After this, any use of `x`, `%y`, or `#z` (in the present program file, function or procedure) will be interpreted as `local:x`, `local:%y`, and `local:#z`, respectively. And after Gekko leaves the current context (program file, function or procedure), these local variables cease to exist.

The Local databank is searched first, if databank searching is active (that is, data- or mixed mode), cf. databank search.

You may use `local<all>` in functions/procedures:

```
procedure test;
  local <all>;
  %x = 2;
  %y = 3;
  first:%z = %x + %y;
end;
test;
val?;
```

Only `%z` will exist (in the first-position databank) after the procedure has been left. Still, it is perhaps better to use a function to return the value.

Note

You are not forced to use the LOCAL keyword, when operating with local variables. Defining `local:%per1 = 2010;` first, and referring to `local:%per1` later on is possible, too. In that sense, the LOCAL keyword is just for convenience, especially if `%per1` is used several times.

Local variables survive READ, CLEAR, etc., but do only live in their local context (program file, function or procedure). Hence, they do no 'pollute' the first-position databank if this is later on written to file.

Note that the Local or Global databanks are always searchable, independent of MODE etc.

Related statements

GLOBAL

4.3.50 LOCK

LOCK is used to set a databank non-editable, so that the data inside cannot be changed, but only read. Per default, databanks are opened non-editable, unless you use OPEN<edit>. See also the inverse UNLOCK statement.

Syntax

```
lock databank;
```

Examples

```
lock mybank;
```

This locks `mybank` (sets it non-editable, provided that it is unlocked already).

Related statements

UNLOCK, OPEN

4.3.51 MAP

A map is a data container, much like a mini-databank, convenient for storage, and for passing variables into and out of functions. It can also be thought of as a named list, where the elements are found (looked up) by means of strings (keys), instead of by consecutive numbers 1, 2, ... etc. Map names always start with the symbol '#', like the other collection types list and matrix. If, for instance, you need to return several variables from a user-defined function, a map is a very convenient container for that purpose.

Syntax

```
#m = ( name = expr, name = expr, ... );      //Definition of a map, where
#m is the map name.
map ?;                                       //show/count maps in open
databanks

Refer to the elemements with brackets or dots:

#m['x']               //Refer to element with name (key) 'x'. Blanks and
other symbols can be used.
#m[x]                 //Simple names may omit the single quotes (only when
the name does not have type symbols)
#m.x                  //Simple names may use dot, in that case type
symbols are allowed
```

It is not legal to use for instance `map m = ... ;`, omitting the '#'.

For instance, the result of the OLS statement could be stored in a list `#ols_stats`, where `#ols_stats[1]` could be residual sum of squares, `#ols_stats[2]` could be standard error, etc. This may get confusing, and a map would allow references to look like `#ols_stats['%rss']`, `#ols_stats['%se']`, etc., using expressive names. For simple names, you can also refer to the elements by means of `#ols_stats.%rss`, `#ols_stats.%se`, etc. Written like this, it is seen that there is a similarity to databanks, where such a reference to the databank `ols_stats` would be written as `ols_stats:%r2`, `ols_stats:%se`, etc. The similarity is not coincidal: maps really *are* mini-databanks!

Examples

The following example shows the definition of a map `#m`, consisting of a string and another map, `#mm`. The map `#mm` contains a string with the same name, and a timeseries.

```
time 2009 2012;
#m = (%i1 = 'a', #mm = (%i1 = 'b', <2010 2011> ts = (1, 2)));
p #m.%i1;       //'a', alternatively p #m['%i1'];
p #m.#mm.%i1;   //'b', alternatively p #m['#mm']['%i1']
p #m.#mm.ts;    //1, 2, alternatively p #m['#mm']['ts']
```

Another example could be the Laspeyres chain index function laspchain(), which returns a map containing an aggregated quantity, and an aggregated price.

```
#p = p1, p2, p3;  //or: #p = ('p1', 'p2', 'p3');
#q = x1, x2, x3;  //or: #q = ('x1', 'x2', 'x3');
#m = laspchain(#p, #q, 2010);
```

Now, `#m['q']` and `#m['p']`, or the shorter `#m.q` and `#m.p` refer to the aggregated quantity and price index (as series).

Object orientation

Gekko does not yet provide the possibility of defining classes, with class methods etc. that an object derived from a given class use. But the map collection can be used together with the fact that all Gekko functions implement so-called UFCS, so that a function like for instance f(x, y) can generally be written as x.f(y). This makes a kind of poor man's object orientation possible, consider the following example:

```
function val volume(map #m);
  %volume = 0;
  if (#m.%type == 'square');
    %volume = #m.%size * #m.%size;
  else;
    if (#m.%type == 'box');
      %volume = #m.%size * #m.%size * #m.%size;
    end;
 end;
 return %volume;
end;

#sq = (%type = 'square', %size = 2, %color = 'red');
#bx = (%type = 'box', %size = 3, %color = 'green');
tell '';
tell 'Type {#sq.%type}: size = {#sq.%size}, volume =
{#sq.volume()}, color = {#sq.%color}';
tell 'Type {#bx.%type}: size = {#bx.%size}, volume =
{#bx.volume()}, color = {#bx.%color}';
```

This produces the following output:

```
Type square: size = 2, volume = 4, color = red
Type box: size = 3, volume = 27, color = green
```

Here, map `#sq` is defined as type 'square', whereas the map `#bx` is defined as type 'box'. When the volume() function is called on such a map, a square type will return size^2, whereas a box type will return size^3. Therefore, `#sq.volume()` = 2*2 = 4, whereas `#bx.volume()` = 3*3*3 = 27. Note that `#sq.volume()` could alternatively be written `volume(#sq)`, but the variant `#sq.volume()` has more object method flavor. So in the example, the volume() function called on a square or a box 'knows' how to calculate the volume of that particular object.

Note

See the page with syntax diagrams if the basics of names, expressions, etc. is confusing.

As it is the case regarding array-series, you may omit the single quotes in indexes, for instance `#m[p]` will correspond to `#m['p']`. Please note that this only applies to simple names without symbols, in other words: timeseries names. So do not expect `#m[%s]` to correspond to `#m['%s']` (it does not).

Maps are 1-dimensional, more dimensions are not supported regarding names/keys. If you need to handle multidimensional data, you may look into array-series (or matrices).

Internally in Gekko, maps are represented in the exact same way as databanks, so in the longer run it is expected that it will be possible to convert seamlessly between maps and databanks.

Note that an assignment like `#m1 = #m2;`, where `#m2` is a map, `#m1` will become a *copy* of `#m2`, not a reference to it. The same is the case regarding function arguments, where manipulating `#m1` inside the function body of `f(#m1)` will not affect `#m1` after the function has been left.

Regarding variable types and the Gekko type system, see the VAR section. In this appendix, variable assignment rules, including variable types, is explained in more detail.

A Gekko map relates to the following in different languages:

- Gekko: `#m = (%a = 'x', %b = 2.2);`
- R: named list, `m <- list(a="x", b=2.2)`
- Python: dict, `m = {"a": "x", "b": 2.2}`
- Matlab: struct array or the more general container map.

Related statements

LIST

4.3.52 MATRIX

A Gekko matrix contains two-dimensional cells with numeric values. Matrix names always start with the symbol #, like the other collection types list and map. You can import/export matrices to/from Excel, but only if the cells of the matrix are numerical values. Matrices can be manipulated with the usual linear algebra operators and functions, like adding, multiplying, inverting, etc. See the list of matrix functions in the end of this section.

If you need to manipulate data "cells" of non-numeric values (like for instance an Excel spreadsheet with values, dates and strings), instead of a matrix, you should use a nested list. Nested lists can be of any dimension, and indexing, selecting, looping etc. works similarly for nested lists and matrices.

See the SHEET statement if you need to transfer Excel cells into a matrix (or nested list).

NOTE: Instead of the former SHOW statement for printing matrices, you should use PRT in Gekko 3.0.

Syntax

```
#m = expression;
matrix #m = expression;
matrix < ROWNAMES = #list  COLNAMES = #list > #m = expression;
matrix ?;                                  //show/count matrices in open
databanks
```

It is no longer legal to use for instance `matrix m = ... ;`, omitting the '#'.

ROWNAMES =	A list containing the names (labels) of the rows of the matrix, to be shown with PRT. You may use quotes (') when creating the list elements, if you need special characters like blanks etc.
COLNAMES =	A list containing the names (labels) of the cols of the matrix, to be shown with PRT. You may use quotes (') when creating the list elements, if you need special characters like blanks etc.
name	The name of the matrix.
expression	Any expression. You may omit the expression if you just need to decorate an already existing matrix with row- or

	colnames.
?	Query regarding a particular matrix, or all matrices

Note to Excel users: to import data from a spreadsheet, use the SHEET statement (SHEET<import matrix>). To export a matrix `#m` from Gekko to Excel, you can use `export <xlsx> #m file = matrix.xlsx;`.

Examples

The MATRIX statement supports quite a lot of the more common matrix capabilities. More capabilities will be added over time. Regarding matrix functions, please consult the Gekko functions page, under the 'Matrix functions' section (for instance determinant, inverse, transpose, diagonal, summation, etc.). These functions are also shown at the end of this help page.

You may construct a 2x2 matrix like this:

```
#m = [1, 2; 3, 4];  //or: matrix #m = [1, 2; 3, 4];
```

The commas separate the row items, and the ';' separates columns. The MATRIX keyword may be omitted, since the right-hand side is guaranteed to be a matrix defintion. To construct matrices, you may use values or other matrices instead of the fixed numbers shown here. In general, matrices are referred to by means of the '#' indicator, just like lists. Use PRT to print out a matrix:

```
prt #m;
```

This will print out the following:

```
 #m
                      1                 2
     1           1.0000            2.0000
     2           3.0000            4.0000
```

Note: after printng a matrix like this, you may use Copy-button in the main Gekko window to copy/paste the matrix to Excel. To decorate with custom row- and col-names, you may do the following (you may alternatively use listfiles to contain the labels, cf. LIST):

```
#rn = ('Agriculture', 'Services etc.');
#cn = ('Employed', 'Unemployed');
```

```
<rownames=#rn colnames=#cn> #m = [1, 2; 3, 4];
prt #m;
```

This will print the following labeled matrix:

```
 #m
                       Employed     Unemployed
  Agriculture            1.0000         2.0000
  Services etc.          3.0000         4.0000
```

You may concatenate existing matrices like this:

```
#a = [#m1; #m2];                //column-wise
#b = [#m1, #m2];                //row-wise
#c = [#m1, #m2; #m3 ,#m4];      //both
```

You may get a list of all matrices or a particular matrix with

```
matrix ?;
```

You may construct matrices filled with 0's, 1's or missing values by means of the functions zeros(n, k), ones(n, k), or miss(n, k), for instance:

```
#m = zeros(5, 10);
```

You can use `+`, `-`, `*` and `/` on two matrices, to add, subtract, multiply or divide. Regarding division, you can only divide a matrix by a value or a 1x1 matrix. Otherwise, use the element-by-element functions multiply() and divide().

You may index a matrix by means of the indexer []. For instance:

```
%v = #m[2, 4];
```

This picks out the element in row 2, column 4. Please note that the indexes are 1-based. The inverse operation:

```
#m[2, 4] = %v;
```

Sub-matrices can be picked out by means of the range dots ('..'), for instance:

```
#m2 = #m[1..2, 5..7];
```

This picks out rows 1 and 2, and combines them with columns 5, 6 and 7. Note that these ranges may not be descending, for instance `#m[2..1, 7..5]`. The inverse operation:

```
#m[1..2, 5..7] = #m2;
```

To select a full row or column, use an 'empty' range like this:

```
#m2 = #m[3, ..];
```

This selects all the items in row 3. You may also use for instance '2..' to pick out the elements from 2 and onwards, or '..10' to pick out the elements from 1 up to and including 10. Use '..' to pick out all rows/columns.

For a column vector `#c` (that is, a n x 1 matrix), you may omit the column index, so in that case, these two will amount to the same:

```
%v = #c[5];
%v = #c[5, 1];
```

You may pack and unpack matrices from timeseries, for instance:

```
reset;
time 2001 2003;
x1 = 1, 2, 3;
x2 = 3, 4, 5;
p #m;
#m = pack(2001, 2003,x1, x2);
y1 = #m[.., 1].unpack(2001, 2003);
y2 = #m[.., 2].unpack(2001, 2003);
p<n> x1, y1, x2, y2;
```

This will pack the two timeseries `x1` and `x2` into a 3 x 2 matrix #m (with data from 2001-2003). You may unpack back to two timeseries again with the unpack() function as shown. The indexes `[.., 1]` and `[.., 2]` pick out all rows of the two columns in `#m`. You may also consult the pack/unpack example in the R_RUN section. Column vectors can be handy, when you use them as a list of values:

```
#m = [100; 150; 120];
for val %i = 1 to #m.rows();
  tell 'Index {%i} has value {#m[%i]}';
end;
```

This will print out the numbers 100, 150 and 120:

```
Index 1 has value 100
Index 2 has value 150
Index 3 has value 120
```

Since `#m` is a column vector, you may use `#m[%i]` instead of the more cumbersome `#m[%i, 1]`.

There are min, max, avg and sum functions, working on rows or columns. For instance, you may decorate a matrix with grand totals like in the code below (where the third row and column are totals). The functions sumr() and sumc() sum the rows and columns, respectively. The last expression, `#m.sumc().sumr()`, could just as well have been stated as `#m.sumr().sumc()`. The `divide(#m1, #m2)` function divides two matrices element by element, but `#m2` may have only 1 row or column stated. In that case, the function works on rows or columns, respectively.

```
#m = [1, 2; 3, 4];
prt [#m, #m.sumr(); #m.sumc(), #m.sumc().sumr()];
prt divide(#m, #m.sumr());
prt divide(#m, #m.sumc());
```

Output:

```
[#m, #m.sumr(); #m.sumc(), #m.sumc().sumr()]
                   1              2              3
    1         1.0000         2.0000         3.0000
    2         3.0000         4.0000         7.0000
    3         4.0000         6.0000        10.0000

divide(#m, #m.sumr())
                   1              2
    1         0.3333         0.6667
    2         0.4286         0.5714

divide(#m, #m.sumc())
                   1              2
    1         0.2500         0.3333
    2         0.7500         0.6667
```

In the first print, the rows sum to 1, and in the second print, the columns sum to 1. Matrices can be exported and imported from Excel, for instance:

```
#m = [1, 2, 3; 4, 5, 6];
export <xlsx> #m file = m.xlsx;
sheet <import matrix> #m2 file=m.xlsx;
prt #m, #m2;
```

The example below estimates a linear least squares model with five parameters. You may consult the OLS section to see the same parameters calculated via the OLS solver, or the R_RUN section to see the same parameters calculated via the R interface.

```
reset;
create lna1, pcp, bul1;
lna1 <1998 2010> = data(' 166.223000  173.221000  179.571000
187.343000  194.888000  202.959000
209.426000  215.134000  222.716000  230.520000  238.518000
246.654000  254.991000') ;
pcp <1998 2010>   = data(' 0.9502030   0.9699920   1.0000000
1.0235000   1.0401100   1.0605400
1.0754700   1.0977800   1.1121200   1.1314800   1.1513000
1.1717600   1.1871600')  ;
bul1 <1998 2010>  = data(' 0.0684791   0.0591698   0.0560344
0.0535439   0.0535003   0.0631703
0.0649875   0.0578112   0.0473207   0.0404508   0.0467488
0.0472923   0.0475191')  ;
time 2000 2010;
create s0, s1, s2, s3, s4, s5;
s0 = dlog(lna1);
s1 = dlog(pcp);
s2 = dlog(pcp.1);
s3 = bul1;
s4 = bul1.1;
s5 = 1;
#x = pack(2000, 2010, s1, s2, s3, s4, s5);
#y = pack(2000, 2010, s0);
#b = inv(t(#x)*#x)*t(#x)*#y;  //OLS formula
prt #b;
```

The statements produce the following parameter estimates:

```
 #b
                       1
     1           0.1445
     2           0.6139
     3           0.1867
     4          -0.3509
     5           0.0298
```

Matrix functions

Matrix functions:

Function name	*Description*	*Examples*
avgc(x)	Average over cols.	`#m2 = avgc(#m1);`

	Returns: matrix	
avgr(x)	Average over rows Returns: matrix	`#m2 = avgr(#m1);`

chol(x)
chol(x, type)
	Cholesky decomposition of matrix x. Accepts type (string), either 'upper' or 'lower'.
	Returns: matrix
		`#m2 = chol(#m1, 'upper');`

cols(*x*)	Returns the number of colums of *x* Returns: val	`%v = cols(#m);`

design(x)	Returns a "design" matrix, equivalent to the same Gauss function. The function is practical for aggregating rows or columns.
	
	Input: is a n x 1 column matrix
	Returns: n x k matrix of 0's and 1's. The input numbers specify the columns in which the 1's should be placed.
	
	In the example, a 4 x 5 matrix `#x` of 1's is defined, and by means of the aggregation matrix `#m`, it is aggregated from size 4 x 5 to 4 x 3. In this case, the new column 1 is the old columns 2 and 3, the new column 2 is the old column 5, and the new column 3 is the old columns 1 and 4.
		`#a = [3; 1; 1; 3; 2];`
		`#m = design(#a);`
		`#x = ones(4, 5);`
		`p #m, #x * #d;`
		
		`#m`
		`                  1`
		`  2                 3`
		`  1           0.0000`
		`0.0000          1.0000`
		`  2           1.0000`
		`0.0000          0.0000`
		`  3           1.0000`
		`0.0000          0.0000`
		`  4           0.0000`
		`0.0000          1.0000`
		`  5           0.0000`
		`1.0000          0.0000`
		
		`#x * #d`
		`                  1`
		`  2                 3`
		`  1           2.0000`
		`1.0000          2.0000`
		`  2           2.0000`
		`1.0000          2.0000`
		`  3           2.0000`
		`1.0000          2.0000`
		`  4           2.0000`
		`1.0000          2.0000`

det(x)	Determinant of a matrix. Returns: val	`%v = det(#m);`
diag(x)	Diagonal. If x is a n x n symmetric matrix, the	`#m2 = diag(#m1);`

	method returns the diagonal as a n x 1 matrix. If x is a n x 1 column vector, the method returns a n x n matrix with this column vector on the diagonal (and zeroes elsewhere). Returns: matrix	
divide(x1, x2)	Element by element division of the two matrices. If x2 is a row vector, each x1 column will be divided with the corresponding value from the row vector. And if x2 is a column vector, each x1 row will be divided with the corresponding value from the column vector. Returns: matrix	`#x = divide(#x1, #x2);`
i(n)	Returns a n x n identity matrix. Returns: matrix	`#m = i(10);`
inv(x)	Inverse of matrix x Returns: matrix	`#m2 = inv(#m1);`
maxc(x)	Max over cols Returns: matrix	`#m2 = maxc(#m1);`
maxr(x)	Max over rows Returns: matrix	`#m2 = maxr(#m1);`
minc(x)	Min over cols Returns: matrix	`#m2 = minc(#m1);`
minr(x)	Min over rows Returns: matrix	`#m2 = minr(#m1);`
m(r, c) or miss(r, c)	Returns a n x k matrix filled with missing values. Cf. also m() function for values. Returns: matrix	`#m = m(5, 10);`

multiply(x1, x2)	Element by element multiplication of the two matrices. If x2 is a row vector, each x1 column will be multiplied with the corresponding value from the row vector. And if x2 is a column vector, each x1 row will be multiplied with the corresponding value from the column vector. Returns: matrix	`#x = multiply(#x1, #x2);`
ones(n, k)	Returns a n x k matrix filled with 1's Returns: matrix	`#m = ones(5, 10);`

pack(v1, v2, ...)
pack(<t1 t2>, v1, v2, ...)
	Using period t1-t2, the timeseries v1, v2, ... are packed into a n x k matrix, where n is the number of observations and k is the number of variables. If the period is omitted, the global time period is used.
	Returns: matrix
		`#m = pack(<2020 2030>, x, y, z); Returns: a 11 x 3 matrix #m with the values.`

rows(*x*)	Returns the number of rows of *x*. Returns: val	`%v = rows(#m);`
sumc(x)	Sum over cols Returns: matrix	`#m2 = sumc(#m1);`
sumr(x)	Sum over rows Returns: matrix	`#m2 = sumr(#m1);`
t(x)	Returns the transpose of a matrix. Returns: matrix	`#m2 = t(#m1);`
trace(x)	Returns the trace of a matrix. Returns: val	`%v = trace(#m);`
unpack(m) unpack(<t1	The column matrix m (with only one column) is	`//This picks out the second column of #m (and all the rows).`

t2>, m)	unpacked into a timeseries spanning the period t1-t2. If the period is omitted, the local/global time period is used.
	The unpack() function is not strictly necessary: you may alternatively assign a nx1 matrix directly to a series (see example).
	Returns: series
		`y = #m[.., 2].unpack(<2020 2030>);`
		`y <2020 2030> = #m[.., 2].unpack();  //same`
		`y <2020 2030> = #m[.., 2]; //also works`

zeros(n, k)	Returns a n x k matrix filled with 0's. Zeroes() can be used as alias. Returns: matrix	`#m = zeros(5, 10);`

Note

See the page with syntax diagrams if the basics of names, expressions, etc. is confusing.

You may use m() to indicate a missing value for a matrix element. You can also use m(r, c) to get a matrix with missing values.

Note that an assignment like `#m1 = #m2;`, where `#m2` is a matrix, `#m1` will become a *copy* of `#m2`, not a reference to it. The same is the case regarding function arguments, where manipulating `#m1` inside the function body of `f(#m1)` will not affect `#m1` after the function has been left.

Matrices are printed with PRT, the former SHOW statement is obsolete.

Regarding variable types and the Gekko type system, see the VAR section. In this appendix, variable assignment rules, including variable types, is explained in more detail.

A Gekko matrix only supports numeric values inside the cells. It relates to the following in different languages:

- Gekko: `#m= [1, 2; 3, 4];`
- R: matrix data structure, `m = matrix(c(1, 2, 3, 4), nrow=2, ncol=2)`
- Python: list of lists (with numeric values), `m = [[1, 2], [3, 4]]`
- Matlab: two-dimensional array of numeric values

Related statements

VAL, SERIES, LIST, SHEET

4.3.53 MEM

Prints a list of all scalar variables (value, date, string). Before Gekko 3.0, such scalar variables were stored in memory (ram), hence the name of the statement. In Gekko 3.0, scalars are stored in databanks, like series and other variable types.

Syntax

```
mem;
```

Example

Consider this example:

```
%s1 = 'cat';
%s2 = 'dog';
%v = 123.45;
%d1 = date(2010); //or use 2010a or 2010a1 to make it a date.
%d2 = 2011q3;
mem;
```

This will produce an overview similar to this:

```
6 scalars found in 'Work' databank
---------------------------------
 type         name      value
---------------------------------
 DATE         %d1       2010
 DATE         %d2       2011q3
 STRING       %s1       'cat'
 STRING       %s2       'dog'
 VAL          %v        123.45
---------------------------------
```

Note

For scalars, you may use VAL?, DATE?, STRING?, to print out these individually.

Otherwise, you can also print scalars with the PRT statement.

Related statements

STRING, DATE, VAL

4.3.54 MENU

MENU is not a statement, but the 'Menu' tab will open up a .html menu file when clicked (regarding the .html file, see 'Related options' below). Menus can for instance be used to organize tables (.gtb) in hierarchies, or call Gekko program files (.gcm). The easiest way to start up the menu system is to click on the 'Menu' tab. Another possibility is via 'Window' --> 'Restart Menu' (in the Gekko user interface).

Menu's are browsable, i.e., you may use the backwards and forwards arrows. The 'Home' button will point the menu to the starting .html file.

Details

The html file will be shown in the 'Menu' tab in the same way as the html file would be shown in an internet browser. The only real difference is that links work in a slightly different way. If the link is to a file with extension .gtb, Gekko will show that particular table (as text in the 'Main' tab, or as html in the 'Menu' tab, depending upon the setting `option table type...`).

For instance, the html may contain the following HTML code:

```
<a href="s43.gtb">Production</a>
<a href="scenario1.gcm">Run scenario 1</a>
```

The first `<a>` tag indicates a link, with link text "Production". The linked file is `s43.gtb`. Since this is a .gtb file, Gekko shows this particular table when the link is clicked. Instead of a .gtb file, you may indicate a .gcm file (program file) inside the quotes. In that case, if the link is clicked, Gekko will RUN the .gcm file.

You may design the HTML pages in any way you like, for instance it can be convenient to style menus by means of a common stylesheet (CSS), so that their design can be controlled centrally. Gekko will show just about all legal HTML code you may come up with, including images etc.: the engine showing the HTML in the 'Menu' tab is in reality the same engine that is used for showing pages in Internet Explorer.

The HTML files can be made by means of any HTML editor, for a free and quite robust editor, you may for instance try the free Kompozer.

There is an automatic menu conversion tool from the older PCIM menus to the new HTML format. See the menu: 'Utilities' --> 'Converters' --> 'PCIM converters' --> 'Convert PCIM menus...'.

Example file

The HTML code below shows a menu ('OVERVIEW') with five items. The two first items link to two different submenus (menu1/menu2.html), whereas the two following links link to two different tables (s1/s2.gtb). The last item gives the opportunity to browse a level up to a parent menu (main.html).

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <link rel="stylesheet" href="styles.css" type="text/css">
    <meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1">
    <title>OVERVIEW</title>
  </head>
  <body>
  <big><b>OVERVIEW</b></big><br>
    <ul>
      <li><a href="menu1.html">SUPPLY BALANCE</a></li>
      <li><a href="menu2.html">IMPORTANT KEY FIGURES ETC.</a></li>
      <li><a href="s1.gtb">Production <img src="table.png"><font
color="gray"> S1 </font></a></li>
      <li><a href="s2.gtb">Productivity <img src="table.png"><font
color="gray"> S2 </font></a></li>
      <li><a href="main.html">***MAIN MENU***
             Back</a></li>
    </ul>
  </body>
</html>
```

You may run Gekko program files via the menus, for instance adding the following list item to the menu system:

```
<li><a href="menutest.gcm">Run menutest.gcm</a></li>
```

This will provide a link ("Run menutest.gcm"), and when the link is clicked, menutest.gcm is executed.

The HTML file uses a stylesheet (styles.css) which can be common the all the HTML files in the menu system. In addition, a small table icon is used (table.png), to indicate that the item points to a table. The stylesheet styles.css could be something like this:

```
body {
  color: #000000;
  font-family: Verdana;
  font-size: 10pt;
  font-style: normal;
  font-variant: normal;
  background-color: white;
}
a {text-decoration: none;}
img {border-style: none;}
```

This sets colors, font, size etc., and indicates that HTML links (`<a>`) should have no underline, and HTML images (`<img>`) no border.

Note

A menu system may be organized in folders and sub-folders, if preferred. In that case, call the html file in the subfolder with `<a href="subfolder/menu1.html">` for instance. To navigate to a parent folder, use '..', for instance `<a href="../menu1.html">`. These are standard html conventions regarding relative paths.

Anything can be put into the html file, but for security reasons Gekko will not open up Internet links in the Menu tab. If an external Internet link is present (for instance http://www.t-t.dk/gekko), Gekko will open that link with the default web browser.

You may use the EDIT statement to edit the files (for instance: `EDIT menu.html;`), but using a special html editor may be easier.

Related options

OPTION menu startfile = menu.html;
OPTION folder menu = ...;
OPTION interface table operators = yes; //click transformations on tables

Related statements

TABLE, EDIT

4.3.55 MODE

The MODE statement switches between *sim* mode (model simulation), *data* mode (data revision programs etc.), and *mixed* mode (mixing sim- and data mode). Mixed mode is default. The modes are reflected on the status bar at the bottom of the main window (cf. the colors in the table below: green, blue or yellow).

- **Sim mode**: Focused on MODEL, READ, SERIES, SIM, PRT/MULPRT, WRITE and similar statements. The Ref (reference) databank is essential in sim mode (to show simulation differences/multipliers). See here regarding an overview of statements that have primarily *sim* flavour.
- **Data mode**: Focused on OPEN, COPY, IMPORT, SERIES, PRT, CLOSE and similar statements. The reference bank is often not used at all in this mode. See here regarding an overview of statements that have primarily *data* flavour.
- **Mixed mode**: models and data handling can be mixed as the user wishes. Please note that this mode is more flexible, but also has more room for errors, if care is not taken (for instance whether a variable is a model variable, or whether a variable is from the first-position databank or stems from some other open databank).

It should be emphasized, however, that most statements and functionality can be used in all modes, but there are some nuances. Below, an overview of the settings associated with the three different modes.

Mode	**Sim**	**Data**	**Mixed**
option databank search = ... In sim-mode, the user should use READ ... TO ... (which is equivalent to OPEN) to open extra 'named' databanks, and in this case, explicit databank colon must be used afterwards to refer to the timeseries. In data- and mixed mode, Gekko will search for a timeseries `x` (without databank colon) in all databanks except Ref in the F2 window.	no	yes	yes
option databank create auto = ... In sim-mode, the user has to first CREATE a new timeseries, before putting data into it with the SERIES statement. This is to avoid that the user accidentally issues a "SERIES x = ... ;" statement, thinking that he or she changes a model variable, when in fact x is not part of the model. (If the timeseries name starts with 'xx', CREATE is not mandatory in sim mode). In data- and mixed modes, a SERIES statement will auto-create the timeseries, if it does not exist beforehand.	no	yes	yes

option solve data create auto = ... In sim-mode, when the user issues a general READ statement, any model variables (contained in the list `#all`) not present in the data file will be auto-created (including missing variables of D-, J-, and Z-types). In data-mode, such creation is not performed, even if a model is present.	yes	no	yes
In addition to this, '**OPTION interface mode**' is set to sim/data/mixed. This option directs the following behavior: • In sim-mode, READ ... TO ... is recommended instead of OPEN. OPEN<edit>, OPEN<first> or OPEN<ref> are warned against. • In sim- and mixed-mode, general READ will tell the user about superfluous variables not in the model. • In data-mode, general READ is warned against • In data-mode, MODEL, SIM, CLONE and MULPRT are warned against.			

Gekko starts out in mixed mode per default. You may set the mode in the gekko.ini file (see INI). If you need Gekko to always start out in a particular mode, you can use a gekko.ini containing for instance the statement `mode sim;` in the same folder as the gekko.exe file.

Syntax

```
mode  mode;
mode ?;
```

mode	Choose between `sim`, `data` or `mixed`. Default is `sim` mode.
?	Shows the current mode.

Example

This statement changes to `sim` mode:

```
mode sim;
```

Note

The MODE functionality will be continuously developed, but the intention is to avoid making modes more complicated than they really are. Modes try to help the user focus on the tasks at hand, rather than being confused about non-relevant Gekko capabilities.

See also the databank search page.

Note that the Local or Global databanks are always searchable, independent on MODE etc.

Related options

```
OPTION databank search = ...
OPTION databank create auto = ...
OPTION solve data create auto = ...
OPTION interface mode = [sim | data | mixed];
```

Related statements

RESET, RESTART, INI

4.3.56 MODEL

The MODEL statement is used in one of two ways:

- Load Gekko equations contained in a .frm file
- Load GAMS equations stored in a GAMS file. GAMS models are not solved, but their equations can be displayed or decomposed. GAMS equations can either be loaded as raw equations or unrolled equations, cf. the GAMS section below.

A MODEL statement loading a Gekko model is best put before READ statements: in that way all model variables not found in the databank will be auto-created when issuing the READ statement.

Failsafe note: if you experience problems solving the model (SIM), try using the Gauss-Seidel method (default) together with the setting `option solve failsafe = yes`. Gekko model equations may be executed individually (one by one) with the PREDICT statement.

Because Gekko uses a so-called cache for model files, the same file will load faster the next time it is read (this is particularly noticeable for large GAMS scalar models).

Syntax

model < DEP=... DUMP GMS > *filename* ;

DEP=	(Optional). Together with the <gms> option, the user can provide a list that identifies what variables individual equations determine. For instance, in a GAMS equation `e_1` specified like `e_1[i, t] .. p[i, t] * q[i, t] =E= v[i, t];` there is the question of which variable is determined in the `e_1` equation? In a system of simultaneous equations, what is determined is a complicated question, but in many cases it makes sense to designate a "dependent" variable. In the `e_1` equation, we would expect the dependent variable to appear on the left-hand side of the equation, but then there is the question of whether this is `p` or `q`? Gekko can try to find the dependent variable in two ways, controlled by `option model gams dep method = ...` . Either it will look for the first variable that is not inside a []-bracket or $-condition; in the above case `p`. Alternatively, it will look at the equation name, and if this is for instance `e_p`, it will assume that `p` is the dependent variable. If these rules do not designate the "right" variable, you may use the `<dep>` local option like this: `model <gms dep=#(listfile dep)> model.gms;`. In the listfile (you may alternatively use a normal list), you can state lines like this: `p; e3; e10;`

	`q; e2;` This tells Gekko that equations `e_3` and `e_10` designate `p` as dependent variable, whereas equation `e_2` designates `q` as dependent variable. These designations overrule the above-mentioned logic (either picking the first variable on the left-hand side, or using the equation name), so only the special cases that do not follow that rule need to be stated. For instance, if `e_1` was determining `q` instead of `p`, the `q`-line of the listfile could be `q; e_1; e_2`. instead of just `q; e_2`. When using the first variable on the left-hand side, Gekko will allow lagged or leaded variables to be dependents. This can be switched off with `option model gams dep current = yes`, after which only current (non-lagged and non-leaded) variables can be identified as dependents. Note: The identification of "dependents" is only used in the DISP and DECOMP statements, not elsewhere. [New in 3.0.2]
DUMP	(Optional). With this option together with the <gms> option, a GAMS model will be dumped in Gekko form, as the file `dump.gcm`. This file can be used to inspect how Gekko translates GAMS-equations, and which variables are indentified as dependents in each equation (and if they are identified via a DEP list). These equations may not run in Gekko, at the moment <dump> is only for debugging. [New in 3.0.2]
GMS	(Optional). With this option, Gekko will read GAMS equations from a .gms file (extension .gms will be added if missing). Gekko will search the .gms file for equations which can be shown with the DISP statement. A MODEL<gms> statement, will set `option model type = gams;` automatically. You may load a GAMS scalar model like for instance `scalar.zip` with `model <gms> scalar.zip;`, see at the bottom of this help page.
filename	Filenames may contain an absolute path like `c:\projects\gekko\bank.gbk`, or a relative path `\gekko\bank.gbk`. Filenames containing blanks and special characters should be put inside quotes. Regarding reading of files, files in libraries can be referred to with colon (for instance `lib1:bank.gbk`), and "zip paths" are allowed too (for instance `c:\projects\data.zip\bank.gbk`). See more on filenames here. If the filename is set to '*', you will be asked to choose the file in Windows Explorer. The extension .frm is automatically added, if it is missing.

Gekko models

Model equations may be decorated with formula codes, and there are also some special markers to indicate different parts of the model:

Equation types

_	These formula codes begin with the character `_`, and the following codes may follow: • position 2: an arbitrary character to indicate the type of equation, for instance `I` for identity, `G` for 'guessed' equation, `S` for stochastic, etc. The chosen characters are completely ignored in simulations etc. and can be thought of as meta information. • position 3 and 4: add-factor, either `J_` or `JD` for additive add-factors, or `JR` for relative add-factors. • position 5: indicate exogenization dummy, `D`. • position 6: not used • position 7: a `Z` for damping in the Gauss-Seidel algorithm Example: `_SJRD_Z` means `S` for stochastic, `JR` for relative add-factor, `D` for exogenization dummy, and `Z` for damping.
Y	An equation with code `Y` means that the equation is run *after* the simulation (for each period). Such Y-type equations are often of technical character, calculating different kinds of add-factors (J-variables) etc.
T	Equations with code `T` are run exactly like Y-type equations. The only difference is that the `T` indicates that such equations are "interesting" and should be part of documentation systems, etc. T can be read as "table variables".
P	Equations with code `P` are not run at all in SIM, since they are not considered part of the model. P-type equations can be used to store mathematical relationships in frm files, together with the normal equations. P-type equations can only be run via the PREDICT statement, and can therefore be thought of as "predict equations". Examples may be an equation like `FRML P    ye = (y[-1] + y + y[+1])/3;`, calculating a moving average of `y`, including leads. Such an equation will trigger a forward-looking algorithm unless stated as a P-type equation. (The equation may still be run as part of a SIM statement, if the RUNAFTER$ part of the model contains a PREDICT of the equation, cf. example below).

Model parts (tags)

Note: The AFTER$ and AFTER2$ tags are placed between normal model equations (which are located at the top of the file). The RUNBEFORE$, RUNAFTER$ and VARLIST$ tags must be placed *after* the normal model equations, and these tags separate each other, so they do not need an ending tag.

AFTER$	This tag indicates that the following variables are to be understood af part of the 'AFTER$' equations. The AFTER$ distinction is only used in the special SIM<after> statement, not anywhere else (a normal SIM statement ignores the tag). See the note above.
AFTER2$	This tag indicates that the following variables are to be understood af part of the 'AFTER2$' equations. In Gekko, there is no difference between AFTER$ and AFTER2$. The AFTER2$ distinction is only used in the SIM<after> statement, not anywhere else (a normal SIM statement ignores the tag). See the note above.
RUNBEFORE$	After a RUNBEFORE$ tag, you may put normal Gekko statements that are to be executed just before the model is simulated (SIM). In these statements, you may refer to the dates `%__simt1` and `%__simt2` which indicate the start and end of the simulation period. See the note above.
RUNAFTER$	After a RUNAFTER$ tag, you may put normal Gekko statements that are to be executed just after the model is simulated (SIM). In these statements, you may refer to the dates `%__simt1` and `%__simt2` which indicate the start and end of the simulation period. The RUNAFTER$ tag must be placed *after* the normal model equations. See the note above.
VARLIST$	You may put a variable list after a VARLIST$ tag, instead of using a separate file for this. See the note above.

The MODEL statement automatically analyzes and splits the model into three internal parts: the prologue (pre-model), the simultaneous part, and the epilogue (post-model). So Gekko will automatically detect any pre- and post-model and will use this information to speed up simulations (hence, for SIM purposes, AFTER$ and AFTER2$ tags are ignored). The simultaneous part of the model is sub-divided into two parts: the feedback variables, and the simultaneous recursive variables. This information is used to speed up the Newton algorithm, by reducing the dimensionality of the simultaneous equations.

The MODEL statement used on a Gekko model emits some files containing lists of exogenous, endogenous, and DJZ-type variables (that is, add-factors and variables used for exogenization). It also emits a file with ordering information. These files are assembled into a .zip-file with the name `[model]__info.zip`, where `[model]` is the name of the model. Gekko also creates the lists `#all`, `#endo`, `#exo`, `#exod`, `#exodjz`, `#exoj`, `#exotrue` and `#exoz` with lists of different kinds of variables. The lists are located in the Global databank, so that they survive READ statements.

The Gekko model equations have the following syntax:

FRML code variable = expression;

code	See the above table regarding equation types
variable	A variable, or an expression on a variable: log(x), dlog(x), dif(x) or pch(x)
expression	Mathematical expression

As an example to illustrate, we will consider a particularly simple equation:

```
FRML _SJ_D  c  = 0.4 * y + 0.4 * y[-1];
```

The equation code is `_SJ_D`, which means that an absolute add factor (`Jc`) will be added to the equation. So our equation is augmented into

```
c  = 0.4 * y + 0.4 * y[-1] + Jc;
```

Last, exogenization dummies are added, too (the `D` in the fifth position of the equation code). This is done in order to ease exogenization of the equation, if needed. In general, such variables are implemented in the following way:

```
c  = (1 - Dc) * (0.4 * y + 0.4 * y[-1] + Jc) + Dc * Zc;
```

Two more variables are added, `Dc` and `Zc`. If `Dc = 0`, the variables `Dc` and `Zc` have no effect, but if `Dc = 1`, the equation reduces to `c = Zc`. In that case, the equation is controlled by the exogenous variable `Zc`. In addition to this equation, Gekko adds two more 'reverted' equations regarding `Jc` and `Zc`. These equations are of Y-type, run after the simulation.

```
FRML Y      Jc = c - (0.4 * y + 0.4 * y[-1]);
FRML Y      Zc = c;
```

If `Dc = 0`, it is easy to see that `Jc` will not change its value via its Y-type equation. But if `Dc = 1`, `Jc` will attain the value that corresponds to what `Zc` has been set to, or `Jc = Zc - (0.4 * y + 0.4 * y[-1])`. Therefore, you may set `Dc = 1`, and `Zc` to some chosen value. When you simulate, `c` will attain that value. Later on, if you reset `Dc = 0` and simulate, you will notice that `c` still assumes the chosen value (but is otherwise free to change if variables on the right-hand side change). So these exogenization dummies can be used to change the levels of endogenous variables, in a sense performing an implicit goals-means analysis, where `c` is the goal, and `Jc` is the means.

The same functionality is often used for log-linear equations. Such an equation would look like:

```
FRML _GJRD Dlog(y) = Dlog(x);
```

The add-factor (J-variable) is now relative (code `JR`), and the resulting equations are as follows:

```
y = y[-1] * exp(log(x) - log(x[-1])) * (1 + JRy);
```

This equation is the 'unfolded' equation used in the simulations (try taking `log()` on both sides of the equation). In general, a JR-variable is always implemented by multiplying the unfolded equation with `(1 + JR)`. Note that in this equation, the relative add-factor corresponds to `Dlog(y) = Dlog(x) + log(1 + JRy)`, so the relative add-factor can be interpreted as an additive add-factor in logarithms (also note that `log(1 + JRy) ≈ JRy` for small values of JRy).

If in doubt about how a particular equation unfolds, including how D-, J- and Z-variables are implemented, you may DISP it and click the link "Show detailed equation".

You may use parameter values instead of numerical values, for instance:

```
val %c = 0.758812;  //Note: you must use the VAL keyword in frml files
FRML _i Y = %c*X1 + (1-%c)*X2;
```

The parameter value must be located inside the model file itself, either before or after the FRML statement(s) using it.

Gekko models: supported functions

In addition to the normal functions like log(), exp(), abs(), pow(), the following functions are possible:

Name	Description	Possible on left-hand side

dif(x) or diff(x)	Absolute time-difference	yes

difa(x) or diffa(x)
dify(x) or diffy(x)
	Yearly absolute time-difference. All four functions do the same thing.	yes

dlog(x)	Logarithmic time-differences	yes

dloga(x)
dlogy(x)
	Yearly logarithmic time-differences. You may use dlogy() as alias.	yes

lag(x, lag)	Lags x a number of periods. Note the sign of the lag: lag(x, 2) = x[-2]. Can be used if x is an expression.	no
max(x1, x2)	Max of the two values	no
min(x1, x2)	Min of the two values	no
movavg(x, lags)	Moving average	no
movsum(x, lags)	Moving sum	no
pch(x)	Percent time-difference	yes

pcha(x)
pchy(x)
	Yearly percent time-difference. You may use pchy() as alias.	yes

Gekko models: variable list

You may add a variable list with variable explanations at the end of the model file (.frm), after a `VARLIST$` (or `VARLIST;`) tag. The list can be 'folded', using indices like qJ{j}{i}. The list must be in UTF-8-format, if for instance 'æ', 'ø' and 'å' or other special characters are to be shown correctly. In Notepad, you can choose encoding UTF-8 under 'Save as'. These variable explanations show up in for instance DISP or DECOMP.

An example:

```
FRML Qu = ......... ;
FRML fMz01 = ......... ;

VARLIST$
----------
Qu
Antal beskæftigede i alle erhverv ekskl. landbrug mv.
(1000 pers.)
Kilde: Statistikbanken, NAT18, branche: jf. fX{j}

----------
fMz{i}     i=01,2,3q,59,s
Den del af importgruppe {i}, der har en generel
substitutionselasticitet til dansk produktion
(mio.kr., 2005-priser, kædede værdier)
Kilde: Nationalregnskabet
Beregning: fMz{i} = Mz{i}/pm{i}
----------
```

Note that each variable has a section delimited by '------'. The rules are as follows:

- Every variable section must end with a line with "---" (at least three of these). The variable list as a whole must end with such a line in order to get the last variable section read.
- You may use {}-placeholders that are auto-unfolded. If you need more than one list in one section, the lists can be delimited by blanks or ';'. Example: `qJ{j}{i} j=t,e;i=a,b` (compact) or `qJ{j}{i} j = t, e   i = a, b` (more spacing).
- The index inside the {}-placeholder can be an arbitrary letter, but only one letter.
- The lists can be of arbitrary depth/dimension, for instance: `qJ{i}{j}{k}{m} i=a,b;j=1,2;k=x,y;m=nm,nk`.
- The lines following the first line (with the variable name) are descriptions. All {}-placeholders are in-substituted in the descriptions, too.
- You can use an arbitrary number of description lines. For instance, (1) description, (2) unit, (3) source, (4) calculation. You may use empty description lines.

Instead of putting the variable list in a model (.frm) file, it can instead exist in a stand-alone file called `varlist.dat` (without the `VARLIST$` line).

Gekko models: examples

If the model file name is `simple.frm`, you may load the file as follows:

```
model simple;
```

Note that extension '.frm' is automatically added if it is missing (you may alternatively use `model *;` to choose the model in Windows Explorer).

Consider the following model file (called simple.frm):

```
FRML _I      y  = c + i + g;
FRML _S      c  = (1 - Dc) * (0.4 * y + 0.4 * y[-1] + Jc) + Dc * Zc;
FRML Y       Jc = c - (0.4 * y + 0.4 * y[-1]);
FRML Y       Zc = c;
FRML T       cy = c/y;
FRML P       ye = (y2[-1] + y2 + y2[+1])/3;  //y2 can be read as y

// --------------------------------------------------------
RUNAFTER$
series  <%__simt1 - 2 %__simt2    >   y2 = y;
series  <%__simt2 + 1 %__simt2 + 1>   y2 %= 0;
predict <%__simt1 - 1 %__simt2    >   ye;

// --------------------------------------------------------
VARLIST$
----------
y
Gross domestic product
(US dollars)
Source: Eurostat

----------
c
Consumption
(US dollars)
Source: Eurostat
----------
```

The two first equations are the 'real' equations which are simultaneous and are simulated in a SIM statement. The two next Y-type equations are technical, making it possible to use `Dc = 1` to exogenize `c` via `Jc` (cf. descriptions above, and see below how to autogenerate these Y-type equations via equation codes). These equations are not part of the simulation, but are run *after* each simulated period. The T-type equation calculates a table variable that is not part of the simultaneous part of the model. Like the two Y-type equations, this equation is run *after* each simulated period (the equation calculates a simple consumption share for later tabelling).

The P-type equation calculates a variable `ye` (expected `y`) from another variable `y2`. This variable is essentially just `y`, but extended with a constant value after the simulation period. This equation is ignored by `SIM`, and can only be used together with the PREDICT statement. In the RUNAFTER$ part of the model, the `ye` equation is called via such a PREDICT statement, and the equation is calculated one year before the simulation start, because `ye` it depends on leaded `y`-values and therefore must be updated before simulation start.

Instead of the two manually constructed Y-type equations above, you may instead use this equation for `c`:

```
FRML _SJ_D  c  = 0.4 * y + 0.4 * y[-1];
```

This equation code automatically emits the two Y-type equations in the above example.

Gekko models: note

You cannot use broken lags, for instance x[-0.3]. This will perhaps be added later on (would be translated into 0.7*x + 0.3*x[-1]). Leads like x[+1] are allowed.

For exponents, please use either a^b or a**b (in addition, pow(a,b) is also possible).

Comments: use // to out-comment the rest of the line or /* ... */ to comment out several lines.

You may put meta-information into the model file (.frm). As of now, Info, Date, Freq, and Signature fields are supported. Example (to be put in the top of the model (.frm) file):

```
// Info: Model used for forecasting 2012-2030
// Date: 7-11-2012 15:37:00
// Freq: q
// Signature: fp88RzyZfJNaoTi3I4X3Ww
```

Gekko will complain if this format deviates, for instance the Info field is to be written with capital 'I', with no blank before the colon, and one blank after the colon. This rigorousness regarding form is to make it easy to spot the information in different .frm files.

The Info, Date and Freq (if q or m) fields will be displayed when loading the model (MODEL statement), and the Signature field is used for verifying that .frm files have not been changed relative to an 'official' version. The signature for a particular .frm file can be obtained with the SIGN statement. For quarterly or monthly models it can be an advantage to state the frequency explicitly, else Gekko may complain that an option freq = ... ; should be set before the MODEL statement.

If you have a variable that is defined implicity, for instance as f(x) = 0, but where the x cannot be isolated, you may use the following pattern:

```
FRML _d   x = x + f(x);
```

where f(x) is an expression from which x cannot be isolated. If you choose option solve method = newton, an equation like the above should solve just fine with SIM.

GAMS models

You may load a GAMS model in two ways: either as "raw" GAMS equations, or as "unrolled" GAMS equations.

- Raw: or instance, the raw GAMS equation `E_y .. y[i, t] =E= sum(j, x[i, j, t]);` could be translated into the corresponding Gekko statement `y[#i] = sum(#j, x[#i, #j]);`, and Gekko can use such translated "raw" equations in the DISP statement and other places.
- Unrolled: Gekko also supports the use of "unrolled" scalar GAMS euations. If in the above GAMS equation `i` runs over `a` and `b`, and `j` runs over `x` and `y`, the raw equation can be unrolled into the following (for the year 2020 here): `y[a, 2020] = x[a, x, 2020] + x[a, y, 2020];` and `y[b, 2020] = x[b, x, 2020] + x[b, y, 2020];` These equations are in a sense "atomic", and Gekko uses them for the DECOMP statement, because they are precise and unequivocal.

To load raw GAMS equations, use for instance `MODEL <gms> eqs.gms;`, where the equations are contained inside the `.gms` file. Gekko will search for `..` inside the .gms file to locate the equations.

To load unrolled equations, you need to construct a `.zip` file with suitable unrolled equations. Per default, you may use the following file names inside the `.zip`:

- `gams.gms`: Unrolled equations
- `dict.txt`: Dictionary with real variable names
- `raw.gms`: Raw GAMS equations (optional)

The unrolled equations and the dictionary can be produced by GAMS' `convert` statement. A simple example:

raw.gms

```
set t / 2000, 2001, 2002, 2003 /;
set t0[t] / 2001, 2002 /;
variable y[t], c[t], g[t];
y.l['2001']  = 499; y.l['2002']  = 531;
c.l['2001']  = 459; c.l['2002']  = 471;
g.fx['2001'] = 40;  g.fx['2002'] = 60;
y.fx['2000'] = 500; y.fx['2003'] = 540;
equation e1[t], e2[t];
e1[t] $ t0[t] ..  y[t] =E= c[t] + g[t];
e2[t] $ t0[t] ..  c[t] =E= 0.3 * y[t-1] + 0.3 * y[t] + 0.3 *
y[t+1];
model m /all/ ;
option mcp=convert;
solve m using mcp;
```

Run this file in GAMS, for instance with the Windows DOS call `/...gams_path/gams.exe raw.gms`, which produces the two files `gams.gms` and `dict.txt`. If you do not want to set `option mcp=convert;` inside your GAMS file, drop the line and instead use the call `/...gams_path/gams.exe raw.gms solver=convert`. Zip `gams.gms`, `dict.txt` and `raw.gms` into `simple.zip` (or download here if you like). Put `simple.zip` in your Gekko working folder, and try this (in Gekko):

```
model<gms>simple.zip;
y <2000 2003> = 500, 499, 531, 540;
c <2001 2002> = 459, 471;
g <2001 2002> = 40, 60;
decomp <2002 2002> y;
```

This fires up DECOMP, using the equations inside `simple.zip`. Because there are no Ref databank values, the decomp cannot show multiplier decompositions.

Inside the .zip file you may optionally put a `ModelInfo.json` file describing file names, for instance:

ModelInfo.json

```
{
   "unrolledModel": "gams.gms",      //GAMS scalar model, with
variables x1, x2, x3, ... (GAMS output)
   "unrolledNames": "dict.txt"       //Translation of x1, x2,
x3, ... into "normal" variable names (GAMS output)
   "rawModel":      "raw.gms",       //Raw model eqs (optional)
}
```

Default values are `gams.gms`, `dict.txt` and `raw.gms` as shown above, so if you are using these names inside your .zip file, you do not need a .json file. You may put other helper files in the .zip file if you wish (for instance databanks etc.). If your equations are scattered inside a system of GAMS files, you have to assemble these files into a single .gms file, but Gekko does not care about extraneous code inside such a file (it only locates the equations, searching for `..`).

For GAMS scalar models, dependent variables can be determined by looking at equation names and left-hand sides (similar to the `option model gams dep method = ...`), but as an alternative, `res_...` variables in the scalar model can be used to determine the dependent variable of each equation (Gekko will look a `res_...` variable in an equation, and if not present, will look at equation names and left-hand sides). For instance, in a GAMS equation like `x + y =e= res_y + z`, the `res_y` variable designates the dependent variable of the equation, in this case `y`. Dependent variables are used for DISP, DECOMP (including flowgraphs), and for html browser generation.

Gekko does not (at the moment) offer to solve a full GAMS model with the SIM statement, but residuals can be computed with SIM<res>. GAMS scalar models can contain variable data, inside the file `gams.gms`. If you want to load this data together with the GAMS equations, use `option gams scalar data = yes;` before the MODEL statement.

Related options

OPTION folder model = [empty];
OPTION model type = default; //default | gams
OPTION model gams dep current = no; //yes | no
OPTION model gams dep method = lhs; //lhs | eqname
OPTION model gams scalar data = no; //yes | no

Related statements

SIM, SERIES, READ

4.3.57 MULPRT

The MULPRT statement prints multipliers of variables, lists or expressions. A 'multiplier' is the difference between values in the first-position and reference databanks.

MULPRT is a specialized version of the more general PRT statement. The key difference between the two print statements is that the operators are different:

- Short operators like `d`, `p`, `m`, `q` etc. cannot be used in the MULPRT statement
- The long operators have different interpretation than in the PRT statement.
- There is the additional operator `v` for 'verbose' MULPRT.

Please note that after any MULPRT, you may click the Copy-button in the main window to copy-paste the print to Excel or other spreadsheets.

Syntax

```
mulprt < period  operators  decimals  width  ROWS  FILTER=... BANK=...
 REF=... > period  elements  HEADING=...  FILE=... ;
```

where:	
period	(Optional). Local period, for instance `2010 2020`, `2010q1 2020q4` or `%per1 %per2+1`.
operators	*operator operator ...*
operator	`lev`, `abs`, `pch`, `gdif`, `v` (note: `v` overrides any other operators)
decimals	DEC=number | NDEC=number | PDEC=number
width	WIDTH=number | NWIDTH=number | PWIDTH=number
elements	*element, element, ...*
element	`variable 'label' < operators  decimals  width >`
details:	

operators	Long operators: `lev`, `abs`, `pch` or `gdif`. These can be switched off by means of prefix `no` (for instance `nopch`), or added to existing default operators by means of underscore (for instance `_lev`). Default operators are `abs` and `pch` (absolute and percentage multiplier is printed). Finally, you may use the `v` operator to get verbose (detailed) output. Also see `option print mulprt ...`. (The so-called 'short' operators cannot be used in MULPRT).
element	The variable can be a variable name, a list (for instance `{#m}`), or an expression. Labels are put in single quotes (will be ignored for lists). Operators here will override other operators (global ones, or those set on the MULPRT statement), so element-operators are local to the particular element.
DEC=	Sets number of decimals, will apply to all kinds of numbers.
NDEC=	Sets number of decimals for non-percentage numbers. See also `option print fields ndec...`.
PDEC=	Sets number of decimals for percentage numbers. See also `option print fields pdec...`.
WIDTH=	Sets width, will apply to all kinds of numbers.
NWIDTH=	Sets width for non-percentage numbers. See also `option print fields nwidth...`.
PWIDTH=	Sets width for percentage numbers. See also `option print fields pwidth...`.
ROWS	If set, the result will be transposed, i.e., with variables running downwards.
FILTER=	A timefilter can be activated or deactivated (see TIMEFILTER statement). With `<FILTER>` or `<FILTER=yes>`, the current timefilter is used. With `<NOFILTER>` or `<FILTER=no>`, any filtering is deactivated. The filter type can also be changed locally, for instance `<FILTER=hide>` hides the out-filtered periods, whereas `<FILTER=avg>` averages the out-filtered periods. See `option timefilter...`.
BANK	(Optional). A bankname where variables are looked up. For instance `MULPRT <bank = b1> x;` is equivalent to `mulprt`

	`b1:x;`. See also `<REF = ...>`. These options can be convenient instead of opening and closing banks.
REF	(Optional). A bankname where reference variables are looked up. For instance `mulprt <bank = b1 ref = b2 m> x;` uses banks `b1` and `b2` for the multiplier. See also `<BANK = ...>`. These options can be convenient instead of opening and closing banks
HEADING=	A heading in quotes.
FILE=	A filename that the print is put into.

Note: You must specify at least one element.

For instance, `mulprt <2010 2015> x1, x1/x2, {#y};` will print out multiplier differences regarding `x1`, `x1/x2` and the variables corresponding to the list `#y`, for the period 2010-2015.

Long operators for MULPRT

lev	Absolute level: x
abs	Absolute multiplier: x-@x[-1]
pch	Relative multiplier: (x/@ -1)*100
gdif	Multiplier in growth rate: (x/x[-1] -1)*100 - (@x/@x[-1] -1) *100
v	Prints verbose output (detailed, overrides other operators)

Note: For MULPRT the so-called 'short' operators are not available (see PRT).

Per default, MULPRT always prints out corresponding to `mulprt <abs pch>`, i.e., printing out the absolute difference and percent difference. These default options can be altered in `option print mulprt ...` (see 'Related options' below). For instance, to only print the absolute multiplier of a variable, use `mulprt<abs>`, to only print the relative multiplier, use `mulprt<pch>`. You can alternatively switch options off with a preceding 'no', for instance `mulprt<nopch>` (same as `mulprt<abs>`) etc. In addition, you can use the 'glue' character '_' to add options to existing options. For instance, `mulprt<_lev>` will correspond to `mulprt<lev abs pch>`, because `lev` is added to the default options (`abs` and `pch`). You may put the codes after individual elements, for instance `mulprt var1<pch> var2<abs>;`, to have `var1` displayed as `pch` and `var2` displayed as `abs`. Codes put on an element override more general codes put directly after MULPRT, so `mulprt<pch> var1 var2<abs>;` yields the same result.

In addition to the above transformations, the print can be formatted regarding the width of each data column, and the number of decimals. Please see the PRT help file regarding the syntax and capabilities.

The output can be transposed by means of the ROWS keyword, for instance `mulprt<rows> gdp, pgdp;`. This is handy for printing a long list of timeseries, or for copy-pasting the cells to a spreadsheet by means of the copy-button in the Gekko interface.

Finally, you can use a timefilter (see TIMEFILTER) in the MULPRT option field. See examples in the PRT help file.

If referencing a variable in a particular databank beware that `mulprt b:variable;` will print the difference between `b:variable` and `ref:variable`, that is, Gekko looks for a variable with the same name in the reference databank.

Examples

After performing some experiment, you may use `mulprt fy;` to print out the multiplier regarding the variable *fY*:

```
                   fy        (E)%
  2011        806.0090       0.06
  2012        917.4147       0.06
  2013        872.7069       0.06
  2014        834.6051       0.06
```

The '(E)' indicates that `fy` is endogenous ('(X)' indicates exogenous)). Regarding width and decimals formatting, or using timefilters, see the examples in the PRT help file.

You may use `mulprt<v>` (`v` for verbose) to obtain more detailed multiplier output:

```
                   fY          %       Reference          %
Difference          %
  2011   1432173.5090       3.82   1431367.5000       3.76
806.0090       0.06
  2012   1464027.0397       2.22   1463109.6250       2.22
917.4147       0.06
  2013   1492288.9569       1.93   1491416.2500       1.93
872.7069       0.06
  2014   1513081.9801       1.39   1512247.3750       1.40
834.6051       0.06
```

The last two columns correspond to the normal MULPRT statement, whereas the four first columns show levels and growth rates from the first-position and reference databanks respectively. Note that the last percentage column is not the difference between the other percentage columns. Such a difference is the growth rate multiplier (`mulprt<gdif>` or `PRT<mp>`). So the 0.06 in the last row above is (834.6051/1512247.3750)*100, not 1.39-1.40.

If you prefer to have levels of the variable shown in a normal MULPRT statement, you can use `mulprt<_lev>`. To add levels in all MULPRT statements, use this:

```
option print mulprt lev = yes;
mulprt fy;  //this and the following MULPRT's will include levels
```

Note

If a model is loaded (see MODEL), the MULPRT statement indicates `(E)` for endogenous, and `(X)` for exogenous variables. Missing values are shown with a `M` instead of numbers. If some variable is missing in the databank, an error message will be issued.

You may change what is printed as default via the `option print mulprt ...` options (see 'Related options' below). For instance you may want to switch off printing of percentage growth permanently: `option print mulprt pch = no`.

The so-called 'short' operators (`d`, `p`, `m`, `q`, etc.) can only be used with the PRT statement.

Related options

OPTION freq a; [a|q|m|d|u]
OPTION print width = 100;

OPTION print fields ndec = 4;
OPTION print fields nwidth = 13;
OPTION print fields pdec = 2;
OPTION print fields pwidth = 8;

OPTION print mulprt lev = no;
OPTION print mulprt abs = yes;
OPTION print mulprt pch = yes;
OPTION print mulprt gdif = no;
OPTION print mulprt v = no;

OPTION timefilter type = hide;
OPTION timefilter = no;

Related statements

PLOT, SHEET, CLIP, PRT, DISP, DECOMP

4.3.58 OLS

The OLS statement performs linear regression (ordinary least squares) on an equation, optionally with linear restrictions on the parameters.

Note: a constant term (intercept) is added automatically, unless suppressed with `<constant = no>`.

In the OLS output, there are different links that can be clicked, showing for instance how the equation fits on data, decomposition with respect to the right-hand side variables, and parameter stability regarding different estimation periods.

Syntax

```
ols <period XTREND=...  XFLAT=... CONSTANT=... DUMP=... DUMPOPTIONS=...>
name  leftside = var1, var2, ...  IMPOSE=... ;
```

period	(Optional). Local period, for instance `2010 2020`, `2010q1 2020q4` or `%per1 %per2+1`.
XTREND=	(Optional). Trend polynomial of the stated degree (must be positive). When using XTREND, Gekko will estimate the trend parameters on a linear timeseries with value -1 in the start period and 0 in the end period. [New in 3.1.2]
XFLAT=	(Optional). Restrictions on the endpoints of the trend polynomial (cf. the XTREND option). This creates so-called 'Finnish' trends, and the arguments is a list containing s{i} or e{i} for start- or end-points, where {i} states the order for which the derivative must be zero. For instance `ols <xtrend=5 xflat=s2, e2>...` will use a polynomial of 5'th degree, where the second-order derivatives are zero in the start- and end-points. This means that `plot dif(ols_trend);` will show a curve that is flat at both ends (`ols_trend` is the trend part of the right-hand side). If `ols <xtrend=5 xflat=s1, e1>...` had been used instead, the curve `plot ols_trend;` would be flat at both ends. [New in 3.1.2]
CONSTANT =	With `<constant = no>`, a constant term is not added automatically.
DUMP=	(Optional). Dumps the results as a FRML equation for use in models. You may use `ols<dump>` to produce a `ols.frm` file. `ols<dump=eqs.frm>` will use the filename `eqs.frm` instead. Note that there is no firm guarantee that a subsequent MODEL statement will load the file, but in most cases it will (FRML

	statements only support a limited subset of general Gekko expressions). If the equation loads, you may consider a SIM<res> to check its residuals. Gekko will put parentheses around all expressions that contain a `+` or `-`. This will introduce superfluous parentheses in expressions like `a * (b + c)` or `exp(a - b)` etc. Note that a limited set of functions like for instance `log(y)` or `dlog(y)` on the left-hand side will produce a valid FRML equation, whereas more complicated expressions on the left-hand side will need manual resolving afterwards. [New in 3.0.6]
DUMPOPTIONS=	(Optional). If you use `ols<dump=eqs.frm dumpoptions='append'>`, the results will be appended to an existing `eqs.frm` file. These options will be augmented with styling, FRML code, etc. [New in 3.0.6]
name	(Optional). A name for the equation, used to name the results. If no name is given, `ols` is used as name.
leftside	The leftside variable (may be an expression)
var1, ...	A list of variable names or expressions. A constant term is added automatically, unless you use option `<constant = no>`.
IMPOSE	(Optional). You can impose linear restrictions on the parameters, via a suitable matrix. One restriction per row of the matrix, cf. example below. Remember to count any coefficients corresponding to a trend polynomium (XTREND).

Results:

Note that if a name is given, `ols` is replaced with that name.

ols_predict	A timeseries with the predicted values
ols_residual	A timeseries with the residuals
#ols_param	A matrix with estimated parameters
#ols_se	A matrix with standard errors on parameters
#ols_t	A matrix with t-values on parameters
#ols_covar	A matrix with the variance-covariance matrix (of parameters)

#ols_corr	A matrix with the correlation matrix (of parameters)
#ols_stats	A matrix containing different measures (analogous to the .stats matrix in AREMOS): 1: Residual sum of squares 2: Standard error 3: Residual mean 4: Root mean square error (RMSE) 5: R squared 6: R bar squared 7: [empty] 8: Dependent variable mean 9: Durbin-Watson with lag 1 (At some point, a map will be used instead for these measures).

Example

This example estimates a linear model with five parameters. You may consult the MATRIX section to see the same parameters calculated with linear algebra, or the R_RUN section to see the same parameters calculated via the R interface.

```
reset;
create lna1, pcp, bul1;
lna1 <1998 2010>  = data(' 166.223000  173.221000  179.571000
187.343000  194.888000  202.959000
  209.426000  215.134000  222.716000  230.520000  238.518000
246.654000  254.991000') ;
pcp <1998 2010>   = data(' 0.9502030   0.9699920   1.0000000
1.0235000   1.0401100   1.0605400
  1.0754700   1.0977800   1.1121200   1.1314800   1.1513000
1.1717600   1.1871600')  ;
bul1 <1998 2010>  = data(' 0.0684791   0.0591698   0.0560344
0.0535439   0.0535003   0.0631703
  0.0649875   0.0578112   0.0473207   0.0404508   0.0467488
0.0472923   0.0475191')  ;
ols <2000 2010> dlog(lna1) = dlog(pcp), dlog(pcp.1), bul1, bul1.1;
```

The statements produce the following screen output:

```
OLS estimation 2000-2010 (n = 11)
dlog(lna1)
------------------------------------------------------------------
```

```
      Variable             Estimate           Std error          T-stat
-----------------------------------------------------------------------
 dlog(pcp)                 0.144517            0.227011            0.64
 dlog(pcp.1)               0.613875            0.236473            2.60
 bul1                      0.186740            0.202534            0.92
 bul1.1                   -0.350908            0.203182            1.73
 CONSTANT                 0.0298039           0.0089418            3.33
-----------------------------------------------------------------------
R2: 0.625034    SEE: 0.00346154    DW: 1.8651
```

In addition to the screen output, the timeseries `ols_predict` and `ols_residual` are produced, together with the matrices `#ols_param`, `#ols_se`, `#ols_t`, `#ols_covar`, `#ols_corr`, and `#ols_stats`. The matrices can be printed out with the PRT statement.

In the example above, you may, for example, restrict the first two parameters to sum to 0.80, and the third and fourth to be equal like this (cf. the MATRIX statement):

```
#r = [1, 1, 0, 0, 0, 0.80; 0, 0, 1, -1, 0, 0];
ols <2000 2010> dlog(lna1) = dlog(pcp), dlog(pcp.1), bul1, bul1.1
IMPOSE = #r;
```

If the parameters are called `b{i}`, the first restriction is equivalent to `1*b1 + 1*b2 + 0*b3 + 0*b4 + 0*b5 = 0.80`, or `b1 + b2 = 0.80`. The second restriction is equivalent to `0*b1 + 0*b2 + 1*b3 + (-1)*b4 + 0*b5 = 0`, or `b3 = b4`. So the last column of the `#r` matrix contains the values that the linear restrictions should sum up to. The restrictions produce the following:

```
OLS estimation 2000-2010 (n = 11)
dlog(lna1)
-----------------------------------------------------------------------
      Variable             Estimate           Std error          T-stat
-----------------------------------------------------------------------
 dlog(pcp)                 0.167642            0.180625            0.93
 dlog(pcp.1)               0.632358            0.180625            3.50
 bul1                    -0.0863480           0.0794747            1.09
 bul1.1                  -0.0863480           0.0794747            1.09
 CONSTANT                 0.0291952           0.0085164            3.43
-----------------------------------------------------------------------
R2: 0.491156    SEE: 0.00349218    DW: 1.6847
```

Note

You may consider R to perform econometrics, cf. this page. But Gekko also has some pretty good interfaces to TSP (with its rock-solid LSQ estimator).

The variables do not need to have similar magnitude to obtain precise parameter estimates (pre-scaling is performed internally).

Instead of `ols<dump>`, some people prefer to compose FRML equations for models by hand, using TELL and PIPE. In this way, the equations can be formatted exactly as the user prefers. To control the formatting of paramaters, you may use the inbuilt `format()` function, for instance using `tell 'FRML y = {format(#ols_param[1], '0.000000')} * x + ({format(#ols_param[2], '0.000000')})';`. The last parenthesis is to deal with `#ols_param[2]` being negative. See more on formatting of strings in the TELL section.

After an OLS, you may use the Copy-button in the main Gekko window to copy/paste (with full precision) the matrix of parameter values/errrors to Excel or other spreadsheets.

OLS produces quite a lot of timeseries containing data for the clickable graphs. You may use `index ols_*;` to obtain a list of these -- for instance, `ols_trend` will contain the trend component of the right-hand side (if a trend is stated).

If there are missing values at the start or end of the data, Gekko will report this and suggest a truncated (shorter) estimation period. You may use PRT to print out the expressions if missing values are reported.

Related options

OPTION fit ols rekur dfmin = 10;

Related statements

ANALYZE, MATRIX, MODEL, R_RUN

4.3.59 OPEN

The OPEN statement opens databanks. In general, there are (potentially) the following databanks in Gekko:

Number	Searchable	Non-searchable
	Local	
1.	First	Ref
2.	Another databank	
3.	Another databank	
...	...	
n'th	Last databank	
	Global	

More info on this databank list on the databank search page. The first-position databank can always be referred to by for instance `first:x`, and the reference databank by `ref:x`.

In sim-mode, the order in the above list is not significant, since all banks that are not the first-position or local/global databank must be referred to by either bankname (like `b2:x`) or operators (like `prt<m>x;`, to print the difference between `first:x` and `ref:x`). In other modes, for instance data-mode, Gekko will search for variables in the databanks shown in the 'Searchable' column. Note here that the Ref databank is never searchable, since the purpose of the Ref databank is to ease different kinds of comparisons.

When looking for a variable `x` in a statement (for instance `prt x;` or at the right-hand side of an expression (for instance `y = x;`), Gekko will look first in the Local databank (if it contains variables), then in the first-position databank (often called Work), then in the second-position, third-position etc. databanks, and finally in the Global databank. The Local databank is used to store temporary variables, for instance variables used in functions and procedures, whereas the Global databank is used to store permanent variables that can survive a RESET or RESTART. These databanks are created at Gekko startup, but they are not shown in the databank list (F2) if they are empty. The same goes for the Ref databank, which is not shown in the F2 list if empty. Therefore, when Gekko starts up in a clean state, only the first-position databank (default: Work) is shown in the F2 list.

The OPEN statement reads timeseries from a databank file into a databank with the same name as the file (minus the extension). A databank opened in the way is called a 'named' databank (in contrast to READ which operates on the first-position and/or reference databanks). Named databanks may be written to if they are opened with OPEN<edit>, and the databanks are closed with the CLOSE statement.

> NOTE: Gekko opens up databanks in the *last* position in the databank list. This behavior deviates from AREMOS and is only relevant for data- or mixed-mode users.

To open a databank file as the first databank in the list of databanks (cf. the F2 window), you may use `open<first>` -- or `open<edit>` if you want it in first position *and* editable. The first-position databank is the bank in which timeseries are found per default, and the reference databanks is often used for multiplier purposes (analyzing differences between two banks, for instance via MULPRT, PLOT<m>, etc.).

Variables in named (OPEN) databanks are referred to by means of colon (`:`), for instance `adam:x` if the opened databank is the file `adam.gbk`, containing the variable `x`. Note that you can get the list of available in-memory databanks by means of the F2 key.

The order in which normal OPEN databanks (that is, opened with OPEN, not for instance OPEN<edit>, OPEN<first> or OPEN<ref>) are opened is only of importance if `option databank search = yes` is set (it is set to `no` as default in sim-mode, but set to `yes` as default in data-mode). So regarding a bank-less variable like `x`, it depends upon the databank search settings how `x` is searched for in the different open databanks (cf. the databank search page). The variable `@x` (same as `ref:x`) is always x taken from the reference databank, and `adam:x` is always x taken from the in-memory databank `adam`.

With `option databank search = yes`, which is default, Gekko will search for a bank-less variable `x` first in the Local databank, then all numbered databanks in the F2 window (bank number 1, 2, and so on), and finally in the Global databank.

Syntax

```
open < EDIT  FIRST  LAST  POS=...  REF  SAVE=...  CREATE format  COLS
DATEFORMAT=...  DATETYPE=... TRACE=... > filename1  AS  alias1,
filename2  AS  alias2, ...  ;
```

EDIT	The databank is opened in first position, as editable. (See also OPEN<create>). If the databank is already open, the databank will be moved to first position as editable.
FIRST	The databank is opened in first position, as non-editable (protected). If the databank is already open, the databank will be moved to first position.

SEC	The databank is opened in second position, as non-editable (protected). Short-hand for `open <pos=2>`.
LAST	The databank is opened in last position, as non-editable (protected). Note: `open <last> bank1;` is equivalent to `open bank1;`. If the databank is already open, an error will be issued.
POS=	(Optional) Indicate an integer value. The databank is opened in the n-th position, as non-editable (protected)
SAVE	With `open <save=no>`, Gekko will not write the databank to file when it is closed, even if the databank contents has changed. In that way, you may change the contents of a databank in RAM only (that is, the data is changed while the databank is open and the session lasts), but without altering the underlying databank file (.gbk). See also `close <save=no>`.
CREATE	If this option is set, Gekko will accept an `open <create> b1`, even if `b1.gbk` does not exist beforehand. Regarding a non-existing file, this is similar to `open <edit>`, the difference is that `open <edit>` puts the bank in first position, as editable. The reason why `<create>` is not the default way of opening a databank is to avoid errors if a user misspells a databank name.
TRACE	(Optional). Default is `yes`. Set `no` to omit reading of data traces. The databank will load faster and use less RAM (if traces are present). Only relevant for Gekko databanks (.gbk files).
format	(Optional). Can be `tsd`, `gbk`, `pcim`, `csv`, `prn`, `xls`, `xlsx`. The default file format is `gbk`.
filename	Filenames may contain an absolute path like `c:\projects\gekko\bank.gbk`, or a relative path `\gekko\bank.gbk`. Filenames containing blanks and special characters should be put inside quotes. Regarding reading of files, files in libraries can be referred to with colon (for instance `lib1:bank.gbk`), and "zip paths" are allowed too (for instance `c:\projects\data.zip\bank.gbk`). See more on filenames here. If the filename is set to '*', you will be asked to choose the file in Windows Explorer. The extension .gbk is automatically added, if it is missing.
COLS	(Optional). Only for .xlsx and .csv files: this indicates whether the timeseries are running downwards in columns. Note that for .csv files, you indicate this in the first 'cell' (date/name).

DATEFORMAT= DATETYPE=	(Optional). These options control the date format for .xlsx and .csv files, and for the use of SHEET in Gekcel. `DATEFORMAT` can be either `'gekko'` (default, for instance `2020q3` or `2020m11`) or a format string like `'yyyy-mm-dd'`, and the latter may contain a `first` or `last` indicator, for instance `'yyyy-mm-dd last'`, which indicates for quarterly, monthly or weekly data that the *last* day of the quarter/month/week is used. `DATETYPE` can be either `'text'` or `'excel'`. In the former case, the dates are understood as text strings (for instance `'2020q3'` or `'2020-09-30'` for a quarterly date), and in the latter case (not relevant for .csv files), the date is understood as an Excel date, which basically counts the days since January 1, 1900. This number would correspond to 44104 for the date 2020-09-31, and can be shown in Excel in different ways depending upon date format settings, language settings, etc., but the internal number itself is unambiguous. When using SHEET in Gekcel, using `DATETYPE='excel'` is advised for consistency. [New in 3.0.5].
alias	(Optional). The name that is used when referencing the databank. Default is the filename minus extension.

Example

Opening a .gbk file called `bk1` is done with

```
open bk1;
```

or by

```
open *;
```

and then selecting the databank. Afterwards, you may reference variables in the databank by means of for instance:

```
prt fy, @fy, bk1:fy;
```

This will print out the `fy` variable from the first-position, reference, and `bk1` databanks. You may use an `AS` alias to shorten the databank name:

```
open bk1 as b;
```

```
prt fy, @fy, b:fy;
```

Try pressing F2 to open up the databanks window. This window provides an overview regarding the different open databanks (including Work and Ref).

If you open up two databanks like this:

```
open bank1;
open bank2;
```

`bank1` will show up above `bank2` in the databank list, because the OPEN statement puts databank in the last position on the list. The following yields the same:

```
open bank1, bank2;
```

You may use `open<first>`, `open<pos=n>`, `open<last>` to open a databank in a specific position on the databank list (F2). The statement `open<edit>mybank;` is the same as `open<first>mybank; unlock mybank;` -- and in that case, a subsequent `xx1 = 100;` will put the series `xx1` into `mybank`:

```
open <edit> mybank;
xx1 = 100;
close mybank;
```

After the CLOSE statement, the altered databank is automatically written to file, so `mybank.gbk` will contain the `xx1` series.

Note

To get an editable databank in the second position or below, OPEN it and use UNLOCK afterwards.

When reading, extension '.gbk' is automatically added if it is missing. Global time settings does not affect the OPEN statement, so all the data in the .gbk file is read into the databank regardless of how the timeperiod is set in Gekko.

See the IMPORT statement for more information on databank formats.

You can use `series ?;` to see what kinds of series reside in all open databanks (including their frequencies).

Banks are opened as 'protected' (non-editable) as default, unless you use `open<edit>` or unlock with UNLOCK (see LOCK/UNLOCK).

You may use the isopen() function to test if a particular databank is open.

See the Gekko menu 'Options' --> 'Program dependency tracking' or use `option global dependency tracking = ...;` to activate dependency tracking, so that the use of external files (for instance program files, read/written databanks etc.) are shown as a list at the end of a Gekko session.

Related options

OPTION databank file copylocal = yes; [yes|no]
OPTION databank search = no; [yes|no]
OPTION folder bank = [empty];
OPTION folder bank1 = [empty];
OPTION folder bank2 = [empty];

Related statements

READ, WRITE, CLONE, LOCK, UNLOCK, CLEAR, CLOSE, CLOSEALL, DELETE

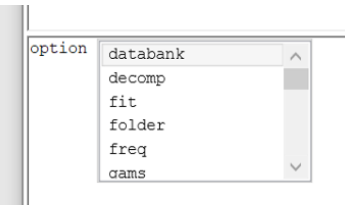
4.3.60 OPTION

Four types of options
Gekko operates with four types of options: global options, general options, block-options and local options. **Global options** are read when Gekko launches and cannot be altered later on (they have the form `option global ...`). **General options** are stated with the OPTION command and apply until changed. **Block options** are stated with the BLOCK command and apply for a block of Gekko statements. **Local options** are stated inside the `<>` local option field and apply to that single statement only. See more here.

The OPTION statement sets various *general option* values, in a hierarcical tree of possibilities. "OPTION ?" gives an overview regarding different options and their current values. Below, the different options are described (with their default values indicated).

Notes:

- Gekko versions >= 3.1.16 support the use of general options inside the local option field, for instance: `prt <2020q1 2021q2; option print freq = simple> y;` (details here).
- Note that Gekko provides suggestions (a small popup-window) for the OPTION statement (to switch suggestions off, use `option interface suggestions = none;`). The popup-window looks like the following, where you can select and click [Enter].



Syntax

option *type* = *value*;

type	One or more space-delimited options
value	• Boolean (e.g.: `yes|no`) • Integer (e.g.: `200`) • Floating point number (e.g.: `0.45`) • String (eg.: `'newton'`) • Name (eg.: `newton`) • Filename (eg.: `t:\forecast\b.gbk`)

	For strings/names/filenames, it is optional whether to use quotes (string like `'q'`) or not (name like `q`). For instance, these all set quarterly frequency:
	`%f = 'q';`
	`option freq = q;`
	`option freq = {%f};`
	`option freq = 'q';`
	`option freq = %f;`

Note: the `=` may be omitted in most cases: `option freq q;` and `option freq = q;` are equivalent. For Gekko 4.0, the `=` will probably become mandatory, so keeping the `=`'s in Gekko program file is probably wise.

Below the full list of options are provided, with their default values. You may set temporary options via the BLOCK structure.

Options

option collapse method = total; [total|avg|first|last]
This option sets the default collapse method. Applies to the COLLAPSE statement and the collapse() function.

option collapse missing d = strict; [strict|flex]
When collapsing from series of `d` (daily) frequency, Gekko will per default be strict regarding missing values (missing days will entail missing data in the lower-frequency timeseries). Set this to `flex` if you prefer flexibility (skipping the missing days). Applies to the COLLAPSE statement and the collapse() function.

option copy respect = no; [yes|no]
Regarding the COPY statement, if no period is given, the global period is used (default: `no`). [New in Gekko 3.1.15].

option databank create auto = yes; [yes|no]
If `yes`, timeseries will be auto-created. For instance, `y1 = 100;` will be possible without `create y1;`, even if `y1` does not exist in the first-position databank. Setting `option databank create auto = yes;` and `option databank search = yes;` can be practical in data revision programs. See also the MODE statement.

option databank file cache = nongbk; [all|nongbk|none]
Whether or not a databank file cache is used for faster databank (re)reading. The `none` option corresponds to Gekko < 3.1.14. Because a Gekko `.gbk` file is essentially a cache file, the speed gains of using a file cache for `.gbk` files are limited, and therefore the default option is `nongbk`. You may use option `all` to cache `.gbk` files, too (will save a bit of unzipping time). The databank file cache applices to all kinds of databank files that can be read with READ or IMPORT.

If `yes`, when reading (READ/IMPORT) or opening (OPEN) a databank, the file will first be copied into a temporary file on the user's hard

option databank file copylocal = yes;
If `yes`, when reading (READ/IMPORT) or opening (OPEN) a databank, the file will first be copied into a temporary file on the user's hard drive. This will often speed up reading/opening files on network drives.

option databank file gbk compress = yes;
option databank file gbk internal = 'databank.data';
Some options regarding the .gbk format, applying only to writing or closing databanks (WRITE/CLOSE). Without compression there are some speed gains while writing and reading, but in most cases the speed gains do not outweigh having to deal with much larger databank files. The internal name for the data file inside the zipped .gbk file is normally `databank.data`, but can be changed here. Gekko 3.1.x always writes gbk databanks in the 1.2 version of that format (but it can read both 1.0, 1.1 and 1.2 gbk databanks, that is, from all older Gekko's).

option databank search = no;
If `yes`, variables will be searched for in all open databanks except the Reference databank (cf. the F2 window). Setting `option databank create auto = yes;` and `option databank search = yes;` can be practical in data revision programs. See also the MODE statement, and the databank searching page. Note that the Local or Global databanks are always searchable, independent on MODE etc.

> NOTE: When running the same series statements again and again iteratively (for instance, iterating over rows and columns in an input-output table until convergence), you may consider switching off data tracing with `option databank trace = no;`. For such solver-like scenarios, data tracing may slow down the calculations needlessly.

option databank trace = yes; [yes|no]
If `yes` (default) timeseries changes (including array-series) are "recorded" in the form of traces. Traces are shown with DISP (with Gekko version >= 3.1.16) and are stored in Gekko databanks (.gbk), too. If, for instance, you activate tracing with `option databank trace = yes;` and issue the following statements: `x1 = 1/100;` and `x2 = x1 + 1;`, a DISP of `x1` will show the trace `x1 = 1/100`, whereas a DISP of `x2` will show the trace `x2 = x1 + 1;` and also link to the trace viewer which will also show you the sub-trace `x1 = 1/100` (even `x1` is subsequently deleted). Databank-wise, any traces are lost if a databank containing traces is READ with a Gekko version < 3.1.16, followed by a WRITE (the same goes for OPEN<edit> and CLOSE). Databank tracing is experimental, and therefore the TRACE2 command has a '2' suffix that will be removed when the functionality is mature enough. [New in 3.1.16].

option databank trace dublets = no; [yes|no]
option databank trace divide = no; [yes|no]
With `dublets = yes`, the logic to avoid traces dublets is *not* applied (and traces will behave like Gekko 3.1.16). With `divide = yes`, trace 'dividers' are shown, separating right-hand side terms. The `divide` option will be improved and is experimental. [New in 3.1.17].

option decomp flowgraph depth = 2;
The default recursive depth when showing a flowgraph window.

option decomp maxlag = 10;
option decomp maxlead = 10;
When decomposing, effects via lags are restricted within these values (larger values = slower decomp). These restrictions do not apply to GAMS scalar models, only

option decom plot zoom = 100;
Adjust zoom level (plot size) for normal plots inside a DECOMP window. Value 100 is default.

option fit ols rekur dfmin = 10;
Minimum degrees of freedom when doing recursive estimations for OLS.

option folder = yes; [yes|no]
If yes, Gekko will look for files in predefined folders (is switched on per default).

option folder bank = '';
option folder bank1 = '';
option folder bank2 = '';
While reading, importing or opening databanks (READ/IMPORT/OPEN), Gekko will first look for databanks in the working folder, and then in the `bank`, `bank1` and `bank2` folders (in that order). If the folder contains blanks, single quotes should be used (for instance `option folder bank = 'c:\my banks'`). Relative paths may be used: `option folder bank = \databanks`. In that case, Gekko will look for a sub-folder `databanks` in the working folder. If `bank` is set, databanks are written to that folder (WRITE).

option folder command = '';
option folder command1 = '';
option folder command2 = '';
While looking for Gekko program files (see RUN), Gekko will first look for .gcm files in the working folder, and then in the `command`, `command1`, `command2` folders (in that order).

option folder help = '';
Folder where Gekko looks for the help system (gekko.chm file). Normally this file is located where Gekko is installed (and comes bundled with the installer files).

option folder menu = '';
Folder where Gekko looks for menu files (.html files).

option folder model = '';
Folder where Gekko looks for model files (.frm files).

option folder pipe = '';
Folder where Gekko pipes out text output files (see the PIPE statement). If no folder is indicated, piped files will end up in the working folder.

option folder table = '';
option folder table1 = '';
option folder table2 = '';
Folders where Gekko looks for table files (.gtb files). Gekko will first look in `table`, then `table1`, and last `table2` folders.

option folder working = '';
This changes the Gekko working folder. The option survives RESET or RESTART statements. (For advanced users: note that you can also use the parameter '-folder' with the gekko.exe file, for instance: `gekko.exe -folder:c:\mygekkofiles`).

---

option freq = a; [a|q|m|w|d|u]
Sets frequency of timeseries to `a` (annual), `q` (quarterly), `m` (monthly), `w` (weekly), `d` (daily) or `u` (undated). Note that week numbering follows the ISO 8601 standard.

---

option gams exe folder = '';
This option starts out empty, and if so, GAMS will try to auto-detect the location of the executable for GAMS (gams.exe). The auto-detection seems to be pretty good at locating a 32-bit GAMS for a 32-bit Gekko, and a 64-bit GAMS for a 64-bit Gekko. Instead of this auto-detection, you may try to set the folder name manually, for instance `option gams exe folder = 'c:\GAMS\win32\24.1';`. Note that only the path is indicated, excluding the file name. Please use quotes if the folder contains dots (.).

option gams fast = yes;
Default GAMS read is using a fast reader (so-called low-level API). If this poses problems, try the more robust normal API by setting `option gams fast = no;`.

option gams time detect auto = no;
option gams time freq = a; [a, q, m, d, u]
option gams time offset = 0;
option gams time prefix = '';
option gams time set = t;
These options describe how the time dimension is obtained from the GAMS variables and parameters. The default values correspond to the time being the GAMS set name `t`, with natural annual values like 2020, 2021, 2022, etc. If the time values are instead, for instance, `t0`, `t1`, `t2`, you may use `option gams time prefix = t; option gams time offset = 2020;`. In that case, `t0` will be understood as 2020, `t1` as 2021, etc. If a variable/parameter does not use a strict set (for instance with the name `t` in this case, cf. `option gams time set`) or this set name is not recognized, Gekko may in that case represent these as for instance `x['2020']`, `x['2021']`, etc., where each of these sub-arrayseries is a timeless series (this representation is pretty much useless). To remedy this problem, either decorate the time dimension in the

gdx file with for instance the set/domain name `'t'`, or use `option gams time detect auto = yes`. With this option, for any variable/parameter where a time dimension is not recognized as a domain name, Gekko will "taste" the individual data lines of the variable/parameter, and try to identify a dimension that seems like a time dimension. For annual time values, if all values of a particular dimension are integers in the range 1500 <= t <= 3000, Gekko will presume that this is a time dimension.

option gams trim = 0;

For GDX files, with this setting set to *n*, only variables/parameters with >= *n* elements (in the GAMS sense) are imported. For instance, setting `option gams trim = 1`, variables/parameters with 0 elements are ignored (this can be set in order to clean up, or for comparison purposes (cf. COMPARE)). When counting GAMS elements, elements that have value 0 or "eps" are not counted.

Note: 'global' options can only be set in a `gekko.ini` file next to the `gekko.exe` file. Hence, their values may be set in that `gekko.ini` file and will be in place as soon as Gekko starts up. In addition, they are in a sense 'super global' in that they will keep their values even after a RESET/RESTART.

option global color = '';

This option can set a color for the lower input part of the main Gekko window (default is white). The option can be used to visually indicate 'special' Gekko versions, for instance with particular fencing (cf. `option global fence`). See possible color names here. (.NET colors), or you may alternatively use a hex color code like for instance `'#FFDFD991'` (remember the `'#'`). Note: the option can only be set in a `gekko.ini` file next to the `gekko.exe` file.

option global dependency tracking = none; [none|simple]

When set to `simple`, Gekko tracks the use of external files (for instance program files, read/written databanks etc.). A list of such files is shown at the end of a Gekko session. The tracking begins when something is run in Gekko (a command, a block of commands, a program file, etc.), and the Gekko 'traffic light' in the lower right changes from green to yellow. The tracking ends when the run ends, and the traffic light changes back to green (or red, if there is a run-time error). If there is a run-time error (for instance because of a missing file), this missing file will be shown in the dependency list, so a proper dependency list presupposes an error-free Gekko run. Note: the option can only be set in a `gekko.ini` file next to the `gekko.exe` file (but you may set the option manually via the Gekko menu 'Options' --> 'Program dependency tracking'). The option covers +these file operations.

The following operations are covered: READ/IMPORT data file, WRITE/EXPORT data file, read listfile, write listfile, MODEL loading (considered a read operation), PIPE statement (considered a write operation), RUN statement (considered a read operation), and SYS calls (considered a write operation: will only be checked against a a possible *black*list, and can be switched off with `option global fence sys = no;`).

option global fence black folders = '';
option global fence black folders read = '';
option global fence black folders write = '';
option global fence white folders = '';
option global fence white folders read = '';
option global fence white folders write = '';
option global fence sys = yes; [yes|no]
With these options it is possible to restrict (blacklist/whitelist) access to files. You indicate the folders (or parts of folders) as one string, delimited by semicolons (`;`). You can distinguish between reading and writing operations, and note that putting something into for instance `option global fence black folders` is equivalent to putting the same thing into both `option global fence black folders read` and `option global fence black folders write` (think of `option global fence black folders` as a convenience option to set the corresponding read/write options at the same time -- the same applies to whitelists). As an example, you may use the blacklist `option global fence black folders = 'g:\data\old; g:\programs\old';` to make sure that files in the two folders `g:\data\old` and `g:\programs\old` are not accessed (neither read from nor written to, including subfolders). These folders could contain old data and old programs. Conversely, a whitelist could be used, if it is known that only data from `g:\data\new` and programs from `g:\programs\new` are to be accessed (either read from or written to, including subfolders). In that case, the whitelist would look like this: `option global fence white folders = 'g:\data\new; g:\programs\new';`. Blacklists and whitelists may be combined, but beware that blacklisted folders always override whitelisted folders. For instance, you may set `option global fence white folders = 'g:\data\new;'; option global fence black folders = 'g:\data\new\testing;';`. In that case, only files from the `g:\data\new` folder may be used, *except for* the files in `g:\data\new\testing`. Note: the options can only be set in a `gekko.ini` file next to the `gekko.exe` file. Note also that there are special rules regarding SYS calls, see the following link. The fencing options cover +these file operations.

The following operations are covered: READ/IMPORT data file, WRITE/EXPORT data file, read listfile, write listfile, MODEL loading (considered a read operation), PIPE statement (considered a write operation), RUN statement (considered a read operation), and SYS calls (considered a write operation: will only be checked against a a possible *black*list, and can be switched off with `option global fence sys = no;`).

option global warnings ignore = '';
option global warnings limit = 5; [-2, -1, 0, positive]
option global warnings pauseat = '';
option global warnings print = yes; [yes|no]
These options control how warnings are printed. Per default warnings are printed while running the program, and at the end of the job -- if there are warnings -- a warning count (summary) is shown, where it is possible to click a "show warnings" link to get a list of the warnings. In that list, you may choose to see warning id numbers, and if you prefer to switch off some of these warnings, you may use for instance `option global warnings ignore = 'w2.1, w3.2';` after which these

warning types are no longer shown (the id numbers themselves have no special significance). You may limit the warning messages in different ways. Use `option global warnings limit = -2;` to pause (popup box) at each warning message, use `option global warnings limit = -1;` to indicate limit = infinite (show all warnings), use `option global warnings limit = 0;` to show no warning messages, and use `option global warnings limit = 5;` (for instance) to limit similar warning messages to at most 5. This limit avoids that Gekko prints a lot of similar warnings messages on screen, for instance while detecting issues in a data file etc. (printing on screen slows down Gekko and clutters the output window). Lastly, you may use `option global warnings print = no;` to have no warnings printed while running the job (the summary at the end of the job is still shown).

---

option interface alias = no; [yes|no]
When this option is set, Gekko will look for at list with the name `#alias` to perform alias substitution of names. For instance, `global:#alias = (('a', 'x'), ('b', 'y[z]'));` will mean that `prt a, b;` is interpreted as `prt x, y[z];`. So the `#alias` list is a list of lists, where the inner lists contain the source name and the destination name. Among other things, `#alias` can be convenient as a bridge between the naming conventions of two different models. You can switch on and off the use of alias with this option, but if you need to use another `#alias` list, you have to first perform a RESET/RESTART.

option interface clipboard decimalseparator = period; [period|comma]
The option is used in three places: (1) in the CLIP statement, (2) regarding the 'Copy'-button on the user interface (copying cells from the last print to the clipboard), and (3) regarding copy-pasting cells from the DECOMP window. The setting is not relevant for SHEET which does not use the clipboard.

option interface csv decimalseparator = period; [period|comma]
Used with IMPORT<csv> and EXPORT<csv>, where the databank is a csv (comma separated) file.

option interface csv delimiter = semicolon; [semicolon|comma|tab]
Used with IMPORT<csv> and EXPORT<csv>. The field delimiter is `;` per default, but can be set to `,` or [tab-symbol] instead. This symbol delimits the columns of data. Regarding EXPORT<csv>, combining `option interface csv delimiter = comma;` with `option interface csv decimalseparator = comma;` will issue a warning. [New in 3.0.5].

option interface csv ignoremissing = no;
When `yes`, all missing values in an exported (EXPORT<csv>) csv file are represented as blank cells. The default is `no`, which makes for instance subsequent summing of Excel cells less prone to missings-bugs. [New in 3.1.16].

option interface csv ndec = 100;
Number of decimals when writing levels in a .csv file. A value > 20 is interpreted as "full machine precision". [New in 3.1.2].

option interface debug = dialog; [dialog|none]

Choose between `dialog` or `none`. If `dialog`, the user may rerun a file on syntax errors, or skip lines on runtime errors.

option interface edit style = gekko; [gekko | gekko2 | rstudio | rstudio2]
The default style is `gekko`. With `gekko2`, an `[Enter]` will execute the line AND move the cursor to the next line. Therefore, with `gekko2`, multiple lines can be executed with `[Enter]` `[Enter]` ... `[Enter]` instead of `[Enter]` `[downarrow]` `[Enter]` `[downarrow]` ... `[Enter]`. With `gekko` and `gekko2`, `[CTRL+Enter]` issues a new line without executing anything (like a newline in Notepad). The `rstudio` style emulates RStudio, where `[Enter]` works like a newline in Notepad, and `[Ctrl+Enter]` executes the line. With `rstudio` style, an issued statement moves the cursor to the next line (similar to `gekko2`). With `rstudio2` style, `[Ctrl+Enter]` does not move to the next line (similar to `gekko`).

option interface errors = normal; [old|normal]
Sets how error messages are shown. Choose `old` for error messages corresponding to Gekko version < 3.1.14. The `normal` (that is: new) error messages handle program structure a lot better.

option interface excel ignoremissing = no;
When `yes`, all missing values in an exported (EXPORT<xlsx>) Excel file are represented as blank cells. The default is `no`, which makes for instance subsequent summing of Excel cells less prone to missings-bugs. [New in 3.1.16].

option interface excel language = danish; [danish|english]
If set to `danish`, CLIP, SHEET and the 'Copy' button will use the Excel code `"ikke.tilgængelig()"` instead of `"na()"` to indicate missing values. Also, if set to `danish`, EXPORT<csv> will use `'#N/A'` for missing values (instead of `'#I/T'`), provided that `option interface csv ignoremissing = no`.

option interface excel modernlook = yes; [yes|no]
Turns on modern looking blue colors in SHEET.

option interface help copylocal = yes;
If this option is active, Gekko will copy the file gekko.chm into a local folder on the user's hard disk, before it is opened (for instance with F1, or the HELP statement). In Windows 7 and 8, problems will arise when trying to open up a .chm files from a network drive, so keeping the option active should eliminate such problems.

option interface mode = data; [sim|data|mixed]
Sets some interface messages/warnings etc. regarding different modes (see the MODE statement). This option is not intended for direct use: please use the MODE statement that changes the mode explicitly.

option interface mute = no; [yes|no]
When this option is set to `yes`, all screen output (or pipe file output) is suppressed. If errors occur, this option is automatically set to `no`. Alternatively, PIPE can also be used to suppress screen output and direct it to a file instead. The option is is primarily set in program files/procedures/functions to avoid voluminous screen output blocking and slowing the user interface. [New in 3.0.6].

option interface prn decimalseparator = period; [period|comma]
Used with IMPORT<prn> and EXPORT<prn>, where the databank is a prn file. [New in 3.1.16].

option interface prn delimiter = blank; [blank|semicolon|comma|tab]
Used with IMPORT<prn> and EXPORT<prn>. The field delimiter is blank per default, but can be changed. Regarding EXPORT<prn>, combining `option interface prn delimiter = comma;` with `option interface prn decimalseparator = comma;` will issue a warning. [New in 3.1.16].

option interface prn ndec = 100;
Number of decimals when writing levels in a .prn file. A value > 20 is interpreted as "full machine precision". [New in 3.1.16].

option interface remote = no; [yes|no]
option interface remote file = '';
If the first option is set to 'yes', Gekko will look for a file named `remote.gcm` (the default file name) in the current working folder. If that file is changed (Gekko looks at Windows time stamps), the whole file is run (corresponding to `RUN remote.gcm;`). This makes remote controlling of a Gekko instance possible from, for instance, an external text editor. For instance, typing Alt+Enter or something similar in the editor might be set up so that a `remote.gcm` file containing the contents of the text line (or block of lines) is created. See also a Gekko-specific add-in for the Sublime text editor here.

option interface sound = no; [yes|no]
option interface sound type = bowl; [bowl|ding|notify|ring]
option interface sound wait = 60;
If active, Gekko will play a sound when a program file finishes (or has an error). The sound type can be `bowl, ding, notify, ring`. The last option is the minimum duration of the job (in seconds), before for Gekko plays a sound.

option interface suggestions = option; [none|option]
If set to `option`, small popup with suggestions will appear when entering options.

option interface table operators = yes;
If `yes`, html tables will include radio buttons to select operators by point-and-click, for instance percentage growth, or multiplier differences. Note: 'operators' was called 'printcodes' in Gekko 2.4 and earlier.

option interface zoom = 100;
The zoom level (default = 100%) regarding font sizes in the user interface (the three tabs in the main window). May be increased on high-res monitors, or for educational purposes (projector).

---

option interpolate method = avg; [total|avg]
This option sets the default interpolate method. Applies to the INTERPOLATE statement and the interpolate() function.

option interpolate olsette trend = yes; [yes|no]

Whether or not the 'Olsette' Denton-like method for INTERPOLATE includes a trend in the OLS estimation used.

---

option library cache = yes; [yes|no]
Whether or not a library cache is used.

---

option menu startfile = menu.html;
The filename for the html menu file shown in the 'Menu' tab (see MENU).

---

option model cache = yes; [yes|no]
Whether or not a model file cache is used for faster model (re)loading.

option model cache max = 20;
The maximum number of compiled models kept in RAM (cached), during a Gekko session. Reduce the number if you for instance are doing a lot of ENDO/EXO simulations and run into memory issues.

option model gams dep current = no; [yes|no]
When reading a GAMS raw model with `model<gms>`, Gekko will try to identify the dependent variable in each GAMS equation. If `option model gams dep method = lhs` (default), Gekko will try to find the first variable on the left-hand side that is not inside a []-bracket or a $-condition. WIth `current = no`, Gekko will identify the variable as dependent, even if it contains a lag or lead. With `current = yes`, Gekko will only look for variables that have no lags and no leads. See also the `model <dep = ...>` local option under MODEL. [New in 3.0.2].

option model gams dep method = both; [lhs|eqname|both]
When reading a GAMS raw model with `model<gms>`, Gekko will try to identify the dependent variable in each GAMS equation. With option `lhs`, Gekko will try to find the first variable on the left-hand side that is not inside a []-bracket or a $-condition. With option `eqname`, Gekko uses the equation name instead, for instance the equationname `e_pc_tot` is understood as identifying the variable `pc` as dependent variable. In both these cases, such identification can be overruled with a list identifying these relationships, cf. the `model <dep = ...>` local option under MODEL. With option `both`, the two methods are combined. [New in 3.0.3].

option model gams scalar data = no; [yes|no]
If `yes`, a GAMS scalar model will be loaded with the model data too, besides the equations (the data is stored in the `gams.gms` file) [New in 3.1.20].

option model infofile = yes; [yes|no]
If `no`, Gekko will not produce a modelname__info.zip file after a MODEL statement (with default model type).

option model type = default; [default|gams]

If `type = gams`, Gekko will handle ENDO, EXO, UNFIX and DISP differently to ease the interaction with GAMS. When you load a Gekko model with MODEL, Gekko automatically sets `option model type = default;`, and when you load a GAMS model with MODEL<gms>, Gekko automatically sets `option model type = gams;`.

---

option plot all dash order = 1; [0 to 6]
For `plot<a>`, where series from the first-position and reference databank are shown alongside each other, this setting determines the dash type (1=solid, 2=dashes or 3=dots). The possibilities are the following: order 0 = no dashes | order 1 = = 1,3,2 | order 2 = 1,2,3 | order 3 = 3,1,2 | order 4 = 2,1,3 | order 5 = 3,2,1 | order 6 = 2,3,1. Default is order = 1, where the first-position series is shown as a solid line, and the reference series is shown as a dotted line. The last possibility in each list of three integers (for instance the 2 in the list 1,3,2) is for a future possibility of also showing a series from the second-position databank.

option plot decimalseparator = period; [comma|period]
The type of decimal separator used for tic labels.

option plot elements max = 200;
Limits the number of curves in a graph, so that PLOT does not crash or stall when accidently feeding with too many elements. See PLOT<nomax>.

option plot lines points = yes; [yes|no]
Whether or not the PLOT graph has individual points indicated with markers. This can also be controlled for each variable via `<type=lines>` vs. `<type=linespoints>`.

option plot palette = gekko; [gekko|excel|dream]
The default palette regarding line colors (can also be set in the .gpt file). Use `excel` for Microsoft Excel colors. Use `dream` for a palette suitable for color-blind users.

option plot using = '';
With this option, a global .gpt PLOT template file can be set, for instance `option plot using = m:\common\gekko.gpt;`. Subsequent PLOT statements will then use that template (unless the PLOT statement itself has a `using` argument).

option plot xlabels annual = at; [at|between]
option plot xlabels nonannual = between; [at|between]
Whether or not the data points are at x-tics, or between them. For quarterly and monthly data, the latter is often the most logical.

option plot xlabels digits = 4; [2|4]
Number of digits in the year (at the x-labels). The option only applies to labels between tics, not labels at tics (see the preceeding options). With two digits, we get 15, 16, 17, instead of 2015, 2016, 2017.

option plot zoom dpiscale = 0;
This option is only intended for use if the automatic dpi (dots per inch) scale detection does not work properly (value 0 means that the automatic detection is used). For instance, set the value to 150, if the dpi scaling should correspond to 150%. The larger the value, the larger the fonts, etc. will be on the plot. To see what

dpi scale Gekko auto-detects, try menu Help --> About... in the main Gekko window. In Windows, the dpi scale can be set under System --> Display --> Scale ("Change the size of text, apps, and other items").

---

option print collapse = none; [avg|total|none]
If this option is set to `avg` or `total`, Gekko will print averages or totals for timeseries of frequencies quarterly or monthly. Cf. also the COLLAPSE statement. Only applies to `option print freq = pretty;`.

option print disp maxlines = 3;
The number of lines of data shown per default in the DISP statement. You may choose -1 for infinite, 0 means that no data are shown.

option print disp model gams scalar = modern; [standard|modern]
Use `modern` for a DISP design that utilizes information from a GAMS scalar model, making the information more similar to what can be seen in DECOMP and a generated html equation browser. With `modern` (and a scalar model), the variables going in and out of equations are known without ambiguities. With `standard`, only the raw GAMS model file is used.

option print elements max = 400;
Limits the number of elements in a print, so that PRT does not crash or stall when accidently feeding with too many elements. See PRT<nomax>.

option print fields ndec = 4;
option print fields nwidth = 13;
option print fields pdec = 2;
option print fields pwidth = 8;
These options set the default decimals and width of number fields when printing with PRT or MULPRT. The first two sets decimals and width for for non-percent fields, and the two last for percent fields. The `ndec` and `nwidth` settings also affect printing of matrices.

option print freq = pretty; [pretty|simple]
If this option is set to `pretty`, timeseries of frequencies are printed with separation, blank lines, etc., for better readability. If the option is set to `simple`, the dates are shown without such separation.

option print index date = 'start'; ['start'|'middle'|'end', or a date enclosed in quotes]
When option `<i>` is used in e.g. PRT or PLOT, without designating a period like for instance `<i=2020>`, Gekko will use the start of the time period to index the timeseries. For such 'naked' `<i>` cases, you may set `option print index date` to `'start'`, `'middle'` or `'end'`, or a specific date enclosed in single quotes, for instance `'2020'` or `'2020q1'`. The option applies to both PRT, PLOT and SHEET.

option print index value = 100;

Sets the index value used with option `<i>`, when indexing a series in e.g. PRT or PLOT. Per default the value is 100, but could be altered to for example 1 instead. The option applies to both PRT, PLOT and SHEET.

option print mulprt abs = yes; [yes|no]
option print mulprt gdif = no; [yes|no]
option print mulprt lev = no; [yes|no]
option print mulprt pch = yes; [yes|no]
option print mulprt v = no; [yes|no]
These options set the default way of printing with the MULPRT statement. Per default, `abs` and `pch` are chosen, so MULPRT shows absolute multiplier difference (`abs`), and relative difference (`pch`). These can be overridden via the option fields in the MULPRT statement.

option print prt abs = yes; [yes|no]
option print prt dif = no; [yes|no]
option print prt gdif = no; [yes|no]
option print prt pch = yes; [yes|no]
These options set the default way of printing with the PRT statement. Per default, `abs` and `pch` are chosen, so PRT shows the absolute level (`abs`), and the growth rate (`pch`). These can be overridden via the option fields in the PRT statement.

option print split = no; [yes|no]
If set, the variables or expressions delimited by comma are shown separately, cf. also the local option PRT<split>. With this general option set, `prt x, y;` is shown as if it had been `prt x; prt y;`. This may be practical for comparisons of data with similar columns, for instance `prt x[#i], @x[#i];`. In that case, you may prefer to use for instance the `<missing = m>` option, so that all columns (`#i`) are shown (and are hence aligned), regardless of whether the sub-series exist or not.

option print width = 100;
Line width (number of characters) when printing to screen

---

option python exe folder = ''; [New in 3.1.8]
This option starts out empty, and if so, Gekko will try to auto-detect the location of the executable folder for Python (`python.exe`). If this auto-detection fails, you may try to set the folder name manually, for instance `option python exe folder = 'c:\Users\Thomas\AppData\Local\Programs\Python\Python36_64\python.exe';`. If this name does not end with .exe, .bat or .cmd, Gekko will append `python.exe` or `\python.exe`. So in the above example, `option python exe folder = 'c:\Users\Thomas\AppData\Local\Programs\Python\Python36_64';` would have been equivalent. If you use a .bat or .cmd file, it could contain a line like `"c:\Users\Thomas\AppData\Local\Programs\Python\Python36_64\python.exe" %1 %2 %3` to preserve the arguments.

---

option r exe folder = '';

This option starts out empty, and if so, Gekko will try to auto-detect the location of the executable folder for R (`R.exe`, or to be precise `Rscript.exe`). If this auto-detection fails, you may try to set the folder name manually, for instance `option r exe folder = 'c:\Program Files\R\R-3.0.0\bin\R.exe';`. Note the single quotes because of the blank in 'Program Files'. If this name does not end with .exe, .bat or .cmd, Gekko will append `R.exe` or `\R.exe`. So in the above example, `option r exe folder = 'c:\Program Files\R\R-3.0.0\bin';` would have been equivalent. If you use a .bat or .cmd file, it could contain a line like `"c:\Program Files\R\R-3.0.0\bin\R.exe" %1 %2 %3` to preserve the arguments.

---

option series array calc missing = error; [error|m|zero]
With option `error` (default), a missing `#i` element in `x` in an expression like for instance `sum(#i, x[#i])` will halt with an error. With option `m`, Gekko will not halt, but will use missing values instead of `x[#i]` -- so the sum will return a missing value. With `zero`, zeroes are used instead of missing elements in `x[#i]`, so the sum is calculated as if the missing elements were skipped.

option series array print missing = error; [error|m|zero|skip]
With option `error` (default), a missing `#i` element in `x` in `prt x[#i];` will halt with an error. With option `m`, missing values will be printed (typically as 'N' instead of 'M' to indicate that it is a non-existing sub-series). With `zero`, zeroes are printed instead, and with `skip`, the missing `#i`'s are skipped (not shown). See also the local option `<missing = ...>` for PRT. As mentioned, with option `missing = m`, the raw `prt x[#i];` will show missing `#i` elements as 'N' because they are non-existing, whereas an expression like `prt x[#i] + 1;` will show them as 'M', since they are now a result of a mathematical expression.

option series data missing = m; [m|zero]
With option `m` (default), missing data/observations in a normal series or array-series are propagated normally, so that an expression containing a missing value will always result in a missing value. With option `zero`, a missing value in an observation will be treated as if the value was 0. There are other options that deal with the question of what to do if a normal series or array-subseries does not exist at all, so this option solely affects the time domain: what to do if an observation inside a series is missing. In some cases, for instance when importing data from GAMS, such a missing observation could sensibly be interpreted as the value 0. See more on this, including examples, on this page, and cf. also the `<missing = ignore>` local option for SERIES and PRT/PLOT/SHEET. [New in 3.0.3].

option series dyn = no; [yes|no]
With this option, lagged endogenous variables like in the statement `x = x[-1] + 1;` accumulate over time, because the statement is run 'dynamically', one period at a time consecutively. The option will only be active, if the right-hand side of the statement is of series type. Because the use of this option entails a speed penalty, the option can only be activated via `BLOCK; ... ; end;` (see BLOCK).

option series dyn check = yes; [yes|no]
With this option, statements like `x = x[-1] + 1;` are checked regarding the existence of lagged endogenous variables on the right-hand side, and is such

variables are present, the statement must be decorated with a `<dyn>`, `<dyn = no>`, or be inside of a suitable BLOCK that sets the same option (`block series dyn = yes|no`). If this is not the case, versions 3.1.7 and higher of Gekko will fail with an error. This is to guard against simply forgetting to put a `<dyn>` or `block series dyn = yes` on an accumulating statement. [New in 3.1.7].

option series failsafe = no; [yes|no]
When this option is set, Gekko will abort with an error message, when a series statement like for instance `y = x;` or `y[2020] = %v;` tries to insert an observation containing a missing value into the left-hand series. The option can be practical for debugging Gekko program files (.gcm), if the source of a missing value is hard to track. The option is only intended for debugging purposes. See also `option solve failsafe`. [New in 3.0.2].

option series normal calc missing = error; [error|m|zero]
With option `error` (default), a missing series `x` in a calculation will will halt with an error. With option `m`, missing values will be used instead of `x`, and with `zero`, zeroes are used instead of the missing `x`.

option series normal print missing = error; [error|m|zero|skip]
With option `error` (default), a missing series `x` in `prt x;` will halt with an error. With option `m`, missing values will be printed (typically as 'N' instead of 'M' to indicate that it is a non-existing series). With `zero`, zeroes are printed instead, and with `skip` the missing `x` series is skipped (not shown). See also the local option `<missing = ...>` for PRT. As mentioned, with option `missing = m`, the raw `prt x;` will show N's because the series is non-existing, whereas an expression like `prt x + 1;` will show the missings as 'M', since they are now a result of a mathematical expression.

---

option sheet collapse = none; [avg|total|none]
If this option is set to `avg` or `total`, Gekko will show averages or totals for timeseries of frequencies quarterly or monthly. Cf. also the COLLAPSE statement. Only applies to `option sheet freq = pretty;`.

option sheet engine = internal; [internal|excel]
For reading and writing Excel spreadsheet files, Gekko will normally use an internal engine to do this. This engine is independent upon Excel being installed on the user's pc. The internal engine only supports .xlsx files, not the older .xls files. If you need to access the older .xls files, you may set the option to `excel` which reads and writes both the old and new format. Beware however, that the `excel` option demands Excel being installed on the pc. It is also a bit slow and unstable, and may leave Excel processes behind, eating up memory.

option sheet freq = simple; [pretty|simple]
If this option is set to `pretty`, timeseries of frequencies are printed with separation, blank lines, etc., for better readability. If the option is set to `simple`, the dates are shown without such separation.

option sheet mulprt abs = yes; [yes|no]
option sheet mulprt gdif = no; [yes|no]

option sheet mulprt lev = no; [yes|no]
option sheet mulprt pch = no; [yes|no]
option sheet mulprt v = no; [yes|no]
These options set the default way of printing with the SHEET<mul> (not implemented yet) statement. Per default, only 'abs' is chosen, so SHEET<mul> shows absolute multiplier difference ('abs'). These can be overridden via the option fields in the SHEET statement.

option sheet prt abs = yes; [yes|no]
option sheet prt dif = no; [yes|no]
option sheet prt gdif = no; [yes|no]
option sheet prt pch = no; [yes|no]
These options set the default way of printing with the SHEET statement. Per default, `abs` is chosen, so SHEET shows the absolute level (`abs`). These can be overridden via the option fields in the SHEET statement.

option sheet cols = no; [yes|no]
option sheet rows = yes; [yes|no]
Per defaut, the SHEET statement prints timeseries row-wise, that is, with variable names in the first column, and time periods in the first row. These options may change the orientation, or you can use SHEET<rows> or SHEET<cols>.

---

option smooth method = linear; [linear|geometric|repeat|spline|overlay]
This option sets the default smoothing method. Applies to the SMOOTH statement and the smooth() function.

---

option solve data create auto = yes; [yes|no]
If yes, when a databank is read by means of the general READ statement (that is, excluding READ<first> or READ<ref>, but including READ<merge>), all model variables not present in the file are auto-created (and filled with missing values). See also the MODE statement.

option solve data ignoremissing = no; [yes|no]
If `yes`, if a variable has a missing value when Gekko tries to simulate (SIM), the number zero will be used instead. Warning: this may get the model to simulate, but the result may be incorrect!

option solve data init = yes; [yes|no]
If `yes`, when simulating lagged values are used as starting values for endogenous variables (this is default and typically the most robust). If no, the current period values are used as starting values for endogenous variables.

option solve data init growth = yes; [yes|no]
option solve data init growth min = -0.02;
option solve data init growth max = 0.06;
If set `yes`, Gekko will look at the historical growth rate of endogenous variables, in order to come up with good initial values. With the sub-options `min` and `max`, you may indicate a range. Per default, the range is set to -2% up to 6%, meaning that only

historical growth rates within that range will be used to initialize endogenous variables. The 'growth' option typically speeds up SIM convergence, provided resonable min/max limits are used.

option solve failsafe = no; [yes|no]
If `yes`, the Gauss algorithm will stop at the exact time when any equation produces a missing value. The option slows the simulations down a bit (which is why it is not set per default). When the option is on, variables from the above-mentioned problematic equation will be printed out on screen automatically (using DISP<info>). See also `option solve newton robust`. and `option series failsafe`.

option solve forward dump = no; [yes|no]
If `yes`, information regarding the Fair-Taylor iterations is kept. With one lead-variable, you can see the iterations by means of printing the variables `ftabs1_1`, `ftabs1_2`, `ftabs1_3`, etc., where the last number is the FT-iteration (try `plot {'ftabs1_*'};`). If Newton-Fair-Taylor ('nfair') is used, there will also be matrices `#ft_1`, `#ft_2`, etc., containing the Jacobi interaction between leaded variables.

option solve forward fair conv = conv1; [conv1|conv2]
Set Fair-Taylor convergence type to `conv1` (default) or `conv2`. This corresponds to the criteria used in the `option solve gauss conv ...` options.

option solve forward fair conv1 abs = 0.001;
option solve forward fair conv1 rel = 0.001;
Relative criterion for the (default) `conv1`-type convergence check, and absolute criterion for the (default) `conv1`-type convergence check. Note that the criteria are less strict than the corresponding Gauss criteria.

option solve forward fair conv2 tabs = 1.0;
option solve forward fair conv2 trel = 0.001;
Relative criterion for the PCIM-like `conv2`-type convergence check, and absolute criterion for the PCIM-like `conv2`-type convergence check. Note that the criteria are less strict than the corresponding Gauss criteria.

option solve forward fair damp = 0.0;
Damping used regarding leaded endogenous variables in the Fair-Taylor algorithm. The larger the factor is, the harder the damping. Setting the factor to 0.0 means no damping at all, whereas setting it to 1.0 means the equations cannot progress.

option solve forward fair itermax = 200;
option solve forward fair itermin = 0;
The maximum and minimum number of iterations done in the Fair-Taylor algorithm.

option solve forward nfair conv = conv1; [conv1|conv2]
Set Newton-Fair-Taylor convergence type to `cnv1` (default) or `conv2`'. This corresponds to the criteria used in the `option solve gauss conv ...` options.

option solve forward method = fair; [fair|nfair|none]
The method used regarding leaded endogenous variables. If set to `fair`, the Fair-Taylor algorithm is used, and if set to `nair`, the Newton-Fair-Taylor method us used.

If set to `none`, no forward-looking method is used. In that case, the leaded endogenous variables are just taken exogenously as their databank values.

option solve forward nfair conv1 abs = 0.001;
option solve forward nfair conv1 rel = 0.001;
Relative criterion for the (default) `conv1`-type convergence check, and absolute criterion for the (default) `conv1`-type convergence check. Note that the criteria are less strict than the corresponding Gauss criteria.

option solve forward nfair conv2 tabs = 1.0;
option solve forward nfair conv2 trel = 0.001;
Relative criterion for the PCIM-like `conv2`-type convergence check, and absolute criterion for the PCIM-like `conv2`-type convergence check. Note that the criteria are less strict than the corresponding Gauss criteria.

option solve forward nfair damp = 0.0;
Damping used regarding leaded endogenous variables in the Newton-Fair-Taylor algorithm. The smaller the factor is, the harder the damping. Setting the factor to 1.0 means no damping at all, whereas setting it to 0.0 means the equations cannot progress.

option solve forward nfair itermax = 200;
option solve forward nfair itermin = 0;
The maximum and minimum number of iterations done in the Newton-Fair-Taylor algorithm. Usually this algorithm need far fewer iterations than standard Fair-Taylor.

option solve forward nfair updatefreq = 100;
How often the Jacobi matrix is updated in the Newton-Fair-Taylor algorithm. For hard problems, you may set `updatefreq = 1`.

option solve forward terminal = const; [const|growth|exo]
This sets terminal conditions regarding leaded endogenous variables. If the simulation period ends in 2100, `y[+1]` in that period will be set to the value `y` is solved for in 2100 (and not the databank value for `y` in 2101). With `growth`, `y[+1]` in 2100 will use the growth rate instead of the level for the solved `y` in 2100. If the option is set to `none`, `y[+1]` in 2100 will be taken as the databank value of `y` in 2101 (this is often not a good choice).

option solve forward terminal feed = internal; [internal|external]
This is a technical option that decides wheather the terminal values are used inside one Fair-Taylor iteration, or only between them.

option solve gauss conv = conv1;
Set Gauss convergence type to `conv1` (default) or `conv2`. Read more about convergence criteria in the help file related to the statement ITERSHOW.

option solve gauss conv1 abs = 0.0001;
option solve gauss conv1 rel = 0.0001;
Relative criterion for the (default) `conv1`-type convergence check, and absolute criterion for the (default) `conv1`-type convergence check.

option solve gauss conv2 tabs = 1.0;
option solve gauss conv2 trel = 0.0001;
Relative criterion for the PCIM-like `conv2`-type convergence check, and absolute criterion for the PCIM-like `conv2`-type convergence check.

option solve gauss conv ignorevars = yes; [yes|no]
If active, variables indicated by means of the CHECKOFF statement will be ignored regarding convergence check.

option solve gauss damp = 0.5;
In the Gauss algorithm, this damps any equations with damping set with the given factor. The larger the factor is, the harder the damping. Setting the factor to 0.0 means no damping at all, whereas setting it to 1.0 means the equations cannot progress. The damped equations have a 'Z' in their formula codes.

option solve gauss dump = no; [yes|no]
This option activates recording of Gauss iterations, for later inspection by means of the ITERSHOW statement. Beware that the option may use a lot of RAM, and that it slows down simulations considerably.

option solve gauss itermax = 200;
option solve gauss itermin = 10;
The maximum and minimum number of iterations done in the Gauss algorithm.

option solve gauss reorder = no; [yes|no]
When active, this option reorders equations in the simultaneous block of the model. This should normally reduce the number of required Gauss iterations for solving the model, but it may sometimes induce starting value problems (therefore the option is set to `no` as default). The option should be issued before a MODEL statement to take effect.

option solve method = gauss; [gauss|newton]
Choose between `gauss` or `newton`

option solve newton backtrack = yes; [yes|no]
If yes, the Newton algorithm tries to backtrack if an iteration step seems too large.

option solve newton conv abs = 0.0001;
Sets the absolute Newton convergence criterion. This is the value that the square root of the sum of squared residuals in all the equations may not exceed. So any residual will be (numerically) lower than this value, and usually a lot lower (since there are many equations).

option solve newton invert = lu; [lu|iter]
The algorithm used to invert the Jacobian matrix. The `lu` option is usually the most robust.

option solve newton itermax = 200;
The maximum number of iterations done in the Newton algorithm, before exiting.

option solve newton robust = yes; [yes|no]

With this option set, the Newton method will be more robust regarding starting values that normally would result in a crash due to illegal values, such as, for instance, the logarithm of a negative number etc. The robust mode is experimental, but if the starting values are legal, `robust = yes` will perform exactly the same iterations as `robust = no`. See also `option solve failsafe`.

option solve newton updatefreq = 15;
This options indicates how often the Newton algorithm should update the Jacobi matrix, when doing fast steps.

option solve print details = no; [yes|no]
If yes, the Newton algorithm will produce quite a lot of extra information regarding the iterations.

option solve print iter = no; [yes|no]
If `yes`, the individual periods are printed out while simulating. Set to `no` as default.

option solve static = no; [yes|no]
If `yes`, all lagged endogenous values are taken as their databank values (i.e. not their simulated values). See also SIM<static> local option.

---

option splice obs = 0; [0 or positive integer]
An integer value `n` limits the length of the overlapping periods in SPLICE (`n` = 0 is interpreted as infinite: that is, no effect). For instance, `option splice obs = 4;` would mean that Gekko uses at most 4 overlapping observations when splicing. Think of it as removing observations/data in the non-primary series until the overlap has the desired max size. This can also be set as a local option, for instance `splice <obs=4> ... ;`

---

option string interpolate format val = ''; [string]
When insterting values inside a strings, Gekko can format the values in different ways. Use `'0.000'` for 3 decimals, `'12:0.000'` for 3 decimals inside a 12 chars wide field, `'12:F3'` for the same, `'-12:0.000'` left-aligned. Instead of `0`'s, `#`'s can be used, too. See also the STRING help page, and the format() in-built function.

---

option system code split = 20; [0 or positive integer]
This is a very technical option related to how Gekko compiles Gekko program files. If > 0, long non-looping program files are internally chopped up in smaller chunks which are put into their own C#-methods. This eases the life of the C# compiler, especially for large/long files. The option tells how many lines of .gcm code are bundled into sub-methods at a time. A value of 20 seems good regarding speed, but the special value 0 switches this splitting off altogether. So you may try the value 0, if Gekko breaks down for mysterious reasons.

option system clone = yes; [yes|no]

This is a very technical option related to how user-defined functions and procedures behave. If set active (default), when calling a function like `%y = f(#x);`, Gekko will clone the function arguments, so that there can be no side-effects on the function arguments after the function call is done. The option may cost some performance/speed if a function is called with very large objects, like very large matrices etc. Setting the option = no only applies to the types series, list, map and matrix (scalars never have side-effects).

option system read encoding = auto; [ansi|utf8|auto]
Determines the character encoding when reading text files. With `auto` (default), Gekko will 'taste' the file to see whether it is ANSI or UTF-8, and try to read it in the right encoding. You can use `ansi` to force ANSI encoding, or `utf8` to force UTF-8 encoding. ANSI is the old Windows-1252 standard, which is nowadays mostly replaced by UTF-8 encoding. Beware that `auto` (but not the other options) also changes Unicode "no-break space" into normal space, and removes any Unicode "soft hyphen" (these Unicodes may appear when copy-pasting from html pages, and Gekko will choke on them).

option system write encoding = ansi; [ansi|utf8]
Determines the character encoding when writing text files. You can use `ansi` (default) to force ANSI encoding, or `utf8` to force UTF-8 encoding. ANSI is the old Windows-1252 standard, which is nowadays mostly replaced with UTF-8 encoding. In the longer run, the default setting for Gekko will become `utf8`.

option system write utf8 bom = no; [yes|no]
When `option system write encoding = utf8` has been chosen, the `bom` option indicates whether a byte order mark should be included or not, when writing a text file. Byte order marks are in any case superfluous for UTF-8 (UTF-8 files have no byte order), but they are often added to indicate that the file is an UTF8-file. You may use `option system write utf8 bom = no` to omit byte order marks when UTF-8 files are written.

option system threads = 5;
Max number of threads used for reading/writing from cache.

---

option table decimalseparator = period; [comma|period]
The type of decimal separator used.

option table html datawidth = 5.5;
For html tables, this is the minimum width of data columns, in so-called 'em' units (in CSS: "width: 5.5em"). The 'em' units are independent of font size.

option table html firstcolwidth = 5.5;
For html tables, this is the minimum width of the first column, in so-called 'em' units (in CSS: "width: 5.5em"). The 'em' units are independent of font size.

option table html font = Arial;
option table html fontsize = 72;
You may choose the font and fontsize for html tables. The fontsize is in percent, so 72 corresponds to 72% (in CSS: "font-size: 72%").

option table html secondcolwidth = 5.5;
For html tables, this is the minimum width of the second column (sometimes containing variable names), in so-called 'em' units (in CSS: "width: 5.5em"). The 'em' units are independent of font size.

option table html specialminus = no; [yes|no]
If `yes`, a non-breaking hyphen is insert instead of the normal minus character. This may avoid some breaking of numbers, but that hyphen is not good for copy-pasting to Excel via right-clicking the table. (But please use the copy-button in the user interface to copy a table to Excel).

option table ignoremissingvars = yes; [yes|no]
If `yes`, missing variables will be shown as 'M', just like missing values for existing variables.

option table mdateformat = ''; [string in quotes]
This option will change for instance "2020m1" to "Jan. 2020" regarding monthly table dates. Options are: 'english-short', 'english-long', 'danish-short', 'danish-long'. These can have a '-lower' appended, for instance 'english-short-lower' (lower first letter of the month). The option must be stated within single quotes.

option table stamp = yes; [yes|no]
If `yes`, a date and time stamp is added to tables.

option table thousandsseparator = no; [yes|no]
Can activate thousandsseparator (either period or comma, depending upon `option table decimalseparator`). Note that you can now use negative decimals places. For instance, using varformat="f9.-2" in the gtb file, numbers are rounded to nearest hundreds. Combining these two, a number like 12345 would be printed as "12.300" or "12,300" depending on the decimalseparator.

option table type = html; [txt|html]
If set to `txt`, tables will be shown in text format in the Main tab. If set to `html`, .html format will be used (shown in the Menu tab). There are some options to control the html layout: see `option table html ...`.

---

option timefilter = no; [yes|no]
Switches the timerfilter on or off (provided a timerfilter has been defined). See TIMEFILTER.

option timefilter type = hide; [hide|avg]
If set to `hide`, applying a timefilter will just hide the out-filtered periods (i.e., they are not shown). If `avg` is used instead, the non-shown periods will be aggregated into the shown periods (as averages). See TIMEFILTER.

Note

As stated above, you may omit the '=' in option statements, but it may be a good idea to keep it in programs for readability.

Setting `option  interface suggestions = option` (which is default) will help the user navigate the option tree.

For setting temporary options, see the BLOCK structure.

4.3.61 PAUSE

PAUSE is for pausing the statement flow: the statement will wait for the user to press [Enter]. See also ACCEPT.

Syntax

pause info;

info	(Optional). Text string to be displayed when a statement flow is paused. You can use '\n' inside the string to insert a new line.

Example

The statement may contain text inside single quotes:

```
pause 'This is the first part of the scenario';
```

Related statements

RETURN, STOP, EXIT, ACCEPT

4.3.62 PIPE

Redirects output to a file (or back to the screen). Directing to a .html file is supported.

If you are going to pipe only parts of your Gekko output to an external file, it is recommended to start at pipe with `PIPE [filename];`, and then use `pipe<pause>;` and `pipe<continue>;` to start and stop piping to the file (and `pipe<stop>;` at the end of the session). If you instead use `pipe<stop>;` and `pipe [filename];` to switch piping to the same file off and on, the external file may become blocked (and the program will run slower, too).

If you just want to suppress screen output, without directing it to an external file, you may use `option interface mute = yes;` (if PIPE is used, the file writing is suppressed).

Syntax

`pipe < HTML APPEND PAUSE CONTINUE STOP > filename ;`

HTML	(Optional). If this option is used, Gekko will insert the text into the `<body>` of an existing html file (if appending), or create a html file from scratch to write into. Note that Gekko appends text to the .html file without any prior formatting. So for instance to append a line to a .html file, you have to include the html tags (for instance: `tell '<p>Hello</p>';`). Html output is reasonably simple to convert into pdf.
APPEND	(Optional). If this option is used, the output will be appended to an existing file, otherwise the file is overwritten (if it exists to begin with).
PAUSE	(Optional). Sets piping on pause
CONTINUE	(Optional). Starts piping again
STOP	(Optional). Stops piping altogether.
filename	(Optional, not used with PAUSE, CONTINUE, STOP options). Filenames may contain an absolute path like `c:\projects\gekko\bank.gbk`, or a relative path

	`\gekko\bank.gbk`. Filenames containing blanks and special characters should be put inside quotes. Regarding reading of files, files in libraries can be referred to with colon (for instance `lib1:bank.gbk`), and "zip paths" are allowed too (for instance `c:\projects\data.zip\bank.gbk`). See more on filenames here.

Any pre-existing file with the same name will be overwritten, unless option APPEND is used. If PIPE<stop> is used, the output is re-directed to the screen.

Example

You may want to keep a print of some variables in an output file:

```
read adambank;
pipe ex.out;
prt fy ul;
pipe <stop>;
```

The file `ex.out` now contains the variable print. If you instead use `pipe <append> ex.out;`, the existing `ex.out` file will be appended to.

The following illustrates piping to html:

```
pipe <html> out.html;
tell '<p>This is the first line</p>';
tell '<p>This is the second line</p>';
pipe <stop>;
```

Afterwards, you may open `out.html` in a web browser to see what it looks like. If you need to pause and continue piping, you may use this:

```
pipe text.txt;
tell 'a';
pipe<pause>;
tell 'b';
pipe<continue>;
tell 'c';
pipe <stop>;
```

After this, the file `text.txt` only contains 'a' and 'c', but not 'b'. If you put TELL statements between PIPE<continue> and PIPE<pause> statements, you will make sure that only the TELL's end up in the file.

Note

When piping, any error messages are also piped to the file. You may consult the 'traffic lights' in the lower right corner of the Gekko window, in order to see if an error occurred and the program aborted (in that case, the light will be red - a running program is shown as yellow).

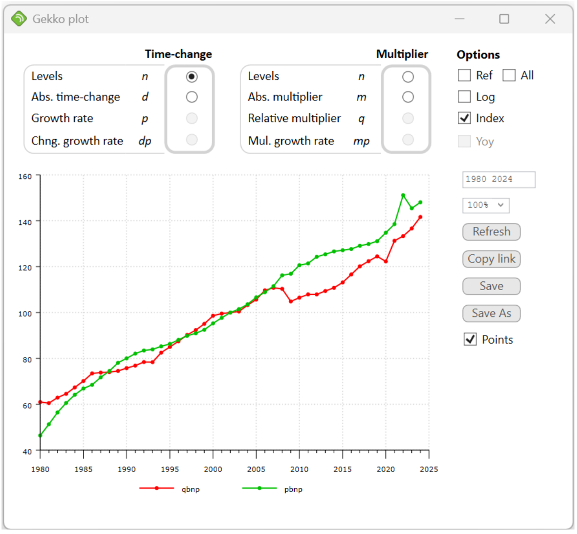
Related options

OPTION folder pipe = [empty];
OPTION interface mute = no;

4.3.63 PLOT

You can create plots of timeseries variables with the PLOT statement. There are many possibilities: for layout options, see the demo plots. Layout options can be stated in the PLOT statement itself, or stored in a .gpt schema file (for reuse of styling). Note that plotting of mixed frequencies is supported.

Guide: plotting
For an easier introductory guide on Gekko plotting, see this page.



Gekko plot

Time-change

Levels	*n*	●
Abs. time-change	*d*	○
Growth rate	*p*	○
Chng. growth rate	*dp*	○

Multiplier

Levels	*n*	○
Abs. multiplier	*m*	○
Relative multiplier	*q*	○
Mul. growth rate	*mp*	○

Options

□ Ref □ All
□ Log
☑ Index
□ Yoy

1980 2024
100%

☑ Points

The PLOT arguments have the same syntax as PRT/MULPRT, SHEET and CLIP. PLOT uses the free open-source gnuplot 5.1 internally, and the options of the PLOT statement and corresponding .gpt file are named to match gnuplot naming conventions. PLOT can create a .svg, .emf, .png or .pdf file silently, if the FILE= option is used. The .emf files are practical for MS Office applications, including Word, but .svg is more modern and looks better on Office, too -- however drag&drop, copy/paste etc. with .svg files does not work as seamlessly for .svg files as .emf files. The .svg format is practical for html pages, and should usually be preferred over .png for such pages. Gnuplot supports many output file formats (so-called 'terminals'), so more formats may be added if needed.

You may use <bank=... ref=...> to locally change the databanks used, instead of using OPEN and CLOSE.

Note: You may use the in-built XML Notepad editor to edit the .gpt files, cf. the XEDIT statement. You can use a global .gpt file via "OPTION plot using = ... ;".

Note that Gekko 3.1.x supports plotting (and printing) series with mixed frequencies.

PLOT uses the same internal component as PRT, so regarding operators and other details, also see the PRT help page.

In the PLOT window, you may choose operators in a layout similar to the DECOMP window. Besides these radio buttons, you may choose 'Ref' for series in the Reference databank, 'All' to show at the same time series from the first-position and Reference databanks, 'Log' to log-transform the input data, 'Index' to index transform the input data (setting index = 100 in the start of the period), 'YoY' to show year-over-year changes for quarterly or monthly data. In addition, you may choose the time period, the zoom level of the graph, and there are buttons to refresh, copy and save the graph (save as .svg, .emf, .png or .pdf file). Finally, there is a checkbox to select/unselect point markers on the graph lines (the small circles).

> Gekko uses gnuplot 5.1 internally for plotting. The gnuplot engine may crash if fed with illegal syntax or nonsensical data. Unfortunately, the gnuplot error messages seem hard to extract into Gekko, but you may use PLOT<dump> and feed the gnuplot.gp file into gnuplot 5.1 to inspect the error messages.

Syntax

```
plot <period  operator   PLOTCODE=...  DUMP NOMAX mainOptions  BANK=...
REF=...  MISSING=...> elements   USING=...  FILE=...  ;

  elements: element1, element2, ...

  element: expression  label  <elementOptions>
```

- If no period is given inside the <...> angle brackets, the global period is used (cf. TIME).
- If a variable without databank indication is not found in the first-position databank, Gekko will look for it in other open databanks if databank search is active (cf. MODE).

The more general options shown above are the following (cf. the 'Main options' table below for all the plot-related options):

period	(Optional). Local period, for instance `2010 2020`, `2010q1 2020q4` or `%per1 %per2+1`.
operator	(Optional). 'Long': `abs`, `dif`, `pch`, `gdif`, or 'short': `n`, `d`, `p`, `dp`, `m`, `q`, `mp`, `l`, `dl`, `r`, `rd`, `rp`, `rdp`, `rl`, `rdl`, `a`, `ad`, `an`, `ap`, `adp`, `al`, `adl`, `i`. Read more about operators under PRT.
variables	Name of the variable(s) printed. Several variables can be printed at once using var1, var2, var3, ... You may also use lists or expressions. Labels may be provided, like `plot x 'Production';` , or `plot x x.fromseries('label');` to use label metadata.

PLOTCODE =	(Optional). The contents of this string must be gnuplot code (for instance 'set' statements separated by semicolon), and the contents is sent to gnuplot and inserted just before the gnuplot 'plot' statement. If plotcode is also stated in the .gpt schema file, both plotcodes are transferred to gnuplot (first the .gpt plotcode, and then the local option plotcode).
DUMP	(Optional). With option `<dump>`, Gekko will put two gnuplot files in the working folder: `gnuplot.gp` (the gnuplot script), and `gnuplot.data` (the gnuplot data). The gnuplot script can be run inside gnuplot 5.1 with the following statement: `load 'gnuplot.gp'` (note the quotes).
NOMAX	(Optional). Do not restrict the number of curves, cf. `option plot elements max`.
BANK	(Optional). A bankname where variables are looked up. For instance `plot <bank = b1> x;` is equivalent to `PLOTb1:x;`. See also `<REF = ...>`. These options can be convenient instead of opening and closing banks.
REF	(Optional). A bankname where reference variables are looked up. For instance `plot <bank = b1 ref = b2 m> x;` uses banks `b1` and `b2` for the multiplier. See also `<BANK = ...>`. These options can be convenient instead of opening and closing banks
MISSING=	(Optional). With `<missing = ignore>`, PLOT will deal with missing array sub-series and missing data values like GAMS, treating them as zero for sums and mathematical expressions, or skipping the printing of a sub-series if it does not exist. The following options are set locally and reverted afterwards: `option series array print missing = skip; option series array calc missing = zero; option series data missing = zero`. See also the appendix page on missings.
USING=	(Optional). Indicates a .gpt file (xml template) to style your plot. You may use '*' as filename to select from .gpt files in the working folder. If no extension is provided, .gpt is added automatically. See also `option plot using = ...` . Filenames may contain an absolute path like `c:\projects\gekko\bank.gbk`, or a relative path `\gekko\bank.gbk`. Filenames containing blanks and special characters should be put inside quotes. Regarding reading of files, files in libraries can be referred to with colon (for instance `lib1:bank.gbk`), and "zip paths" are allowed too (for instance `c:\projects\data.zip\bank.gbk`). See more on filenames here.

FILE=	(Optional). Use extension `emf` (default), `png`, `svg` or `pdf`. The resulting file is in .emf format as default. Such a file can be imported into many Windows programs such as Word and others. You may use a filename with explicit extension .png/.svg/.pdf instead, and PLOT will produce the file in that format. The .svg files are very useful for insertion into html document. Filenames may contain an absolute path like `c:\projects\gekko\bank.gbk`, or a relative path `\gekko\bank.gbk`. Filenames containing blanks and special characters should be put inside quotes. Regarding reading of files, files in libraries can be referred to with colon (for instance `lib1:bank.gbk`), and "zip paths" are allowed too (for instance `c:\projects\data.zip\bank.gbk`). See more on filenames here.

You may use an operator to indicate which kind of data transformation you would like on your variables, for instance `plot<d>`, `plot<q>`, `plot<pch>`. As in the PRT statement, you may also use element-specific operators (for instance `plot unemp, gdp<p>;`). See the PRT statement regarding the use of operators.

In addition to the above options, you may put graph-options inside either the main option field (`plot <...>`), or inside the element option fields (`plot x1<...>, x2<...>`). These options can alternatively be stored in an external xml-based .gpt file, for instance `plot x1, x2 using=p;` will use the file `p.gpt` to style the graph. The structure of the .gpt file corresponds to the distinction between PLOT main options and element options. Cf. the example section below.

Main options

Located inside the `plot<...>` option field, or in .gpt files directly inside the `<gekkoplot>` tag. The first column of the table is before the '=', and the second column of the table is after, for instance `plot <type = linespoints>;`. Some of the right-hand side expressions may require quotes around them, for instance `plot<font = 'Verdana'>;`, not `plot<font = Verdana>;`. If in doubt, try using quotes.

type	The most used of the following are the line-types `linespoints` and `lines`, together with `boxes` (column chart). • `linespoints`, `lines`: Normal lines, with or without point markers. • `boxes`: bar chart/histogram. If several timeseries are boxes, these will be clustered, unless the stacked option is set. • `filledcurves`: lines where the area below each line is an area. If the stacked option is set, the areas are stacked instead of overlayed. • `steps`: Step-wise lines, a bit box-like. • `points`: Just the point markers, no lines. • `dots`: Just tiny dots.

	• `impulses`: vertical lines instead of boxes. Quotes may be omitted.
dashtype	'1':normal, '2':dashed, '3':dotted, ... (default: '1'). More. Quotes should be used.
linewidth	A number. Default: 3.
linecolor	You may use named colors (for instance 'red') or color codes known from html (for instance '#0000FF'). Default: color is taken from the palette setting. Quotes should be used.
pointtype	An integer. These points are shown for each period, if the linetype is linespoints or points. • 1:'+' • 2:'x' • 3:'*' • 4:box • ... etc. Default: 7 (circle). More.
pointsize	A number. The size of the points. Default: 0.5.
fillstyle	String. Only relevant for linetype `boxes`. You may use many combinations, for instance 'empty', 'solid', 'solid 0.5' (50% transparent), 'solid border', 'pattern 0', 'pattern 1', etc. To provide a black border around the boxes, use for instance 'solid 1.0 border linetype -1'. Default: 'solid'. More. Quotes should be used.
title	The title of the entire plot. Quotes should be used.
subtitle	A subtitle underneath the title. Quotes should be used.
font	Set for instance 'Verdana', 'Arial', 'Times', 'Courier New' ... . Default is 'Verdana'. Quotes should be used.
fontsize	Default is 12.
bold	Use this option to indicate bold font type for different elements. Choose from 'title', 'ytitle', 'xtics', 'ytics', 'key', or indicate several

	of these separated with commas, for instance 'title, ytitle, key'. Quotes should be used.
italic	Use this option to indicate italic font type for different elements. Choose from 'title', 'ytitle', 'xtics', 'ytics', 'key', or indicate several of these separated with commas, for instance 'title, ytitle, key'. Quotes should be used.
tics	You may choose between 'in' or 'out'. Default: 'out'. Quotes should be used. Regarding number formatting, see `option plot decimalseparator = ...`.
grid	Choose between yes|no|xline|yline. If `yes`, both vertical and horizontal lines are shown. If `xline`, only vertical lines are shown. If `yline`, only horizontal lines are shown. Default: `yes`. Quotes can be omitted.
gridstyle	This sets how gridlines are formatted, stated in gnuplot syntax. Quotes should be used. • Default is the following, which are light grey dashed lines: `'linecolor rgb "#d3d3d3" dashtype 3 linewidth 1.5'`. • The gridstyle `'linecolor black dashtype 2 linewidth 2.0'` will provide black dashed lines which look ok in Word. • To emulate the solid grey style of Gekko 2.2.4 and earlier, use `'linecolor rgb "#f0f0f0" linetype 1 linewidth 1'`. In general, the same dashed lines can look quite different in different "environments". So there may be differences in the Gekko window versus inside Word versus pdf (exported from Word) versus html (via svg) versus pdf (exported from html) versus printed from Word or a browser. In general, there will be small differences between the .emf, .svg, .png and .pdf files. For html pages, please use .svg instead of .png for better quality.
key	In gnuplot, the legend is called 'key'. This sets how the legend is to be displayed, stated in gnuplot syntax. Default is the following, which is outside of the plot area, at the bottom center: `'out horiz bot center Left reverse'`. Quotes should be used. To remove the key completely, you can use gnuplot-code `'set key off'`: `plot<plotcode='set key off'>;`.
palette	You may use a comma-separated list of named colors (for instance: "red, blue, green", or rgb color codes (like "#0000FF, #FFFF00, #00FFFF"). Default is this: "red, web-green, web-blue,

	orange, dark-blue, magenta, brown4, dark-violet, grey50, black" (these are gnuplot internal color names). More. Quotes should be used.
stack	If the element is set active (`<stack>`), `boxes` are stacked instead of clustered, and `filledcurves` are stacked instead of overlayed. Default: `no`.
boxwidth	The width of the boxes. Set to 1 for max width. Default: 0.75.
boxgap	The gap between clusters of boxes (only relevant if you are using two or more boxes at the same time). Default: 2, that is, a gap of what two boxes would fill.
separate	With the `separate` option, lines and boxes are separated, so that all lines (non-boxes) are shown at the top of the plot (with labels on the left y axis), whereas all boxes are shown at the bottom of the plot (with labels on the right y2 axis). For instance, this is practical for residual plots, so that the residuals do not interfere with observed/fitted lines. The option will override any `y2` options regarding the lines. This functionality is Gekko-specific and does not yet work for stacked boxes (option `boxstack`), where the scaling will not be precise (this will be fixed in a future version). Default: `no`.
xline	Vertical line at the given period. Several lines may be given. For instance: `<xline>2020q2</xline>`.
xlinebefore	Vertical line between the given period and the period before. Several lines may be given. For instance: `<xlinebefore>2020q2</xlinebefore>`.
xlineafter	Vertical line between the given period and the period after. Several lines may be given. `<xlineafter>2020q2</xlineafter>`
xzeroaxis x2zeroaxis	The `xzeroaxis` is the horiontal axis corresponding to y=0, and the `x2zeroaxis` is the axis corresponding to y2=0 (the right-hand side y-axis). These `xzeroaxes` will only be shown if the y or y2 values change sign. Default for `xzeroaxis` is `yes`, and default for `x2zeroaxis` is `no`.
ymirror	Mirror the left y axis on the right side. You may choose between '0':none, '1':only tics, '2':tics with labels, and '3': tics with labels + axis label. If the y2 axis is used, the `ymirror` setting is ignored.

ytitle y2title	A title for the y or y2 axis. Quotes should be used. If the title should break, you may use a '\n', for instance `plot<ytitle='Balance\nof payments'>;`.
yline y2line	Horizontal line at the given y- or y2-value. Several lines may be given. For instance: `<yline>150</yline>`. The lines are not shown if the lines are outside of the y values of the variables shown in the graph (you may combined with `ymaxsoft`/`yminsoft`). You may alternatively do horizontal lines manually, for instance: `plot 150 '' <type=lines dashtype='3' linecolor='gray'>, x1, x2, x3;`. The quotes `''` after `150` indicated that the label is skipped. Putting the line first means that it will appear in the background (looks better for `type=boxes`).
ymax y2max	Fixed max for the y or y2 values. Will overrule any maxhard or maxsoft values. Can cut off data points.
ymaxhard y2maxhard	All values > maxhard are filtered out, but all values < maxhard are shown. This setting is practical for filtering out outliers. Think of 'hard' as being capable of cutting off data points.
ymaxsoft y2maxsoft	All values are shown, but the axis will not scale down below ymaxsoft. This keeps a sensible scale, even if the y or y2 values are very small. Think of 'soft' as being incapable of cutting off any data points.
ymin y2min	Fixed min for the y or y2 values. Will overrule any minhard or minsoft values. Can cut off data points.
yminhard y2minhard	All values < minhard are filtered out, but all values > minhard are shown.This setting is practical for filtering out outliers. Think of 'hard' as being capable of cutting off data points.
yminsoft y2minsoft	All values are shown, but the axis will not scale up above yminsoft. This keeps a sensible scale, even if the y or y2 values are very small. Think of 'soft' as being incapable of cutting off any data points.
label	A free-floating label is inserted at the given position. Several labels may be given. Quotes should be used. [Not available yet]
plotcode	The contents of this `<plotcode>` tag must be gnuplot code (for instance 'set' statements separated by semicolon), and the contents is sent to gnuplot and inserted just before the gnuplot

	'plot' statement. If plotcode is also stated as a local option to the PLOT statement, both plotcodes are transferred to gnuplot (first the .gpt plotcode, and then the local option plotcode).
arrow	An arrow is inserted between the given positions. Several arrows may be given. [Not available yet]

Element options

Located in the element options, for instance `plot x1<...>, x2<...>`, or in .gpt files inside the `<lines>` tag (which contains `<line>` tags).

type	See under the main options.
dashtype	See under the main options.
linewidth	See under the main options.
linecolor	See under the main options.
pointtype	See under the main options.
pointsize	See under the main options.
fillstyle	See under the main options.
y2	Set y2 to indicate the you want the series shown at the y2 axis (right-hand y axis).

Example using PLOT options versus gpt file

For instance, you may produce a graph with dashed lines using this:

```
plot <type=lines linecolor='blue'> rfy<dashtype='1'>,
rfcp<dashtype='2'>, rfm<dashtype='3'>, rfe<dashtype='4'>;
```

Here, in the main option field, the linetype is stated (`type=lines`), including the linecolor (`color='blue'`). These can also be stated individually in the elements options, if needed. In the element options, four dashtypes are given, for instance `dashtype = '1'`.

Instead of the above PLOT statement, you may use:

```
plot rfy, rfcp, rfm, rfe using=p;
```

together with the following .gpt file (xml):

```
<gekkoplot>
  <type>lines</type>
  <linecolor>blue</linecolor>
  <lines>
    <line>
      <dashtype>1</dashtype>
    </line>
    <line>
      <dashtype>2</dashtype>
    </line>
    <line>
      <dashtype>3</dashtype>
    </line>
    <line>
      <dashtype>4</dashtype>
    </line>
  </lines>
</gekkoplot>
```

As you can see, the structure in the first PLOT statement corresponds to the structure of the .gpt file. You may also combine PLOT options and gpt files: in that case, the PLOT options will override the gpt options. So for instance, `plot rfy, rfcp, rfm<color='red'>, rfe using=p;` would make the third line red instead of blue.

Other examples

The statement

```
plot <p> x1, x2;
```

plots percentage growth of the timeseries `x1` and `x2` from the first-position databank.

```
for %i = fY, fX, fE;
  plot {%i} file=graph_{%i};
end;
```

This creates three graphs that are put into three different .emf files.

You may 'piggyback' gnuplot code along with the PLOT statement, for instance:

```
plot <plotcode = 'set xtics rotate by 90'> fy, fe, fcp;
```

This rotates the x-tic labels. If you need to state several gnuplot statements, you can separate them with ';'.

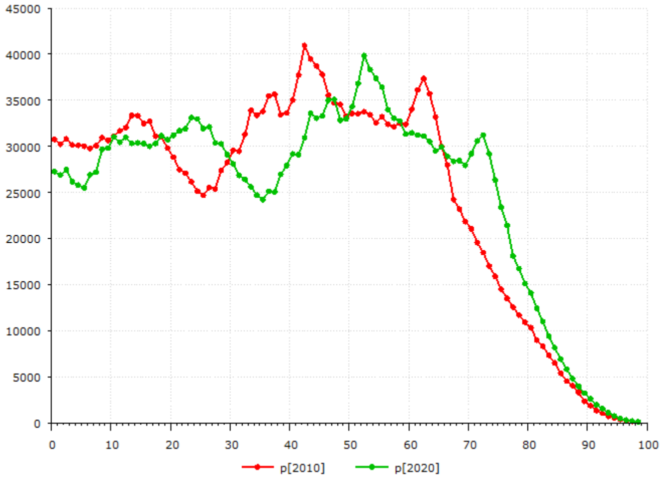
Age profiles

A special kind of plot shows the population of part of the population in age groups. If the annual array-series `p!a` is defined for different age-groups, for instance `p[30]` for 30-year olds, this array-series can be transformed via the `rotate()` function into another array-series `p!u` defined over the special 'undated' (`u`) frequency (now defined for different years like `p!u[2020]`).

```
p = series(1);  //definition of p!a
time 2010 2010;
p[1]=30763;p[2]=30245;p[3]=30815;p[4]=30148;p[5]=30128;p[6]
=30025;p[7]=29782;p[8]=30081;p[9]=30949;p[10]=30662;
p[11]=31110;p[12]=31687;p[13]=32047;p[14]=33354;p[15]=33328;p[16]
=32485;p[17]=32721;p[18]=31099;
p[19]=31014;p[20]=29813;p[21]=28828;p[22]=27482;p[23]=27095;p[24]
=26161;p[25]=25148;p[26]=24692;
p[27]=25552;p[28]=25396;p[29]=27388;p[30]=28247;p[31]=29575;p[32]
=29460;p[33]=31299;p[34]=33925;
p[35]=33371;p[36]=33795;p[37]=35484;p[38]=35658;p[39]=33441;p[40]
=33635;p[41]=35034;p[42]=37751;
p[43]=40948;p[44]=39479;p[45]=38715;p[46]=37808;p[47]=35552;p[48]
=34724;p[49]=34556;p[50]=33315;
p[51]=33557;p[52]=33536;p[53]=33749;p[54]=33433;p[55]=32546;p[56]
=33232;p[57]=32415;p[58]=32114;
p[59]=32524;p[60]=32391;p[61]=34024;p[62]=36128;p[63]=37357;p[64]
=35715;p[65]=33192;p[66]=29981;
p[67]=27998;p[68]=24250;p[69]=23210;p[70]=21832;p[71]=21054;p[72]
=19568;p[73]=18497;p[74]=17021;
p[75]=15906;p[76]=14517;p[77]=13512;p[78]=12562;p[79]=11708;p[80]
=10933;p[81]=10325;p[82]=8973;
p[83]=8333;p[84]=7341;p[85]=6515;p[86]=5397;p[87]=4546;p[88]
=4041;p[89]=3310;p[90]=2351;p[91]=1872;
p[92]=1333;p[93]=1050;p[94]=720;p[95]=571;p[96]=367;p[97]=254;p[98]
=160;p[99]=117;
time 2020 2020;
p[1]=27264;p[2]=26908;p[3]=27469;p[4]=26166;p[5]=25795;p[6]
=25508;p[7]=26932;p[8]=27216;p[9]=29669;
p[10]=29818;p[11]=31033;p[12]=30449;p[13]=30993;p[14]=30335;p[15]
=30378;p[16]=30302;p[17]=30028;
p[18]=30318;p[19]=31164;p[20]=30731;p[21]=31201;p[22]=31704;p[23]
=31914;p[24]=33129;p[25]=32977;
p[26]=31932;p[27]=32133;p[28]=30374;p[29]=30287;p[30]=29122;p[31]
=28110;p[32]=26856;p[33]=26424;
p[34]=25606;p[35]=24717;p[36]=24215;p[37]=25122;p[38]=25045;p[39]
=26969;p[40]=27926;p[41]=29178;
p[42]=29110;p[43]=30941;p[44]=33585;p[45]=33056;p[46]=33317;p[47]
```

```
=35020;p[48]=35098;p[49]=32844;p[50]=33005;
p[51]=34311;p[52]=36833;p[53]=39870;p[54]=38345;p[55]=37392;p[56]
=36442;p[57]=33988;p[58]=33062;
p[59]=32753;p[60]=31330;p[61]=31473;p[62]=31238;p[63]=31115;p[64]
=30537;p[65]=29512;p[66]=29949;
p[67]=28929;p[68]=28386;p[69]=28458;p[70]=27964;p[71]=29209;p[72]
=30604;p[73]=31225;p[74]=29191;
p[75]=26370;p[76]=23418;p[77]=21426;p[78]=18105;p[79]=16738;p[80]
=15133;p[81]=14107;p[82]=12447;
p[83]=10997;p[84]=9411;p[85]=8152;p[86]=6914;p[87]=5836;p[88]
=4815;p[89]=3976;p[90]=3233;p[91]=2625;
p[92]=1976;p[93]=1552;p[94]=1110;p[95]=751;p[96]=483;p[97]
=317;p[98]=220;p[99]=132;
time 2010 2020;
prt <n> p;
option freq u; //undated
time 1u 99u;
p = rotate(p!a, 1); //definition of p!u
#t = 2010, 2020; #t = #t.strings();
plot p[#t];
```

This produces the following plot, with the age dimension on the x-axis.



Plot file editor

At the moment, Gekko uses xml files to store the plot settings. In the longer run, another format might be chosen, and an graphical interface to change these settings might be provided.

Until then, it is recommended that you use the in-built XML Notepad editor to edit the XML files, cf. the XEDIT statement (if used, choose 'View' --> 'Expand All' to unfold all XML nodes). You may also use Notepad (cf. the EDIT statement), but it is recommended to use a specific XML editor for editing the tables. Using a simple text editor like Notepad entails some potential problems; there will be no check that the

XML syntax is correct. Also, the XML syntax represents some characters in a special way: notably the `<` , `>`, and `&` characters (these should be written `&lt;`, `&gt;`, and `&amp;`).

If the file is not in valid XML syntax, Gekko will complain that the file is invalid and abort.

Note

You may close the PLOT graph window by pressing the 'Esc' button. Besides, to quickly close all PLOT windows, you may alternatively use the CUT statement or the "Close all PLOT and DECOMP windows" button.

PLOT produces a graph by means of the open-source program gnuplot as an embedded underlying engine.

In the graph window, you may change the so-called operators by clicking on the radio buttons and checkboxes. This way, you can quickly change to for instance percentage growth rate etc. You may copy-paste the graph to e.g. a word-processor like Word by clicking the ‘Copy Link’ button, but in Word you have to use the link with Insert --> Pictures --> This Device (Word does not support putting a .svg file directly on the clipboard). There are also options to save the graph as a emf/svg/png/pdf files.

When plotting variables, a `!Search` yellow warning sticker may be shown on the plot. When a variable does not have an explicit databank indicated like for instance `b:x`, and when databank searching is active (`option databank search = yes`), Gekko will search for the variable in the numbered databanks (numbers 1, 2, 3, and so on, cf. the F2 databank list window). When searching, and when a variable is found in position 2 or lower in the databank list, the yellow warning sticker is shown. This warning intends to guard against the presence of unexpected variable "holes" in the first-position databank. (The warning will never be shown regarding 'Ref' type plots, because in that case, variables are always taken from the databank in the 'REF' position in the F2 databank list).

Per default, PLOT will place annual and undated data at the x-tics, and quarterly and monthly data between x-tics. See `option plot xlabels ...` options, also if you prefer to use 15, 16, 17 etc., instead of 2015, 2016, 2017, etc.

Please note that the same graph may look different in different "environments". The Gekko graph window shows a .svg file, and the same .svg file may look a bit different when imported into Word (or converted to pdf or printed from Word). Both .emf, .svg and .pdf are vector formats, whereas .png is a raster format (bitmap).

See the rotate() function if you need to plot for instance age profiles of array-series that are defined in the age dimension.

Planned enhancements:

- Stacked curves and histograms shown as shares (summing to 1 or 100), perhaps using operator <s> for shares.
- Multi-plots.
- 3D-plots.
- Scatterplots.
- Free-floating labels and arrows.
- Outputting in more file formats like latex, etc.

Related options

OPTION plot all dash order = 1;
OPTION plot decimalseparator = period; [comma|period]
OPTION plot elements max = 200;
OPTION plot lines points = yes; [yes|no]
OPTION plot palette = gekko; [gekko|excel|dream]
OPTION plot using = [filename];
OPTION plot xlabels annual = at; [at|between]
OPTION plot xlabels nonannual = between; [at|between]
OPTION plot xlabels digits = 4; [2|4]
OPTION plot zoom dpiscale = 0;
OPTION print index date = 'start';
OPTION print index value = 100;

Related statements

PRT, SHEET, CLIP, TABLE, XEDIT, EDIT, CUT

4.3.64 PROCEDURE

PROCEDURE is used to define user-defined procedures. Such procedures do not return variables like a user-defined FUNCTION, but are instead used in a similar way to in-built Gekko statements. A procedure without arguments is similar to running a program file, cf. RUN. See the LOCAL keyword if you need variables defined inside the procedure to be inaccessible/nonexistent outside the procedure.

Libraries
Beginning with Gekko 3.1.12, it is possible to organize user-defined functions and procedures in libraries. These are basically just .zip-files that are loaded with the LIBRARY statement.

Beware: arguments with negative numbers
Procedures are basically simplified user functions, omitting parentheses and commas for convenience. A procedure call like `f 1 2 3;`, could always be 'translated' into a completely equivalent function call `f(1, 2, 3);` instead. But how to interpret a procedure call like for instance `f 1 -2 3;`? In Gekko, this is not equivalent to `f(1, -2, 3);`, but instead to `f(1-2, 3);`, because blanks are ignored in interpretation of mathematical expressions. If you need to use negative arguments in a procedure, you are advised to either:

- Put a parenthesis around the negative number when calling the procedure, for instance `f 1 (-2) 3;`.
- Transform the procedure into an equivalent stand-alone user function `f(1, -2, 3)`, cf. FUNCTION.

You may decorate a procedure with a <>-option field containing an optional time period. Procedures allow optional parameters with default values, and the procedure may prompt (ask) the user about these parameters, if `f?` is used instead of `f`, where `f` is the name of the procedure.

Syntax

```
procedure procname <date t1, date t2>, type1 var1 label1 = default1,
type2 var2 label2 = default2, ...;
  body ;
end;
```

t1, t2	(Optional). You may state optional time period parameters inside <>-brackets, for instance `procedure f <date %t1, date %t2>, series x;` after which `%t1` and `%t2` are assigned to for instance 2020 and 2030 in the call `f <2020 2030 z;`. If the procedure is called without <>-brackets, for instance `f z;` , the parameters `%t1` and `%t2` are assigned

	to the local/global time period instead. Using <>-brackets in a procedure call does not in itself change the local time period inside the procedure: use for instance the BLOCK structure to do that. See examples.
type1, ...	Types of variables: `series`, `val`, `date`, `string`, `list`, `map`, `matrix`. You may also use the special `name` type for parameters, which behaves 100% as a `string` inside the procedure, but where the single quotes are omitted when calling the procedure from outside (the shorter call `f y` is used instead of `f 'y'`).
var1, ...	The parameters/variables/expressions
label1, ...	(Optional). A label for the parameter, used if the procedure is promting (called with `f?`). See more about prompting below.
default 1, ...	(Optional). A default value for the parameter. If the parameter is omitted, the default value is used. If the procedure is asked to prompt (called with `f?`) and the parameter is omitted, the default value is shown in the dialog box. In the dialog box, `Enter` or `Escape` will return the default value, and fire up the next dialog box (for the next optional parameter). If a `;` is entered in the dialog box, all the remaining parameters attain their default values, and no more dialog boxes are shown. For string input, the use of quotes (`'`) in the input box is optional. At the moment, only `val`, `date` and `string` types can be used for prompting input boxes.
procname	The procedure name. The name cannot be the same as an existing Gekko statement name.
body	The procedure body, that is, the statements to be performed. If the body contains a RETURN statement, the procedure will abort at that point (just like a normal program file, cf. RUN).

Tip: if you need to stop execution at a specific point/line to inspect variables etc., try inserting a STOP statement. This will abort from all called program files/procedures/functions, without executing anything more after the STOP (this is not the case regarding RETURN). Therefore, STOP can be practical for debugging, etc.

When calling the procedure, the arguments are separated with blanks, not commas (this makes procedures syntactically more similar to Gekko statements than Gekko functions).

Examples

The following examples illustrate the use of PROCEDURE. This procedure has no arguments, and functions rather like a .gcm file (cf. RUN):

```
procedure now;
  tell currentTime();
end;
//----------------
now;
```

The following procedure multiplies two values, and prints out the result:

```
procedure mulval val %x, val %y;
  tell '';
  tell string(%x*%y);
end;
//----------------
mulval 3 4; //12
```

Note that the mulval arguments are separated with blanks, not commas. The following procedure adds three periods to the date 2000, printing out 2003:

```
procedure add3 date %x;
  tell '';
  tell string(%x+3);
end;
//----------------
add3 2000a1;
```

The following procedure prints out 'sunshine':

```
procedure shine string %x;
  tell '';
  tell %x + 'shine';
end;
//----------------
shine 'sun';
```

If you prefer to call the procedure with `shine sun;` instead of `shine 'sun';`, you may use the name type:

```
procedure shine name %x;
  tell '';
  tell %x + 'shine';  //%x behaves completely as a string
end;
//----------------
shine sun;  //the name type only has to do with how the procedure
is called
```

The following procedure adds 'a' to a list, printing the elements 'x1', 'x2', 'a':

```
procedure adda list #x;
  #add = #x + 'a';
  print #add;
end;
//-----------------
#xx = x1, x2;  //or: #xx = ('x1', 'x2');
adda #xx;
```

Dividing pairs of series:

```
procedure div list #x, list #y;
  for val %i = 1 to #x.len();
    prt {#x[%i]}/{#y[%i]};
  end;
end;
//-----------------
create a1, a2, b1, b2;
series a1 = 10; series a2 = 20;
series b1 = 15; series b2 = 25;
div ('a1', 'a2') ('b1', 'b2');  //you cannot use "div a1, a2 b1,
b2;" here.
```

The following multiplies two matrices:

```
procedure mulmatrix matrix #x, matrix #y;
  print #x * #y;
end;
//-----------------
#a = [1, 2; 3, 4];
#b = [9, 8; 7, 6];
#z = [1, 1; 1, 1];
mulmatrix #a #b+#z;
```

This following procedure divides two series and prints the result:

```
procedure divser series x, series y;
  prt x/y;
end;
//-----------------
time 2000 2001;
create xx, yy;
xx = 2;
yy = 3;
divser xx yy;
```

Alternatively, the same kind of procedure can be made with name types:

```
procedure divser name %x, name %y;
  prt {%x}/{%y};
end;
//-----------------
time 2000 2001;
create xx, yy;
xx = 2;
yy = 3;
divser xx yy;
```

Using `name` instead of `series` type has some advantages if, for instance, you wanted to pick out the second series from the Ref databank instead. In that case, you could just use `prt {%x}/@{%y};` instead of `prt {%x}/{%y};` inside the procedure body.

Local period example

```
procedure f <date %t1, date %t2>;
  block time %t1 %t2;
    y = 100;
  end;
end;

time 2001 2001;
f;                     //y will be set to 100 in 2001
f <2003 2003>;         //y will be set to 100 in 2003
prt <2001 2003 n> y;

//Result:
//                  y
//2001       100.0000
//2002              M
//2003       100.0000
```

When calling `f;`, the global time period is used for `%t1` and `%t2`, whereas when calling `f <2003 2003>;`, we get `%t1` = `%t2` = `2003` inside the procedure. See also the similar example regarding user-defined functions.

Prompt and default values example

Gekko procedures allow default values, and prompting regarding these.

```
procedure f val %x1, val %x2 'parameter 2' = 1, val %x3 'parameter
3' = 2;
  tell string(10000 * %x1 + 100 * %x2 + %x3);
end;
f 9 3 4;  //--> 90304
f 9 3;    //--> 90302
f 9;      //--> 90102
f? 9 3;   //enter 5 into the dialog box --> 90305
```

```
f? 9;     //enter 6 and 7 into the dialog boxes --> 90607
f? 9;     //enter 6 and ';' into the dialog box --> 90602
```

Beware that `f` or `f?` will fail with an error, since the first parameter is required. As shown regarding the last procedure call, you may terminate a sequence of input boxes with `;`, which means the default values are used for the current and following parameters. Pressing `Enter` or `Escape` returns the default value, and opens up the next input box. For prompt input, only the variable types `val`, `date`, `string` and `name` are supported at the moment (for name type, use for instance `... , name %x2 'parameter 2' = 'x', ...`).

Note

Procedures and user functions do not live in databanks, and are hence not affected by CLEAR, CLOSE, READ, etc., but are removed with RESTART or RESET. See also FUNCTION if you need to use return variables.

If a procedure is defined without <>-brackets to indicate time, it may still be called with <>-brackets. In that case, the time period inside the brackets is just ignored.

If you need to run the same piece of code many times (for instance inside a loop), defining and calling a procedure is efficient. Running a .gcm file entails some fixed loading, parsing and compiling costs each time it is called. These costs are not present when calling a procedure.

You can at most use 14 arguments, else use maps to bundle incoming arguments. Per default, all arguments are passed by value, not by reference (cf. `option system clone`). This means that procedures cannot have so-called side-effects on the incoming arguments.

It is planned to introduce the type `namelist` in addition to the `name` type, so that an argument like `(a, b, c)` can mean `('a', 'b', 'c')` internally.

Related statements

FUNCTION, LIBRARY, LOCAL, RUN

4.3.65 PRT

The PRT statement prints variables or expressions (you may use P, PRI or PRINT as synonyms for PRT). Note that printing mixed frequency timeseries is supported.

Guide: printing
For an easier introductory guide on Gekko printing, see this page.

Regarding series printing, the output can be transformed and formatted in different ways: transformations are done with so-called operators (for instance `p` or `pch` for percent change). Formatting includes transposing the result ('rows' option), setting width, decimals etc. The MULPRT statement is very similar to PRT, but compares data in the first-position and reference databanks. You may use the more detailed DISP statement, if you need more specific information regarding a series (data period, label, etc.: DISP also allows for equation browsing, if a model is loaded).

For variables taken from other banks than the first-position databank, you may use colon (':') to indicate the bankname, and for series you may use '!' to indicate a frequency different from the current frequency (for instance `b2:x!q` will print the quarterly series `x!q`, taken from the bank `b2`). You may use `@x` to indicate variables taken from the Ref (reference) databank (alternatively, you can use `ref:x`), and you may use `<bank=... ref=...>` to locally change the databanks used, instead of first using OPEN and then CLOSE. An array-series `y` can be printed with `prt y;`, where Gekko will print out all the elements in all dimensions. Otherwise `prt y[#i];` may be used to print all `#i` (a list of strings) elements in a dimension, or `y['a']` or the shorter `y[a]` can be used to print single elements.

PRT can also print out other variable types than series, for instance `prt b2:#m;` will print out `#m`, where `#m` could for instance be a list) from bank `b2`. Scalars can be printed with for instance `prt %v;`, where `%v` could for instance be a value. To show/count the number of variables of a particular type, you may use `val ?`, `date ?`, `string ?`, `series ?`, `list ?`, `map ?`, `matrix ?`, or MEM (for scalars).

Please note that after any PRT, you may click the Copy-button in the main window to copy-paste the print to Excel or other spreadsheets (CLIP will do the same thing, else see SHEET). Note that Gekko 3.0 supports printing (and plotting) series with mixed frequencies.

Syntax

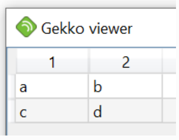
```
prt < period  operators I=... decimals  width  ROWS  FILTER=...
TITLE=...  COLLAPSE=... NOMAX  BANK=...  REF=...  SPLIT  MISSING=...
VIEW >  elements   FILE=filename;
```

- If no period is given inside the <...> angle brackets, the global period is used (cf. TIME).
- If a variable without databank indication is not found in the first-position databank, Gekko will look for it in other open databanks if databank search is active (cf. MODE).

where:	
period	(Optional). Local period, for instance `2010 2020`, `2010q1 2020q4` or `%per1 %per2+1`.
operators	*operator operator* ...
operator	'Long': `abs`, `dif`, `pch`, `gdif`, or 'short': `n`, `d`, `p`, `dp`, `m`, `q`, `mp`, `l`, `dl`, `r`, `rd`, `rp`, `rdp`, `rl`, `rdl`, `a`, `an`, `ad`, `ap`, `adp`, `al`, `adl`, `i`
decimals	DEC=number | NDEC=number | PDEC=number
width	WIDTH=number | NWIDTH=number | PWIDTH=number
elements	*element element* ...
element	variable label < *operators decimals width* >
details:	
operators	(See tables in examples section). Long operators: `abs`, `dif`, `pch` or `gdif`. These can be switched off by means of prefix `no` (for instance `nopch`), or added to existing default operators by means of underscore (for instance `_dif`). Default operators are `abs` and `pch` (absolute level and percentage growth is printed). Short operators: `d`, `p` and `dp` for time-transformations, and `m`, `q` and `mp` for multiplier-transformations. Prefix the time-transformations with `r` for reference values, or `a` for both first-position and reference values. You may use `i` or `i=...` for index=100 on the midpoint period or a chosen period. See also `option print prt ...`.
element	The variable can be a variable name, a list (for instance `{#m}`), or an expression. A label can be provided after the variable (must be a string, and will be ignored for lists). Operators here will override

	other operators (globals ones, or those set on the PRT statement), so element-operators are local to the particular element. You may use for instance `prt x x.fromseries('label');` to use label metadata.
I	Uses index = 100 for each print element, similar to the REBASE statement or rebase() function. For instance, `prt <i> x1, x2;` will index `x1` and `x2` at the start of the period, whereas `prt <i=2020q1> x1, x2;` will index `x1` and `x2` in the period `2020q1`. If you need to combine `<i>` with the print operators, use for instance `<i d>` to show time differences (and not `<id>`). Note that the indexing operation is performed *before* any print operator, as if REBASE or rebase() had been performed beforehand. Beware that while for instance `<i p>` is always equal to `<p>`, the multiplier version `<i q>` is not equal to `<q>`, because `<i q>` will always be = 0 at the index date.
YOY	Year-on-year: with this option, operators like `d` (time-differences) or `p` (percentage growth) use 4 lags for quarterly and 12 lags for monthly series. So for instance, for a quarterly series `x!q`, the statement `prt <yoy p> x!q;` shows the year-on-year growth rate `(x/x[-4]-1)*100` rather than the normal quarterly growth rate `(x/x[-1]-1)*100`.
DEC	Sets number of decimals, will apply to all kinds of numbers.
NDEC	Sets number of decimals for non-percentage numbers. See also `option print fields ndec...`.
PDEC	Sets number of decimals for percentage numbers. See also "`option print fields pdec..`.
WIDTH	Sets width, will apply to all kinds of numbers.
NWIDTH	Sets width for non-percentage numbers. See also `option print fields nwidth...`.
PWIDTH	Sets width for percentage numbers. See also `option print fields pwidth...`.
ROWS	If set, the result will be transposed, i.e., with variables running downwards. Corresponds to TRANSPOSE=yes for SHEET and CLIP.
FILTER	A timefilter can be activated or deactivated (see TIMEFILTER statement). With `<FILTER>` or `<FILTER=yes>`, the current timefilter is used. With `<NOFILTER>` or `<FILTER=no>`, any filtering is deactivated.

	The filter type can also be changed locally, for instance `<FILTER=hide>` hides the out-filtered periods, whereas `<FILTER=avg>` averages the out-filtered periods. See `option timefilter...`.
TITLE	A title in quotes. You can use HEADING as alias.
COLLAPSE	(Optional). This option will collapse quarterly or monthly data into annual averages or totals. Use `prt<collapse>`, `prt<collapse=avg>` or `prt<collapse=total>`. You may set the collapse globally, cf. `option print collapse = [avg|total|none];`. Only applies for `option print freq = pretty;` (which is default).
NOMAX	(Optional). Do not restrict the number of variables, cf. `option print elements max`.
BANK	(Optional). A bankname where variables are looked up. For instance `prt <bank = b1> x;` is equivalent to `prt b1:x;`. See also `<REF = ...>`. These options can be convenient instead of opening and closing banks.
REF	(Optional). A bankname where reference variables are looked up. For instance `prt <bank = b1 ref = b2 m> x;` uses banks `b1` and `b2` for the multiplier. See also `<BANK = ...>`. These options can be convenient instead of opening and closing banks.
SPLIT	(Optional). If set, the variables or expressions delimited by comma are shown separately. In this way, `prt x, y;` is shown as if it had been `prt x; prt y;`. This may be practical for comparisons of data with similar columns, for instance `prt <split> x[#i], @x[#i];`. In that case, you may prefer to use for instance the `<missing = m>` option, so that all columns (`#i`) are shown (and are hence aligned), regardless of whether the sub-series exist or not.
MISSING =	(Optional). With `<missing = ignore>`, PRT will deal with missing array sub-series and missing data values like GAMS, treating them as zero for sums and mathematical expressions, or skipping the printing of a sub-series if it does not exist. The following options are set locally and reverted afterwards: `option series array print missing = skip; option series array calc missing = zero; option series data missing = zero`. See also the appendix page on missings.
VIEW	(Optional). Used for nested lists. With this option, Gekko starts up a graphical viewer, which is primarily (at the moment) intended for viewing nested lists. For instance, a list like `#m = (('a', 'b'), ('c', 'd'));` would look like the following with `prt <view> #m;`:

	 1 2 a b c d In the longer run, views of series/array-series etc. may be implemented, too. The viewer handles convenient scrolling of large lists, and supports sorting etc.
FILE	(Optional). A filename that the print is put into. Filenames may contain an absolute path like `c:\projects\gekko\bank.gbk`, or a relative path `\gekko\bank.gbk`. Filenames containing blanks and special characters should be put inside quotes. Regarding reading of files, files in libraries can be referred to with colon (for instance `lib1:bank.gbk`), and "zip paths" are allowed too (for instance `c:\projects\data.zip\bank.gbk`). See more on filenames here.

Operators

There are two kinds of operators available. The easiest to remember are the 'long' ones: namely `abs`, `dif`, `pch` and `gdif`:

'Long' operators for PRT

abs	Absolute level: x
dif	Absolute time change: x-x[-1]
pch	Growth rate: (x/x[-1] -1)*100
gdif	Change in growth rate: (x/x[-1] -1)*100 - (x[-1]/x[-2] -1)*100

As default, PRT always prints out corresponding to `prt<abs pch>`, i.e., printing out the absolute level and the growth rate. These default options can be altered in `option print prt ...` (see 'Related options' below). For instance, to only print the level of a variable, use `prt<abs>`, to only print the growth rate, use `prt<pch>`, and to print absolute time change, use `prt<dif>`. You can alternatively switch off options with prefix `no`, for instance `prt<nopch>` (same as `prt<abs>`) etc. In addition, you can use the 'glue' prefix '_' to add options to existing options. For instance, `prt<_dif>` corresponds to `prt<abs dif pch>`, because `dif` is added to the default operators (`abs` and `pch`). You may put the codes after individual elements, for instance `prt var1<pch> var2<dif>`, to have `var1` displayed as `pch` and `var2` displayed as `dif`. Codes put on an element override more general codes put directly after PRT, so `prt<pch> var1 var2<dif>` yields the same result.

As a supplement to these 'long' operators, there are some more advanced (and shorter) operators of type `d`, `p`, `m`, `q` etc. These are perhaps less mnemotechnic, but more concise (the `@` below indicates values taken from the reference databank):

'Short' operators for PRT

n	Absolute level: x. Equivalent to no use of operators.
d	Absolute time change: x-x[-1]
p	Growth rate: (x/x[-1] -1)*100
dp	Change in growth rate: (x/x[-1] -1)*100 - (x[-1]/x[-2] -1)*100
m	Absolute multiplier: x-@x
q	Relative multiplier: (x/@x-1)*100
mp	Multiplier in growth rate: (x/x[-1] -1)*100 - (@x/@x[-1] -1)*100
l	Log: log(x). [New in 3.0.3].
dl	Log-difference: log(x)-log(x[-1]). [New in 3.0.3].
r	Absolute level in reference databank: @x. Code 'rn' is equivalent.
rd	Absolute time change in the reference databank: x-x[-1]
rp	Growth rate in reference databank: (@x/@x[-1] -1)*100
rdp	Change in growth rate in reference databank: (@x/@x[-1] -1)*100 - (@x[-1]/@x[-2] -1)*100
rl	Log: log(@x)
rdl	Log-difference: log(@x)-log(@x[-1])
a	n + r (showing both, including percentage growth)
an	n + r (showing both, no percentage growth)
ad	d + rd (showing both)
ap	p + rp (showing both)

adp	dp + rdp (showing both)
al	l + rl (showing both)
adl	dl + rdl (showing both)
i i=...	Proportionally adjust the series to become 100 in the midpoint period (using `<i>`) or a chosen period like 2000 (using `<i=2000>`): x/x[indexperiod]*100 or @x/@x[indexperiod]*100. Correspondence: note that for instance `prt <i=2000> x;` yields the same result as `prt x.rebase(2000);`, cf. REBASE.

There is the following logic to the above codes. The important codes to remember are `d` for absolute time change, `p` for percent time change, `m` for absolute multiplier, and `q` for relative multiplier. Then the combination `dp` is easily read as time change in growth rate, and `mp` as multiplier difference in growth rate. Log-transformations are done with `l` or `dl` operators. This covers the first and second sections of the above table. The third section is just the first section with prefix `r` (for reference databank), and shows that same transformations as in the first section, just for the reference databank values instead of the first-position databank values. Of course, you can always write `prt <p> @gdp;` instead of `prt <rp> gdp;`, but for longer expressions and lists, the prefix `r` comes in handy.

Formatting, filters etc.

In addition to the above transformations, the print can be formatted regarding the width of each data column, and the number of decimals. Width and decimals can be set in the PRT option field, or in element option fields. In the PRT option field, you may indicate the width and number of decimals like this: `prt<width=10 dec=3>`, or you may set absolute and percentage fields individually: `prt<nwidth=10 ndec=0 pwidth=6 pdec=1>`. This will yield absolute fields 10 characters wide with no decimals, and percentage fields 6 characters wide with 1 decimal. The width and decimals formatting can also be applied individually on each element, for instance `prt gdp<n dec=0> pgdp<p dec=1>`, printing `gdp` in levels with no decimals, and `pgdp` as growth rate with 1 decimal.

The output can be transposed by means of the `rows` keyword, for instance `prt<rows>gdp pgdp;`. This is handy for printing a long list of timeseries, or for copy-pasting the cells to a spreadsheet by means of the copy-button in the Gekko interface.

Finally, you can use a timefilter (see TIMEFILTER) in the PRT option field. This is convenient for suppressing individual observations when printing long time periods. First, you need to select the filtered periods, and then you can use for instance`prt <filter>` or `prt<nofilter>`. More advanced use is `prt<filter=avg>` in which case the out-filtered periods are aggregated into the shown periods (rather than simply skipped). See examples below.

Mixed frequencies

PRT can print out series of mixed frequencies in the same 'table' (frequencies `a`, `q`, `m` mixed, or frequencies `m`, `d` mixed), for example:

```
time 2001 2002;
xx1 = 10, 20;
option freq q;
xx2 = 1, 2, 3, 4, 5, 6, 7, 8;
option freq a;
prt xx1, xx2!q;  //or: xx1!a, xx2!q
```

The following is printed:

```
                    xx1           %          xx2!q          %
  2001
  q1                                         1.0000          M
  q2                                         2.0000     100.00
  q3                                         3.0000      50.00
  q4                                         4.0000      33.33
  a             10.0000           M

  2002
  q1                                         5.0000      25.00
  q2                                         6.0000      20.00
  q3                                         7.0000      16.67
  q4                                         8.0000      14.29
  a             20.0000      100.00
```

You may mix frequencies `!a`, `!q` and `!m` or `!m` and `!d` as you like, and mixed frequencies work for PLOT, too. Note that Gekko supports auto-collapse when printing, for instance `prt <collapse=total> xx2!q;` (this will print yearly totals).

Examples

A simple example:

```
time 2009 2012;
x = 100, 110, 120, 110;
p = 1.00, 1.02, 1.04, 1.06;
prt <2010 2012> x, x/p 'real';
prt <2010 2012 i> x, p;
```

This gives the following result, with both absolute levels and percentage growth:

```
                       x           %            real           %
2010            110.0000       10.00        107.8431        7.84
2011            120.0000        9.09        115.3846        6.99
2012            110.0000       -8.33        103.7736      -10.06

2010             91.6667       10.00         98.0769        2.00
2011            100.0000        9.09        100.0000        1.96
2012             91.6667       -8.33        101.9231        1.92
```

The label for x/p is given as a string after the expression. Please use a space to delimit variable and label. If a model is loaded, and x is a model variable, the first '%' would become '(E)%', because (E) or (X) are used to indicate endogenous or exogenous variables. The <i> option is used to set index = 100 in the midddle of the period, which can be useful for variables with different scale (especially in plots, not that i is not an operator like d, p, m, etc, but an option).

Using operator p inside the option brackets would provide you with the growth rates alone (the long operator pch has the same effect):

```
prt <p> x, x/p;
```

The print can alternatively be transposed with the <rows> option like this:

```
prt <2010 2012 p rows> x, x/p;
```

gives:

```
             2010       2011       2012
    x       10.00       9.09      -8.33
  x/p        7.84       6.99     -10.06
```

To set index=100 for the middle period (here: 2010), use <i> or <i=2010>. This can be practical for variables with differing levels:

```
prt <i> x, p;
```

with the following result:

```
                       x           %               p           %
2009             90.9091     -100.00         98.0392           M
2010            100.0000       10.00        100.0000        2.00
2011            109.0909        9.09        101.9608        1.96
```

```
 2012        100.0000     -8.33        103.9216      1.92
```

You may mix variables, lists, expressions and operators as you wish (just separate the elements with commas):

```
time 2009 2012;
xa = 10, 12, 11, 14;
xb = 6, 5, 7, 6;
time 2010 2012;
#m = xa, xb;  //or: #m = ('xa', 'xb');
prt <n> xa, xa[-1]*xb/xb[-1], xa/xa[2009], {#m};
```

Please note that the list `#m` is inside {}-curlies, because we are referring to the variables corresponding to the strings in the list. This prints out absolute time-differences in these variables/lists/expressions:

```
                                 xa[

                    xa  -1]*xb/xb[-1]    xa/xa[2009]            xa
             xb
  2010         12.0000         8.3333         1.2000       12.0000
        5.0000
  2011         11.0000        16.8000         1.1000       11.0000
        7.0000
  2012         14.0000         9.4286         1.4000       14.0000
        6.0000
```

If `xa` and `xb` are understood as, for instance, sectors `a` and `b`, you may instead use a list of these sector names, and then use `x{#m}` to auto-unfold into the variable names `xa` and `xb`.

```
time 2009 2012; xa = (10, 12, 11, 14); xb = (6, 5, 7, 6); time 2010
2012;
%i = 'a';
#m = a, b;      //or: ('a', 'b')
prt <n> x{%i};  //prints xa
prt <n> x{#m};  //prints xa, xb
```

Multiple operators can be used in one option field, as this example shows:

```
prt <n p r rp m q> fy;
```

This corresponds to a hand-made version of the MULPRT<v> statement, printing levels/growth in the first-position and reference databanks in addition to the multiplier differences (absolute and relative). As you see, this PRT statement contains multiplier differences (codes `m` and `q`), so by means of using short operators you are free to mix multiplier values into the print.

Formatting can be applied in the following way:

```
prt <nwidth=10 ndec=0 pwidth=6 pdec=1> fy;
```

This prints out the absolute levels and percentage growth rates (this way of printing is default, see `option print prt...`), with width 10 and no decimals for the levels, and width 6 and 1 decimal for the growth rates. You may use formatting on each element, for instance:

```
prt fy, fx, fm<ndec=0 pdec=1>;
```

In that case, only the last variable (`fm`) has a different number of decimals. Fixed periods can be indicated in brackets (for instance `[2005]` means that 2005-values are taken). Different databanks may be indicated:

```
prt fy, @fy, old:fy, old:{#m};
```

This prints out the variable `fy` from the first-position databank, `fy` from the Ref (reference) databank (@-indicator), fy from a databank with the name `old`, and variables corresponding to the list of strings `#m`, all taken from the `old` databank. The latter is opened by means of the OPEN statement (note that you can use F2 to see the list of open databanks).

Wild-card lists (inside braces {...}) may be used instead of regular lists:

```
prt {'fx*2'};
```

This will print out all variables starting with `fX` and ending with `2` in the first-position databank (use '?' as a single-character wild-card). In come statements, you may use for instance `fx*2` directly instead of `{'fx*2'}`, but in PRT this would be ambiguous (is `fx*2` a wildcard, or is it `fx` multiplied by `2`?).

You may insert labels or a heading into the PRT statement, the former only for non-list items.

```
prt fy 'GDP', ul 'Unemployment' heading = 'Scenario A';
```

To filter out periods, first define a TIMEFILTER. For instance:

```
timefilter 2003, 2005..2008, 2010..2015 by 2;
```

This way, the periods 2004, 2009, 2011 and 2013 will be hidden, whereas all other periods are shown. Note that TIMEFILTER defines the periods positively, i.e. the periods that are to be *included* when printing. If you print now, these four periods will just be skipped. To temporarily disable the filter in the print, use `prt<nofilter>`. To have the skipped periods aggregated into the shown periods, use `prt<filter=avg>`. For instance:

```
prt<2003 2015 filter=avg>fY;
```

gives the following:

```
                        fY        [%]
  2003          1314180.0000      0.38
  2004-2005     1360794.5000      2.37
  2006          1423985.0000      3.39
  2007          1446530.0000      1.58
  2008          1430309.0000     -1.12
  2009-2010     1367633.0000     -1.79
  2011-2012     1447238.5625      2.99
  2013-2014     1501831.8125      1.67
  2015          1534174.6250      1.45
```

For e.g. 2004-2005, the average for these two periods is shown. For the absolute level (the 'fY' column), a simple average is used, whereas for the percentage column ('[%]'), a more complicated averaging of growth rates is performed, in order to yield a consistent average growth rate for these two periods.

Looping over lists in combination with PRT is pretty straightforward. You may write:

```
for string %i = yf, x;
  for string %j = nf, nz;
    prt f{%i}{%j};
  end;
end;
```

This will print the series `fyfnf`, `fyfnz`, `fxnf`, `fxnz`. If you are using Gekko interactively, you can obtain non-executing linebreaks by means of Ctrl+Enter, and you execute the block of statements by means of marking the block before pressing [Enter]. Alternatively, and better, RUN the statements from a program file.

The example above can be done more easily (and will print them in one print):

```
#i = yf, x;
#j = nf, nz;
prt f{#i}{#j};
```

Note

If a model is loaded (see MODEL), the PRT statement indicates '(E)' for endogenous, and '(X)' for exogenous variables. Missing values are shown with a 'M' instead of numbers. If some variable is missing in the databank (or the databank does not exist), an error message will be issued.

Note that `prt <m> mybank:x;` prints out the difference `mybank:x - @x` (and `prt <r> mybank:x;` prints out `@x`), where `@x` is `x` in the Ref (reference) databank. So when printing a multiplier involving a named (OPEN) databank, Gekko will look for the same variable in the reference databank, in order to compute the multiplier.

You may change what is printed as default via the `option print prt ...` options (see 'Related options' below). For instance you may want to switch off printing of percentage growth permanently: `option print prt pch = no`.

You may use P, PRI or PRINT instead of PRT. You may use `diff` instead of the `dif` operator.

Related options

Relevant options regarding the PRT statement:

```
OPTION freq a; [a|q|m|d|u]
OPTION print collapse = none; [avg|total|none]; //show aggregates for
quarters and months
OPTION print freq = pretty; [pretty|simple];    //for quarters and
months
OPTION print width = 100;
OPTION print fields ndec = 4;
OPTION print fields nwidth = 13;
OPTION print fields pdec = 2;
OPTION print fields pwidth = 8;
OPTION print index date = 'start';
OPTION print index value = 100;
OPTION print prt abs = yes;
OPTION print prt dif = no;
OPTION print prt pch = yes;
OPTION print prt gdif = no;
OPTION series array print missing = error; [error|m|zero|skip]
OPTION series data print missing = error; [error|m|zero|skip]
OPTION series normal print missing = error; [error|m|zero|skip]
OPTION timefilter type = hide;
OPTION timefilter = no;
```

Related statements

MULPRT, PLOT, SHEET, CLIP, DISP, DECOMP

4.3.66 PREDICT

The PREDICT statement is used to predict an equation, calculating the right-hand side and assigning the result to the left-hand side variable. At the moment PREDICT is only used to predict model equations. In that case, predicting an equation can be thought of as performing a simulation of a model consisting of only one equation.

Consider the following model equation:

```
FRML _i    y = c + i + g;
```

Here, the statement `predict <2020 2030> y;` will calculate `c + i + g` over the given period, and assign the result to `y`.

Note the following about PREDICT:

- You may use PREDICT on any equation in a Gekko model, including equations of p-type (that is, equations that are part of the model file, but are not used in the simulation).
- PREDICT will calculate the left-hand side dyncamically if the left-hand side appears with lags on the right-hand side (corresponding to `series<dyn>` or `block series dyn = yes;`). Therefore, if the equation is `FRML _i    y = y[-1] + 1;`, using `predict y;` will accumulate `y` with 1 for each period.
- PREDICT is similar to `sim<res>`, with only one equation being predicted. But note (1) that lags may accumulate in PREDICT (cf. the above point), and (2) that predicted equations may influence each other if several PREDICTs are issued (or if PREDICT is issued on a list of names).

Syntax

predict < period > equations;

- If no period is given inside the <...> angle brackets, the global period is used (cf. TIME).

where:	
period	(Optional). Local period, for instance `2010 2020`, `2010q1 2020q4` or `%per1 %per2+1`.
equations	List of equations, designated by names of left-hand side variables

Examples

Consider the following model:

```
FRML _s   dlog(y) = 0.52 * dlog(yw) + 0.28 * log(yw[-1] - y[-1]);
FRML _s   log(yw) = log(x) - 0.3 * log(p) + 12.34;
```

Here, we are defining an intermediate variable `yw` in the second equation that in principle could be merged into the first equation. But `yw` may defined an equilibrium value of particular interest, but when new data for `x` and `p` arrives, the databank value of `yw` should reflect this. By means of PREDICT we may avoid making copies of equations for use in data preparation. Consider these two cases:

```
open <edit> newdata;
yw <2020 2020> = exp(log(x) - 0.3 * log(p) + 12.34);
close newdata;
```

Instead, we may use PREDICT, where the equation is fetched from `mymodel.frm`:

```
model mymodel;
open <edit> newdata;
predict <2020 2020> yw;
close newdata;
```

Note that in the latter example, the equation is not written explicitly, but is fetched from the model. If you were going to PREDICT several equations, the names of these can be stated as a list of equation names `#equnames`, with syntax `predict {#equnames};`.

Note

See the examples in the MODEL section regarding other uses of PREDICT (for instance, in the RUNAFTER$ section of a model).

Related statements

MODEL, SIM, RES

4.3.67 R_RUN

The R_RUN statement is used as an interface to R. The interface allows easy transfer of matrices from Gekko to R, execution of a R program, and easy returning of matrices from R to Gekko. Instead of matrices, you may alternatively use Apache Arrow files to communicate with R, cf. the example at the end of this page.

Compatibility note regarding syntax:
Starting with Gekko 3.1.8, the R interface syntax is simplified, and the former statements R_FILE, R_EXPORT are no longer needed. If you are using the deprecated syntax, you may easily upgrade to the new syntax. For instance, you may translate `R_FILE ols.r; R_EXPORT <target = 'data1'> #x, #y; r_run;` into simply `r_run <target = 'data1'> #x, #y file = ols.r;`, merging the three statements. The old syntax will work for a while, but please consider converting to the new syntax.

When executing R_RUN, you need R on your system, and Gekko will attempt to auto-detect the location of R. If this fails, you may indicate the location via `option r exe folder = ... ;` (Gekko will automatically add `R.exe` or `\R.exe`, unless the path ends with .exe, .bat or .cmd). If you just need to export matrices for use in R (without returning to Gekko), try the EXPORT<r> statement. Regarding an equivalent interface to Python, see PYTHON_RUN.

Syntax

```
r_run  <MUTE TARGET=...> matrix1, matrix2, ...  FILE = filename ;
r_run  <MUTE TARGET=...> filename ;
```

MUTE	(Optional). With this option set, R is run silently in Gekko. Alternatively, R output is shown in the Gekko main window. Do not used <mute> when debugging your R program, since it shows potential R error messages.
TARGE T =	(Optional string). If for instance `<target = 'data1' >`, the matrices are inserted at the exact location in the R file, where there is a line starting with `gekkoimport data1`. If the option is not given, the matrices are inserted at the top of the R file (this is often sufficient, the target logic is intended for larger R programs)
FILE =	Filenames may contain an absolute path like `c:\projects\gekko\bank.gbk`, or a relative path `\gekko\bank.gbk`. Filenames containing blanks and special characters should be put inside quotes. Regarding reading of files, files in libraries can be referred to with colon (for instance `lib1:bank.gbk`), and "zip paths" are allowed too (for instance `c:\projects\data.zip\bank.gbk`). See more on

	filenames here.

Example syntax:

```
r_run <target = 'data1'> #x, #y file = ols.r;
```

Example

The example below estimates (in R) a linear least squares model with five parameters. You may consult the OLS section to see the same parameters calculated via the OLS solver, or the MATRIX section to see the same parameters calculated via linear algebra. See also the Python interface.

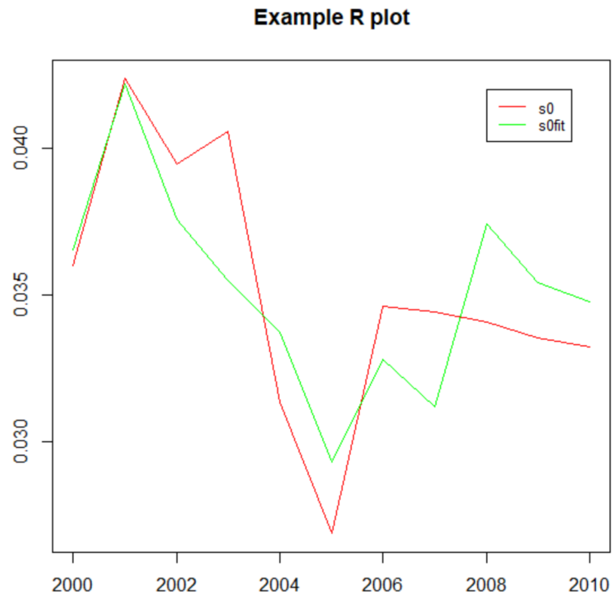
First, put the following R file `ols.r` into your working folder:

```
gekkoimport data1                # Gekko data (matrices x and y) is
inserted here
fit <- lm(y ~ x)                 # ols estimation
summary(fit)                     # prints output
beta <- fit$coefficients         # estimated parameters
yfit <- fit$fitted.values        # predicted values for y
gekkoexport(beta)                # writes beta vector back to Gekko
gekkoexport(yfit)                # writes fitted values back to Gekko

# ---------- example of plotting via R -----------
# somewhat convoluted because Rscript.exe is used as the R engine,
therefore showing a plot
# window is a bit more difficult than using for instance RStudio.
library(tcltk)
windows()
t = seq(2000, 2010)
plot(t, y, type="l", col="red", ann=FALSE)
lines(t, yfit, col="green")
title(main="Example R plot")
legend(2008, 0.042, legend=c("s0", "s0fit"), col=c("red", "green"),
lty=1:1, cex=0.8)
capture <- tk_messageBox(message = "")
```

Next, you can run the following program in Gekko:

```
reset; cls;
lna1 <1998 2010> = data('166.223000  173.221000  179.571000
187.343000  194.888000  202.959000
  209.426000  215.134000  222.716000  230.520000  238.518000
246.654000  254.991000') ;
pcp <1998 2010> = data('0.9502030   0.9699920   1.0000000
1.0235000   1.0401100   1.0605400
  1.0754700   1.0977800   1.1121200   1.1314800   1.1513000
1.1717600   1.1871600')  ;
bul1 <1998 2010> = data('0.0684791   0.0591698   0.0560344
0.0535439   0.0535003   0.0631703
  0.0649875   0.0578112   0.0473207   0.0404508   0.0467488
0.0472923   0.0475191')  ;
%t1 = 2000;
%t2 = 2010;
time %t1 %t2;
s0 = dlog(lna1);
s1 = dlog(pcp);
s2 = dlog(pcp.1);
s3 = bul1;
s4 = bul1.1;
#x = pack(%t1, %t2, s1, s2, s3, s4); //matrix
#y = pack(%t1, %t2, s0); //matrix
r_run <target = 'data1'> #x, #y  file = ols.r; //returns matrices
#beta and #yfit from R
prt #beta;
s0fit = #yfit[.., 1].unpack(%t1, %t2);
plot s0, s0fit;
```

The program prints R output on the screen, and plots actual and predicted values. Plotting is done using both R's plot function, and Gekko's own PLOT (the R plot is shown below).



The #beta vector looks like this:

```
#beta
                      1
     1           0.0298
```

```
      2           0.1445
      3           0.6139
      4           0.1867
      5          -0.3509
```

Some of the output from R shown in Gekko is the following (cf. the same example in the OLS section):

```
Coefficients:
              Estimate Std. Error t value Pr(>|t|)
 (Intercept)  0.029804   0.008942   3.333   0.0157 *
 x1           0.144517   0.227011   0.637   0.5479
 x2           0.613875   0.236473   2.596   0.0409 *
 x3           0.186740   0.202534   0.922   0.3921
 x4          -0.350908   0.203182  -1.727   0.1349
 ---
 Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

 Residual standard error: 0.003462 on 6 degrees of freedom
 Multiple R-squared:  0.625,   Adjusted R-squared:  0.3751
 F-statistic:   2.5 on 4 and 6 DF,  p-value: 0.1516
```

Note that in this example, the `<target= 'data1'>` option and the corresponding `gekkoimport data1` in the `ols.r` file are not really necessary, since the data could just be put at the top of the R file anyway. The code that is injected into the R file before it is executed looks like the following:

```
x = c(0.0304674549413991, 0.0232281261192072,
0.0160983506716728, .... )
dim(x) = c(11, 4)
y = c(0.0360024370055795, 0.0423704884205201,
0.0394838732643257, .... )
dim(y) = c(11, 1)
```

And the file that R produces for Gekko to consume looks like the following (this is actually what the gekkoexport() function in R does):

```
R2Gekko version 1.0
-------------------
name =  beta
rows =  5
cols =  1
0.02980389
0.1445173
0.6138751
0.1867401
-0.3509083
-------------------
...
```

This text-based way of interchanging data back and forth works fine, as long as the datasets are not too voluminous (otherwise see the following section). The interface is more stable than COM-based automation, and interchange of values, text, etc. could also be provided if needed.

Apache Arrow files

As an alternative to matrix-based communication with R, Gekko also supports the so-called Apache Arrow format. This is a dataframe-like format that can be understood by R, Python, Julia, Matlab and many others. In the longer run, Gekko will provide dataframes, too, but for the moment it is possible to write/export all timeseries from a Gekko databank (or subset of a databank) to an arrow file for consumption in, say, R or Python. At the moment, only series can be exported as arrow files, metadata is not included, and IMPORT<arrow> does not work yet (will soon). The Arrow interface may change, so please do not yet use the interface in "serious" production code. There is a tremendous potential in the Arrow project regarding easy, fast and reliable transfer of data between software packages. (See the equivalent Python example under PYTHON_RUN).

The following is a simple demo, highlighting some of the capabilities.

First, store the following `test1.r` file in your working folder:

```
library(arrow)   # may need: install.packages("arrow")
library(dplyr)   # may need: install.packages("dplyr")
df1 <- read_feather("test1.arrow")
print(df1)
df2 <- select(filter(df1, dims == 0), c(name, freq, per1, value))
print(df2)
```

Next, run the following Gekko statements:

```
reset; time 2021 2023;
x = 1, 2, 3;
series x1 = series(1); //1-dim array-series
x1[i] = 2, 3, 4;
x1[j] = 3, 4, 5;
series x2 = series(2); //2-dim array-series
x2[x, y] = 4, 5, 6;
x2[x, z] = 5, 6, 7;
export <arrow> test1.arrow;
r_run test1.r;
```

The data consists of a normal series `x`, a 1-dimensional array-series `x1`, and a 2-dimensional array-series `x2`. The following R-output is shown in Gekko, after the `test1.arrow` file is read by R:

```
# A tibble: 15 x 7
   name  freq   dims dim1  dim2   per1 value
   <chr> <chr> <int> <chr> <chr> <int> <dbl>
 1 x     a         0 <NA>  <NA>   2021     1
 2 x     a         0 <NA>  <NA>   2022     2
 3 x     a         0 <NA>  <NA>   2023     3
 4 x1    a         1 i     <NA>   2021     2
 5 x1    a         1 i     <NA>   2022     3
 6 x1    a         1 i     <NA>   2023     4
 7 x1    a         1 j     <NA>   2021     3
 8 x1    a         1 j     <NA>   2022     4
 9 x1    a         1 j     <NA>   2023     5
10 x2    a         2 x     y      2021     4
11 x2    a         2 x     y      2022     5
12 x2    a         2 x     y      2023     6
13 x2    a         2 x     z      2021     5
14 x2    a         2 x     z      2022     6
15 x2    a         2 x     z      2023     7

# A tibble: 3 x 4
  name  freq    per1 value
  <chr> <chr> <int> <dbl>
1 x     a       2021     1
2 x     a       2022     2
3 x     a       2023     3
```

The first dataframe (df1) consists of all the data. The columns have the names name, freq, dims, dim1, dim2, per1, and value. These are either strings (chr), integers (int) or double precison numbers (dbl). In the dataframe <NA> represents missing data. The name and freq represent the Gekko name (for instance x!a), and dims represents the number of dimensions. Columns dim1 and dim2 are the two potential dimensions, per1 is the year (for daily frequency there will be a per2 representing month, and per3 representing day), and value is the observation.

In the second dataframe (df2), dims == 0 is selected (that is, selecting only normal timeseries), and only columns name, freq, per1, value are shown. This resembles a more Gekko-like print of the x timeseries. EXPORT<arrow> supports all Gekko frequencies, but the particular dataframe layout may change.

Note

Note that with r_run<mute>, you will not see any potential R errors on the screen. So please do not use <mute> when you are still debugging the R program.

Note that at the moment, the gekkoexport() function only takes one argument/matrix at the time.

You need to have R installed on your computer. Gekko will try to auto-detect the location of the R files on your system. Gekko uses Rscript.exe, not R.exe, in order for R to return output dynamically line by line. Typical R location is something like this: `c:\Program Files\R\R-3.6.2\bin\x64\Rscript.exe`. You may manually indicate the path via this option: `option r exe folder = ... ;`.

You can also use EXPORT<r> to export matrices to a file suitable for R.

Related options

OPTION r exe folder = ... ; //you may use this, if the auto-detection of the location of R fails

Related statements

PYTHON_RUN, OLS, MATRIX, EXPORT<r>, EXPORT<arrow>

4.3.68 PYTHON_RUN

The PYTHON_RUN statement is used as an interface to Python. The interface allows easy transfer of matrices from Gekko to Python, execution of a Python program, and easy returning of matrices from Python to Gekko. Instead of matrices, you may alternatively use Apache Arrow files to communicate with Python, cf. the example at the end of this page.

When executing PYTHON_RUN, you need Python on your system, and Gekko will attempt to auto-detect the location of Python (trying to locate it from the Windows PATH). If this fails, you may indicate the location via `option python exe folder = ... ;`. If you just need to export matrices for use in Python (without returning to Gekko), try the EXPORT<python> statement. Regarding an equivalent interface to R, see R_RUN.

Syntax

```
python_run  <MUTE TARGET=...> matrix1, matrix2, ...  FILE = filename ;
python_run  <MUTE TARGET=...> filename ;
```

MUTE	(Optional). With this option set, Python is run silently in Gekko. Alternatively, Python output is shown in the Gekko main window. Do not used <mute> when debugging your Python program, since it shows potential Python error messages.
TARGET =	(Optional string). If for instance `<target = 'data1' >`, the matrices are inserted at the exact location in the Python file, where there is a line starting with `gekkoimport data1`. If the option is not given, the matrices are inserted at the top of the Python file (this is often sufficient, the target logic is intended for larger Python programs)
filename	Filenames may contain an absolute path like `c:\projects\gekko\bank.gbk`, or a relative path `\gekko\bank.gbk`. Filenames containing blanks and special characters should be put inside quotes. Regarding reading of files, files in libraries can be referred to with colon (for instance `lib1:bank.gbk`), and "zip paths" are allowed too (for instance `c:\projects\data.zip\bank.gbk`). See more on filenames here.

Example syntax:

```
python_run <target = 'data1'> #x, #y file = ols.py;
```

Example

The example below estimates (in Python) a linear least squares model with five parameters. You may consult the OLS section to see the same parameters calculated via the OLS solver, or the MATRIX section to see the same parameters calculated via linear algebra. See also the R interface.

First, put the following Python file `ols.py` into your working folder:

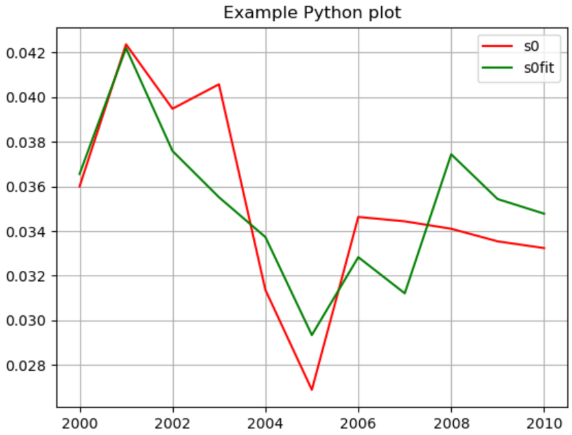
```
gekkoimport data1               # Gekko data is inserted here
import statsmodels.api as sm    # OLS functionality
x = sm.add_constant(x)          # constant term (ones column)
model = sm.OLS(y, x)            # define the model
results = model.fit()           # fit the model
print(results.summary())        # print results
beta = results.params           # estimated parameters
yfit = results.predict()        # predicted values for y
gekkoexport(beta)               # writes beta vector back to Gekko
gekkoexport(yfit)               # writes fitted values back to
Gekko

# ---------- example of plotting via Python ----------
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
t = np.arange(2000, 2011, 1)
fig, ax = plt.subplots()
ax.plot(t, y, label='s0')
ax.plot(t, yfit, label='s0fit')
ax.set(title='Python plot')
ax.grid()
plt.legend()
plt.show()
```

Next, you can run the following program in Gekko:

```
reset; cls;
lna1 <1998 2010> = data('166.223000  173.221000  179.571000
187.343000  194.888000  202.959000
  209.426000  215.134000  222.716000  230.520000  238.518000
246.654000  254.991000') ;
pcp <1998 2010> = data('0.9502030   0.9699920   1.0000000
1.0235000   1.0401100   1.0605400
  1.0754700   1.0977800   1.1121200   1.1314800   1.1513000
1.1717600   1.1871600')  ;
bul1 <1998 2010> = data('0.0684791   0.0591698   0.0560344
0.0535439   0.0535003   0.0631703
  0.0649875   0.0578112   0.0473207   0.0404508   0.0467488
0.0472923   0.0475191')  ;
%t1 = 2000;
%t2 = 2010;
time %t1 %t2;
s0 = dlog(lna1);
```

```
s1 = dlog(pcp);
s2 = dlog(pcp.1);
s3 = bul1;
s4 = bul1.1;
#x = pack(%t1, %t2, s1, s2, s3, s4); //matrix
#y = pack(%t1, %t2, s0); //matrix
python_run <target = 'data1'>  #x, #y  file = ols.py; //returns
matrices #beta and #yfit from Python
prt #beta;
s0fit = #yfit[.., 1].unpack(%t1, %t2);
plot s0, s0fit;
```

The program prints Python output on the screen, and plots actual and predicted values. Plotting is done using both Python's Matplotlib, and Gekko's own PLOT (the Python plot is shown below).



The #beta vector looks like this:

```
#beta
                         1
     1          0.0298
     2          0.1445
     3          0.6139
     4          0.1867
     5         -0.3509
```

Some of the output from Python shown in Gekko is the following (cf. the same example in the OLS section):

```
 ==================================================================
============
                  coef    std err          t      P>|t|      [0.025
     0.975]
 ------------------------------------------------------------------
------------
```

```
 const          0.0298      0.009      3.333      0.016       0.008
      0.052
 x1             0.1445      0.227      0.637      0.548      -0.411
      0.700
 x2             0.6139      0.236      2.596      0.041       0.035
      1.193
 x3             0.1867      0.203      0.922      0.392      -0.309
      0.682
 x4            -0.3509      0.203     -1.727      0.135      -0.848
      0.146
 ==================================================================
============
 Omnibus:                        1.122   Durbin-Watson:
       1.865
 Prob(Omnibus):                  0.571   Jarque-Bera (JB):
       0.895
 Skew:                           0.544   Prob(JB):
       0.639
 Kurtosis:                       2.122   Cond. No.
        253.
 ==================================================================
============
```

Note that in this example, the `<target= 'data1'>` option and the corresponding `gekkoimport data1` in the `ols.py` file are not really necessary, since the data could just be put at the top of the Python file anyway. The code that is injected into the Python file before it is executed looks like the following:

```
x =
numpy.array([[0.0304674549413991,0.0206121780628441, ...], ...])
y = numpy.array([[0.0360024370055795],[0.0423704884205201], ... ])
```

And the file that Python produces for Gekko to consume looks like the following (this is actually what the gekkoexport() function in Python does):

```
Python2Gekko version 1.0
------------------------
name = beta
rows = 5
cols = 1
0.0298038917827622
0.1445172984105897
0.6138751350035226
0.18674011629660836
-0.35090825010498294
-------------------
...
```

This text-based way of interchanging data back and forth works fine, as long as the datasets are not too voluminous (otherwise see the following section). The interface is more stable than COM-based automation, and interchange of values, text, etc. could also be provided if needed.

Apache Arrow files

As an alternative to matrix-based communication with Python, Gekko also supports the so-called Apache Arrow format. This is a dataframe-like format that can be understood by R, Python, Julia, Matlab and many others. In the longer run, Gekko will provide dataframes, too, but for the moment it is possible to write/export all timeseries from a Gekko databank (or subset of a databank) to an arrow file for consumption in, say, R or Python. At the moment, only series can be exported as arrow files, metadata is not included, and IMPORT<arrow> does not work yet (will soon). The Arrow interface may change, so please do not yet use the interface in "serious" production code. There is a tremendous potential in the Arrow project regarding easy, fast and reliable transfer of data between software packages. (See the equivalent R example under R_RUN).

The following is a simple demo, highlighting some of the capabilities.

First, store the following `test1.py` file in your working folder:

```
# first time use, first install the Apache Arrow package:
# "pip install pyarrow" or "conda install -c conda-forge pyarrow"
import pandas as pd
df1 = pd.read_feather('test1.arrow')
print(df1)
df2 = df1.loc[df1['dims'] == 0][['name', 'freq', 'per1', 'value']]
print(df2)
```

Next, run the following Gekko statements:

```
reset; time 2021 2023;
x = 1, 2, 3;
series x1 = series(1); //1-dim array-series
x1[i] = 2, 3, 4;
x1[j] = 3, 4, 5;
series x2 = series(2); //2-dim array-series
x2[x, y] = 4, 5, 6;
x2[x, z] = 5, 6, 7;
export <arrow> test1.arrow;
python_run test1.py;
```

The data consists of a normal series `x`, a 1-dimensional array-series `x1`, and a 2-dimensional array-series `x2`. The following Python-output is shown in Gekko, after the `test1.arrow` file is read by Python:

```
    name freq  dims  dim1  dim2  per1  value
```

```
0     x    a     0  None  None  2021    1.0
1     x    a     0  None  None  2022    2.0
2     x    a     0  None  None  2023    3.0
3    x1    a     1     i  None  2021    2.0
4    x1    a     1     i  None  2022    3.0
5    x1    a     1     i  None  2023    4.0
6    x1    a     1     j  None  2021    3.0
7    x1    a     1     j  None  2022    4.0
8    x1    a     1     j  None  2023    5.0
9    x2    a     2     x     y  2021    4.0
10   x2    a     2     x     y  2022    5.0
11   x2    a     2     x     y  2023    6.0
12   x2    a     2     x     z  2021    5.0
13   x2    a     2     x     z  2022    6.0
14   x2    a     2     x     z  2023    7.0

  name freq  per1  value
0    x    a  2021    1.0
1    x    a  2022    2.0
2    x    a  2023    3.0
```

The first dataframe (`df1`) consists of all the data. The columns have the names `name`, `freq`, `dims`, `dim1`, `dim2`, `per1`, and `value`. These are either strings, integers or double precison numbers. In the dataframe `None` represents missing data. The `name` and `freq` represent the Gekko name (for instance `x!a`), and `dims` represents the number of dimensions. Columns `dim1` and `dim2` are the two potential dimensions, `per1` is the year (for daily frequency there will be a `per2` representing month, and `per3` representing day), and `value` is the observation.

In the second dataframe (`df2`), `dims == 0` is selected (that is, selecting only normal timeseries), and only columns `name`, `freq`, `per1`, `value` are shown. This resembles a more Gekko-like print of the `x` timeseries. EXPORT<arrow> supports all Gekko frequencies, but the particular dataframe layout may change.

Note

Note that with `python_run<mute>`, you will not see any potential Python errors on the screen. So please do not use `<mute>` when you are still debugging the Python program.

Note that at the moment, the `gekkoexport()` function only takes one argument/matrix at the time. The `gekkoexport()` function will work with simple names, else you must indicate the name as a string, for instance `gekkoexport(results.params, 'beta')`.

You need to have Python installed on your computer. Gekko will try to auto-detect the location of Python from the Windows PATH, otherwise you may indicate the path via this option: `option python exe folder = ... ;` (Gekko will automatically add `python.exe` or `\python.exe`, unless the path ends with .exe, .bat or .cmd). To locate your python.exe file location, you may try this in Python: `import sys; print(sys.executable)` (this works for Python 3, for Python 2 you must omit the parentheses).

You can also use EXPORT<python> to export matrices to a file suitable for Python.

Related options

OPTION python exe folder = ... ; //you may use this, if the auto-detection of the location of Python fails

Related statements

R_RUN, OLS, MATRIX, EXPORT<python>, EXPORT<arrow>

4.3.69 REBASE

REBASE calculates an index series by dividing every observation of an existing series by a single observation or an average of several observations of the same timeseries. A bank name and/or prefix can be indicated, and the index value can be stated (default: 100). The statement ignores the global time period, and a local time period cannot be set.

Instead of the REBASE statement, you may alternatively use the similar `rebase()` function, for instance `PLOT x.rebase(2020);` (see under functions).

Syntax

rebase < FROMBANK=... TOBANK=... PREFIX=... INDEX=... > *variables date1 date2* ;

variables	A list of variable names (may include bank names)
date1	Rebase date (where the timeseries value attains 100, or the value stated with `<index = ...>`.
date2	(Optional). Rebase ending date. If this is active, instead of using only *date1*, the average of values over the period *date1*-*date2* is adjusted so that it (the average) attains the specified value.
BANK	Deprecated, use FROMBANK.
FROMBANK=	(Optional). A databank name where input timeseries are to be found, if databank names are not provided in the list of variables.
TOBANK=	(Optional). A databank name where all resulting timeseries end up. This databank has to be editable. [New in 3.1.9]
PREFIX=	(Optional). A prefix name regarding the resulting timeseries.
INDEX=	(Optional). The value of the index in the index period(s). Default = 100.

- If a databank name is not provided, the variable is not searched for in other databanks than the first-position databank.
- Looping: with a list like for instance `#m = a, b;`, you may use `rebase <prefix=i> x{#m} 2020;` to rebase `xa` into `ixa`, and `xb` into `ixb`.

Note: If the timeseries is quarterly or monthly, date1 is annual (a year), and date2 is not stated, Gekko will use the first period of the year as start date, and the last period of the year as end date. If the timeseries y is quarterly, `rebase y 2010;` is the same as `rebase y 2010q1 2010q4;`.

If you are using TOBANK or PREFIX (or both), new timeseries objects are created. Otherwise, it is the existing timeseries objects that are altered.

Example

```
reset;
mode data;
time 2010 2012;
y = 2, 3, 4;
rebase <prefix=i1> y 2011;
rebase <prefix=i2> y 2011 2012;
prt <n> y, i1y, i2y;
```

The result is the following:

```
                      y           i1y            i2y
  2010           2.0000       66.6667        57.1429
  2011           3.0000      100.0000        85.7143
  2012           4.0000      133.3333       114.2857
```

In the first one (`i1y`), the index series has the value 100 in 2011. In the second one (`i2y`), the index series has an average of 100 in 2011 and 2012 (the average of 85.7143 and 114.2857 = 100).

The following example illustrates the use of `FROMBANK`, `TOBANK`, `PREFIX` and `INDEX`.

```
reset; time 2010 2012;
y1 = 2, 3, 4;
open <edit> temp;
  y2 = 3, 4, 5;
close temp;
open temp;
open <edit> temp2;
clear temp2;
rebase <frombank=temp tobank=temp2 prefix=re index=100> work:y1, y2
2011;
prt temp2:rey1, temp2:rey2;
```

When the REBASE statement is issued, the databank `temp2` is in first position, `Work` is in second position, and `temp` is in third position. Of these, only `temp2` and `Work` are editable. The first series, `work:y1` has a databank indicated, and therefore `y1` is taken from `Work`. The second series has no databank indicated and is therefore taken from `temp` (because of the `FROMBANK` option). Both the rebased series are put into the `temp2` databank (because of the `TOBANK` option), and both are prefixed with `re` (because of the `PREFIX` option).

Note

The former BANK option is now called FROMBANK. The old option will still work for a while.

Related statements

SERIES, SPLICE

4.3.70 READ

The READ statement puts variables from a .gbk file (or other formats) into the first-position databank. A .gbk file is a Gekko-specific binary databank format that stores series, values, dates, strings, lists, maps, and matrices.

Before reading, the first-position databank is cleared, so after reading, the first-position databank will correspond to the file (this behavior may be altered with the `<merge>` option). After reading (optionally merging) data from the file into the first-position databank, the reference databank is always constructed as an exact copy of the first-position databank (a 'clone').

Because READ clears the first-position databank, it may often be practical to store variables in the Global databank. It can be practical to put general settings etc. in that bank, for instance default periods, often-used lists, etc. Variables in the Global databank will survive READ and CLEAR statements, and are in that sense long-lived.

When modelling using Gekko program files, it is advised to put a MODEL statement and an `option freq = ... ;` statement before the READ statement, so that READ knows the model and its frequency. When doing this, READ will auto-create technical model variables like add-factors etc. (D-, J-, and Z-type variables), and will also indicate if the lists of variables in the model and the databank are not matching (note: such auto-creation is not done for `data` mode).

READ is intended for .gbk files, and can be thought of as an enhanced version of IMPORT. In contrast to IMPORT, READ clears the first-position databank, reads all data regardless of the global time period (unless a time period or <respect> is used), and finally makes Ref a clone of the first-position databank. There are the following equivalences: READ = CLEAR<first> + IMPORT<all> + CLONE, and the inverse: IMPORT = READ<first merge respect>.

Because Gekko uses a so-called cache for databank files, the same file will load faster the next time it is read.

Syntax

```
read < period  ALL  FIRST  REF  MERGE > filename TO bankname;
```

filename	Filenames may contain an absolute path like `c:\projects\gekko\bank.gbk`, or a relative path `\gekko\bank.gbk`. Filenames containing blanks and special characters should be put inside quotes. Regarding reading of files, files in libraries can be referred to with colon (for instance `lib1:bank.gbk`), and "zip paths" are allowed too (for instance `c:\projects\data.zip\bank.gbk`). See more on filenames here.

	If the filename is set to '*', you will be asked to choose the file in Windows Explorer. The extension .gbk is automatically added, if it is missing.
period	(Optional). Without a time period indicated, Gekko will read all the data for all observations in the file. When a period is indicated, the read data(bank) is truncated.
FIRST	Reads the file into the first-position databank (#1 on the F2 window list).
REF	Reads the file into the reference databank (shown as REF on the F2 window list).
MERGE	(Optional). If MERGE is set, the data is merged into the existing first-position databank.
RESPECT	(Optional). With this option, if no period is given, the global period is used.
TO	(Optional). If `TO bankname` is indicated, Gekko will put the data into a seperate 'named' databank alongside the Work and Ref databanks. For instance, after `read adambk TO a;`, you may refer to the variables by means of colon, for instance `prt a:var1;`. If you use `read adambk TO *;`, the bankname will be the same as the file name. It should be noted that the databank will be read-only (protected) when opened like this (`read ... TO ...` is essentially the same as an OPEN statement)
TRACE=	(Optional). Default is `yes`. Set `no` to omit reading of data traces. The databank will load faster and use less RAM (if traces are present). Only relevant for Gekko databanks (.gbk files).

Examples

Reading a .gbk file called adambk.gbk is done with

```
read adambk;
```

or by writing

```
read *;
```

and then selecting the databank. Note that after such a READ of data into the first-position databank, the reference databank will always be created as an excact copy of the first-position databank. This behavior is practical for modeling purposes. You can merge with existing data in the first-position databank like this:

```
read <merge> adambk;
```

This merges `adam.gbk` with any pre-existing data in the first-position databank. Full or relative path names are possible:

```
read otherbanks\adam3;
```

This will look for `adam3.gbk` in the subfolder `otherbanks`, relative the the Gekko working folder.

Use the TO keyword like this:

```
read forecst2 TO f2;
```

This reads ` forecst2.gbk` into the named databank `f2`. After this, you may use for instance `prt f2:gdp;` to print out the timeseries `gdp` from this databank. This databank will be read-only. You may use `read forecst2 TO *;` if you wish to use the filename as databank name. It is possible to use `read * TO *;`. Using `read ... TO ...` is essentially the same as an OPEN statement.

Note

When reading, extension .gbk is automatically added if it is missing. Global time settings do not affect the READ statement, so all the data in the .gbk file is read into the first-position databank regardless of how the timeperiod is set in Gekko. (Use`read <respect>` to restrict the read data to the global time period).

Annual, quarterly, monthly, weekly, daily and undated data may co-exist as series in the same .gbk file, together with other variables types.

The gbk format is currently 1.2 (corresponding to Gekko 3.0/3.1.x) and comes in the following versions:

gbk file format versions

1.0	(July 2011). The file extension is .tsdx. Inside this zip-file there is a .tsd file, and a xml file with meta-information (does not state the databank

	format number, which is implicitly "1.0"). Only supports timeseries. Can be read by **Gekko 1.3.1** and later.
1.1	(November 2012). The file extension is .tsdx or .gbk. Inside there is a binary protobuffer file (either with extension .bin (older) or .data (newer)), and a xml file with meta-information, where databankVersion = "1.1". Only supports timeseries. Can be read by **Gekko 1.5.8** and later.
1.2	(November 2018). The file extension is .gbk. Inside there is a binary protobuffer file (databank.data) and a xml file with meta-information (DatabankInfo.xml), where databankVersion = "1.2". It supports seven variable types: timeseries (including array-series), val, date, string, list, map, matrix. Can be read by **Gekko 3.0** and later. The two older Gekko versions 2.5.1 and 2.5.2 can also read the 1.2 format.

If you need an old Gekko version (for instance Gekko 1.8, 2.2 or 2.4) to read a Gekko 3.x.x gbk databank, you can use Gekko 2.5.2 as a bridge. Write the databank in Gekko 3.x.x, read the databank in Gekko 2.5.2, write the databank in Gekko 2.5.2, and finally read the databank in the old Gekko version.

The option `copylocal` below copies the targeted file to a temporary file on the user's local hard disk before reading. This copying is typically very fast, and afterwards reading the temporary file is faster and more reliable, if the targeted file is located on a network drive. In general, this is a recommended option that alleviates some potential network problems.

The `cache = nongbk` option is set to skip `gbk` files. This is because the speed gains would be limited, because the internals of a .gbk file are essentially already a (protobuf) cache file. If the option is set to `all`, you may see a speed gain, because it is faster to calculate a MD5 hash code for a file than unzipping it.

The `.gbk` file may contain information regarding its corresponding model, last simulation period etc. If so, when READing the databank, a link to this model info is provided. This can be practical when in doubt about when the variables in a given databank were simulated, the simulation period, the model name and signature, etc.

To convert .tsd (or other formats) into a .gbk file, just read it with `import<tsd>;`, and WRITE it. Please note that a .tsd file operates with 8 significant digits (or less), so there will typically be a loss of precision compared to a .gbk file (which is in double-precision).

See the Gekko menu 'Options' --> 'Program dependency tracking' or use `option global dependency tracking = ...;` to activate dependency tracking, so that the use of external files (for instance program files, read/written databanks etc.) are shown as a list at the end of a Gekko session.

Related options

OPTION databank create auto = no; [yes|no]

OPTION databank create message = yes; [yes|no]
OPTION databank file cache = nongbk; [all|nongbk|none]
OPTION databank file copylocal = yes;
OPTION folder bank = [empty];
OPTION folder bank1 = [empty];
OPTION folder bank2 = [empty];
OPTION solve data create auto = yes; [yes|no]

Related statements

IMPORT, EXPORT, WRITE, OPEN, CLONE, DOWNLOAD

4.3.71 RENAME

RENAME changes names of variables inside a databank. If the rename operation involves two different databanks, the variable is moved between the banks (and possibly also renamed, as in for instance `rename bank1:x as bank2:y;`).

Note that 'naked' wildcards are allowed in this statement, so you may for instance use `a*b?c` as wildcard instead of the more cumbersome `{'a*b?c'}`.

Syntax

```
rename < BANK=... > variables1  AS  variables2;
```

BANK=	(Optional). A databank name indicating where the timeseries are located.
variables1	Variable name(s) or list(s)
variables2	Variable name(s) or list(s)

- If a databank name is not provided, the variable is not searched for in other databanks than the first-position databank.
- Looping: with a list like for instance `#m = a, b;`, you may use `rename x{#m} to y{#m};` to rename `xa` to `ya`, and `xb` to `yb`.

Example

Consider the two series `a1` and `a2` residing in the first-position databank. If we want to rename those into `b1` and `b2`, we could use:

```
rename a1, a2 AS b1, b2;
```

Lists can be used instead:

```
#a = a1, a2;  //or: ('a1', 'a2')
#b = b1, b2;
rename {#a} AS {#b};  //without the {}-curlies, the #a list itself
is renamed to #b!
```

Rename works across banks, too, in reality moving the variable:

```
rename b1:x AS b2:x;  //moving between banks
```

Wildcards are used like this:

```
rename x*2 AS y_*;
```

All variables starting with `x` and ending with `2` will obtain prefix `y_`.

See the similar COPY statement for more examples. RENAME is in reality similar to a COPY where the original object is deleted after copying. Also see the INDEX statement and the wildcard page regarding the syntax rules of wildcards.

Note

When using two lists of names, the lists must have corresponding (equal) length.

If preferred, you may use `rename ... TO ...` instead of `rename ... AS ...` .

Related statements

COPY, DELETE

4.3.72 RESET

The RESET statement is used to reset the workspace, similar to closing and reopening the Gekko application. With RESET, Gekko will not try to run any gekko.ini files to reload models, databanks, options, etc. (use RESTART to do that).

Among other things, RESET closes all opened databanks, clears the Work, Ref, Local and Global databanks, and in addition options, models, libraries etc. are cleared. Please note that when resetting, the frequency is always set to annual, and the time period is set from `t-10` to `t`, where `t` is the current year. See the RESTART or CLOSEALL statement, if you need frequency or time settings and other things like mode to persist after resetting.

Examples (clearing workspace)

Use this syntax to reset the workspace:

```
reset;
```

Clears the workspace (i.e. all Gekko RAM objects, including user functions and procedures). It does not run gekko.ini from the program and/or working folders, even if these files exist.

Note

The INI statement can be used to run Gekko.ini files after a RESET (RESET followed by INI is equivalent to RESTART).

Related statements

RESTART, INI, DELETE, CLOSE, CLOSEALL, CLEAR, CLS, CUT

4.3.73 RESTART

The RESTART statement is used to restart the workspace, similar to closing and reopening the Gekko application, and running any gekko.ini files present in the program folder (where gekko.exe is located) or working folder.

Among other things, RESTART closes all opened databanks, clears the Work, Ref, Local and Global databanks, and in addition options, models, libraries etc. are cleared. The gekko.ini files may reload options, models, databanks, libraries, etc. (use RESET to avoid such reloading). Beware that when restarting, the frequency is always set to annual, and the timeperiod is set from `t-10` to `t`, where `t` is the current year. If you need frequency or time settings to persist after a restart, you may put these into a gekko.ini file.

If you need persistent variables (settings) that survive CLEAR, READ, etc., you may put these in the Global databank. Beware the RESTART and RESET also clear the Global and Local databanks (you may use CLOSEALL to avoid that).

.

NOTE: RESTART is equivalent to `RESET; INI;`, and can be convenient in interactive sessions, where a RESTART statement may reload a given model/bank, etc. Note however, that RESTART will not fail, if a gekko.ini file is not found. This can have unintended consequences if the gekko.ini file is inadvertently deleted, so as a safer alternative to RESTART in program files, the user may put `RESET; RUN gekko.ini;` as the first line in the user's main Gekko program file. In that case, Gekko will abort with an error, if the gekko.ini file is not found.

Syntax

restart;

Examples

Use this syntax to restart the workspace:

```
restart;
```

Clears the workspace (i.e. all Gekko RAM objects, including user functions and procedures), and runs gekko.ini from the program and/or working folder if these files exist.

You might put the following statements into gekko.ini (this file would typically be located in your working folder):

```
//gekko.ini file for sim-mode
//---------------------------------
cls;
mode sim;
option folder model \models;
option folder bank \databanks;
option solve method = newton;
option freq = q;  //a (annual) is default
time 2012q1 2020q4;
model mymodel;
read mydatabank;
//---------------------------------
```

So in the gekko.ini file, put an (optional) a CLS statement first, set the mode, and then OPTION statements including time settings. Finally MODEL and READ statements (best in that order).

If you need to start up in data-mode, you could use the following file, also typically put inside the working folder:

```
//gekko.ini file for data-mode
//---------------------------------
cls;
mode data;
global:%path = 'm:\common\databanks';
option folder bank {global:%path};
option freq = q;  //a (annual) is default
time 1990q1 2015q4;
RUN lib.gcm;  //library of procedures/functions
//---------------------------------
```

```
//lib.gcm
//---------------------------------
procedure openbank name %bank;
  open {global:%path}\{%bank};
end;
//---------------------------------
```

In this example, a string `%path` is put into the Global databank, so that it can be accessed during the entire session. With `option folder bank` pointing to some central databank repository, existing databanks can be easily opened with OPEN, without indicating the full file path.

You may, alongside `gekko.ini` in the working folder, also put a `lib.gcm` program file with user-defined functions and procedures. If you RUN that file from the gekko.ini file, you will always have your user-defined functions/procedures at hand after a RESTART. In this example, after a RESTART, you may use the procedure `openbank bk2;`, which will open up `m:\common\databanks\bk2.gbk`.

As mentioned, you may put a `gekko.ini` file in the program folder (where `gekko.exe` is located, cf. Help --> About... in the Gekko main window). This file is always run before any other statements (including any `gekko.ini` in the working folder), so a `gekko.ini` in the program folder could contain general settings that change seldomly, like mode, frequency, time period, paths, etc..

Note

The RESET statement clears up in the same way as RESTART, but will skip any existing gekko.ini file.

The INI statement can be used to run the `gekko.ini` file separately (RESTART is the same as RESET followed by INI).

Note: With `option interface remote = yes`, Gekko may be remote-controlled from a special `remote.gcm` program file in the working folder (cf. OPTION).

Related statements

RESET, INI, DELETE, CLOSE, CLOSEALL, CLEAR, CLS, CUT

4.3.74 RETURN

The RETURN statement returns from a program file or procedure/function (i.e., does not execute the remainder of the file). It will return to any 'parent' program file/procedure/function calling that particular 'child' program file/procedure/function. If you need to return from the whole 'call stack' of program files/procedures/functions at once, you may alternatively use the STOP statement. (To stop/abort a program while it is running, you may use the red stop button in the user interface).

Syntax

```
return ;                              //return from program file
```

If RETURN is used to return from a FUNCTION (that is, if the function returns one or more variables), the following syntax is used:

```
return expression ;                   //return from function
```

Example (program files)

To see how RETURN works in program files, try creating the two program files `file1.gcm` and `file2.gcm` as shown below. Then call the first one of these from the statement prompt with `RUN file1;`. Next, if you issue a `prt %i;`, the value will be 1101, because the third line `file2.gcm` is never executed. With a STOP instead of RETURN in `file2.gcm`, the value of `%i` would be 1001, because the third line of `file1.gcm` (the 'parent' program file) would not have been have been executed.

```
------------- file1.gcm --------------
%i = 1000;
RUN file2;
%i = %i + 100;
--------------------------------------

------------- file2.gcm --------------
%i = %i + 1;
return;
%i = %i + 10;
--------------------------------------

RUN file1;
prt %i;
```

Note

- The similar STOP statement returns from all program files/procedures/functions at once.
- The EXIT statement effectively issues a STOP, and then afterwards closes the Gekko application.
- In function definitions, at least one RETURN statement is mandatory.

Note

If you wish to comment out a section of the program file, you may use `//` to comment out a single line, or `/*` followed by `*/` to comment out an arbitrary section (for instance spanning multiple lines).

Related statements

STOP, EXIT

4.3.75 RUN

The RUN statement runs a program file containing Gekko statements. The default extension for program files is `.gcm`. With a statement like `run test;`, Gekko will look for `test.gcm` in the working folder. If not found, it will additionally look in the folders designated with `option folder command = ...`, `option folder command1 = ...`, and `option folder command2 = ...`.

You may also run statements from outside Gekko, either via `gekko.exe` or via remote control (cf. below).

See also PROCEDURE, which can be thought of as a program file that also accepts arguments. If you need to call a program file many times, for instance from inside of a loop, calling a procedure instead will often run much faster because of the lack of re-parsing and re-compiling.

Syntax

`RUN filename;`

filename	Filenames may contain an absolute path like `c:\projects\gekko\bank.gbk`, or a relative path `\gekko\bank.gbk`. Filenames containing blanks and special characters should be put inside quotes. Regarding reading of files, files in libraries can be referred to with colon (for instance `lib1:bank.gbk`), and "zip paths" are allowed too (for instance `c:\projects\data.zip\bank.gbk`). See more on filenames here. The extension .gcm is automatically added, if it is missing. If the filename is set to '*', you will be asked to choose the file in Windows Explorer.

Tip: if you need to stop execution at a specific point/line to inspect variables etc., try inserting a STOP statement. This will abort from all called program files/procedures/functions, without executing anything more after the STOP (this is not the case regarding RETURN). Therefore, STOP can be practical for debugging, etc.

Example

You may run the program file `scenario.gcm` like this:

```
RUN scenario;
```

Or, if located in the sub-folder `\scenarios`:

```
RUN scenarios\scenario1;
```

This will run `scenario1.gcm` in the subfolder `\scenarios` (relative to the working folder). You may also use a wildcard `*` to open a dialog box for choosing the .gcm file:

```
RUN *;
```

The extension .gcm is added automatically if not provided. Other extensions may be used, just use them:

```
RUN gekko.ini;
```

will run the gekko.ini file (same as the INI statement).

---

gekko.exe parameters (batch job)

This is for more advanced users, but you may call `gekko.exe` with parameters, if you need to run Gekko sessions as batch jobs. The location of `gekko.exe` can be found via the Gekko menu: Help --> About... under "Program folder". Use this `gekko.exe` location, and indicate a working folder, like the following:

```
call "c:\Program Files\Gekko\gekko.exe" "-folder:c:\Gekkotest" "-
noini" "run r1.gcm; exit;"
```

This code can be issued from for instance a Windows CMD window. Here, Gekko is located in the folder `c:\Program Files\Gekko`, and the working folder is `c:\Gekkotest`. Any `gekko.ini` file in that working folder is skipped (`-noini`), and finally the two statements in the last quoted block are executed (`run r1.gcm; exit;`). In this case, the program file `r1.gcm` is run, and after this, Gekko is closed with the `EXIT` statement (if this is omitted, the Gekko window will persist, and you will have to close it manually). You may inser any Gekko statements, separated by semicolons (`;`). In the absence of a `-noini` parameter, Gekko will *first* run a possible `gekko.ini` file from the working folder, and *then* run any Gekko statements. To make Gekko run completely behind the scences ("stealth") without opening the graphical user interface at all, you may use the `-hide` parameter. Gekko will generally produce a `gekkooutput.txt` file for the user to inspect afterwards (for instance to inspect error messages). This file will not be generated if the `-nolog` parameter is used.

You may put code like the above in a batch file (`.bat`) that can be started simply by double-clicking it. If the CMD window is located at the `gekko.exe` folder, you may alternatively use the following simpler code:

```
gekko.exe -folder:c:\User\Gekkotest -noini run r1.gcm; exit;
```

But calling `gekko.exe` in this way is prone to problems with blank spaces in for instance folder or filenames. Therefore, using `call` with quotes (`"`) is recommended.

Calling `gekko.exe` from for instance a R or Python session is possible, but note that the inverse is also possible, calling R or Python from Gekko (cf. R_RUN and PYTHON_RUN). If you need to call Gekko from Excel, see the Gekcel project.

Remote control

With `option interface remote = yes;`, Gekko may be remote-controlled from a special `remote.gcm` program file in the working folder (cf. the description under OPTION). This is handy if you need to remote-control an existing Gekko instance from some other program, for instance a text editor. The above-mentioned `gekko.exe` parameters starts up a new Gekko instance, so you can use remote control to avoid that. You may try the following:

1. Start up Gekko normally
2. Type `option interface remote = yes;`
3. With an external text editor create a file named `remote.gcm`, containing the line `tell 'Hello from remote control';`. Put this file in the Gekko working folder.
4. Try changing the TELL line in `remote.gcm` to something else: Gekko will respond to that change.

Note that if you start with (3) above, and then fire up Gekko, Gekko will not run the `remote.gcm` file. Gekko only reacts when it detects *changes* in such a file.

Note

If you need to run the same piece of code many times (for instance inside a loop), consider using a PROCEDURE instead of calling RUN on a file. Running a .gcm file entails some fixed loading, parsing and compiling costs each time it is called. These costs are not present when using a procedure (only when it is loaded, not when called).

Instead of EDIT, most people use a "real" external text editor like for instance VS Code or Sublime Text to edit Gekko program files (.gcm). For VS Code, see its integrated Gekko extension (developed by the Gekko editor). For Sublime Text, see these open-source modules for Sublime and Gekko integration (including remote control).

Related options

```
OPTION folder command = [empty];
OPTION folder command1 = [empty];
OPTION folder command2 = [empty];
OPTION folder working = [empty];
OPTION interface debug = dialog; [dialog|none]
OPTION interface errors = normal; [old|normal]
OPTION interface remote = no; [yes|no]
OPTION interface sound = no; [yes|no]
OPTION interface sound type = bowl; [bowl|ding|notify|ring]
OPTION interface sound wait = 60;
OPTION interface warnings limit = 5;
```

Related statements

RETURN, STOP, PROCEDURE, FUNCTION, EDIT

4.3.76 SERIES

The SERIES (or SER) statement alters timeseries variables (often just called 'series'). Series variables have no starting symbol like `%` (scalars) or `#` (collections), but they may include a frequency indicator `!`, for instance `x!q` for `x` in a quarterly version. When Gekko starts up, the default frequency is annual, so `x` will be understood as `x!a`. A Gekko series can have annual (`a`), quarterly (`q`), monthly (`m`), weekly (`w`), daily (`d`) or undated (`u`) frequency.

Guide: timeseries
For an easier introductory guide on Gekko timeseries, see this page. For an introductory guide on array-timeseries, see this page.

Data tracing
Gekko timeseries statements support data tracing, making it possible to trace a series values backwards in time to its origins, see this page.

Compatibility note regarding lags:
Since series statements are calculated in a vector-like fashion in Gekko 3.0/3.1.x (in contrast to the Gekko 2.x versions), lags no longer automatically accumulate period-for-period, if the left-hand side variable is present with lags on the right-hand side (lagged dependent/endogenous variable). See more on the help page on dynamic statements. In Gekko versions >= 3.1.7, a statement like for instance `x = x[-1] + 1` will need to be decorated with an indication regarding whether it should accumulate (`<dyn>`) or not (`<dyn = no>`), cf. examples on the linked help page. Note that for absolute accumulations like `x <dyn> = x[-1] + 1`, you may use `x <d>= 1`, `x ^= 1` or `dif(x) = 1` instead (and there are also variants regarding relative accumulations, see below). Note that setting `<dyn>` (or `block series dyn = yes`) unnecessarily entails a speed penalty and should therefore not be used unless needed.

Gekko has to kinds of timeseries: **normal** series and **array-series**. Array-series allow the use of multidimensional indexes, for instance `x['a', 'b']`, picking out a sub-series with `'a'` in the first dimension and `'b'` in the second dimension. This could be for instance input-output cells, indicating the providing (`'a'`) and receiving (`'b'`) sector of intermediate goods. Array-series are quite similar to the map collection, but with special capabilities convenient for series data. The sub-series `x['a', 'b']`, or the shorter notation `x[a, b]`, is internally a normal series. Therefore, an array-series can be thought of as a container that contains a collection of normal series that can be accessed via the indexes.

Normal series (including array-subseries) will use the global time setting regarding the time period they are calculated over (cf. TIME), unless a local time period is indicated in the <>-option field.

If []-brackets are used to the right of a series variable, for a normal series it may either indicate a date (`x[2025]` or `x[2020q3]`) or a lag/lead (`x[-2]` or `x[+1]`). For an array-series, the []-index is used to pick out elements, for instance `x[a, b]`. These may be combined, like `x[a, b][2025]`. Beware that lags and leads must start with the symbol `-` or `+`, respectively, otherwise they are not interpreted as leads (so if `%i = 1`, you must use `x[+%i]`, not just `x[%i]`). For integer lags like `x[-1]` or `x[-2]`, you may use the shorter form `x.1` and `x.2`, too.

To put values into a series, you can use a list of values on the right-hand side, for instance `x = 1, -2, 3;` (the list of values could also be put inside a parenthesis). If you need to use blank-separated numbers, you can use the data() function, for instance `x = data('1 -2  3');`.

If you need to use alias names for series, you can use an `#alias` list to assign one name to another. Cf. the last part of this help page.

Note that a bank-less variable like for instance `x` on the right-hand side of a SERIES expression may be searched for in other databanks than the first-position databank, cf. the databank search page. Beware that the keys `[Tab]` or `[Ctrl+Space]` offer autocompletion on timeseries names (cf. here).

Syntax

```
variable <period operator KEEP=... LABEL=... SOURCE=... UNITS=...
STAMP=... DYN  MISSING=...> = expression;
variable[date] = expression;       //updating for one period
variable[indexes] = expression;    //array-series
f(variable) = expression;          //left-side function: dif(), pch(),
dlog(), log()
series ?;                          //show an overview series count from
all open databanks
```

Note: The `series` keyword known from versions prior to Gekko 3.0/3.1.x is now superfluous. If you prefer to use it, you can still use `series <2010 2020> x = ... ;` instead of `x <2010 2020> = ... ;`.

period	(Optional). Local period, for instance `2010 2020`, `2010q1 2020q4` or `%per1 %per2+1`.
operator	The operator can be `d`, `p`, `m`, `q`, `mp`, `l` or `dl`. See the 'Operators' section below.
KEEP=	If `<keep=p>` is used, Gekko will keep the growth rate of the left-hand series intact after the period over which the series is updated. For instance, `x <2020 2025 m keep=p> = 100;` will add 100 to `x` over the period 2020-25. The `keep=p` setting makes sure that the

	growth rate of `x` regarding 2026 and later already existing observations in `x` is the same as before the update. You may alternatively use `<keep=d>` in the same manner, to keep absolute changes rather than relative changes. Other `keep` operators than `d` or `p` are not implemented yet.
LABEL=	(Optional). Label (string) for the series, cf. DOC.
SOURCE=	(Optional). Source (string) for the series, cf. DOC.
UNITS=	(Optional). Units (string) for the series, cf. DOC.
STAMP=	(Optional). Stamp (string) for the series, cf. DOC.

DYN	(Optional). With this option, lagged dependent/endogenous variables like `x[-1]` in the expression `x = x[-1] + 1;` accumulate over time. Using `<dyn>` entails a speed penalty, so please do not use if not needed (in this particular case, `x ^= 1;`, `x <d>= 1;` or `dif(x) = 1;` could be used instead). Using the `<dyn>` local option, or equivalently putting the expression inside a `block series dyn = yes; ... ; end;` basically means that the expression is run *n* times successively, for each observation in the time period. Therefore, the following two are equivalent:
	Using <dyn>
	`x <2021 2023 dyn> = x[-1] + 1;  //the <dyn> option implicitly runs the expression three times`
	Explicit
	`x <2021 2021> = x[-1] + 1;  //these three lines are equivalent to the use of <dyn> above`
	`x <2022 2022> = x[-1] + 1;`
	`x <2023 2023> = x[-1] + 1;`
	Starting from Gekko 3.1.7, statements like `x = x[-1] + 1;` must have an indication of whether they are to be run dynamically or not, cf. the page on dynamic statements for much more information on such 'dynamics errors' and how to handle them.
	If the endogenous variable does not appear lagged on the right-hand side (like `x[-1]` or some other lag or lag function), running the expression in one go over 2021-23 yields the same result as running it in three tempi. Therefore, `<dyn>` is a waste of effort in such cases. It is not disallowed to combine `<dyn>` with time-operators like for instance `^` or `<d>`, or left-hand side functions like `dif()`, but such combinations can be rather hard to understand.

MISSING=	(Optional). With `<missing = ignore>`, SERIES will deal with missing array sub-series and missing data values like GAMS, treating them as zero for sums and mathematical expressions. The following options are set locally and reverted afterwards: `option series array calc missing = zero; option series data missing = zero`. See also the appendix page on missings.
variable	Left-side name
expression	Any expression
In addition to `=` (assignment), the following variants can also be used (see the 'Operators' section below): • `+=` add to existing • `-=` subtract from existing • `*=` multiply to existing • `/=` divide from existing • `^=` set absolute time change • `%=` set percent time change • `#=` add to percent time change	

- If no period is given inside the <...> angle brackets, the global period is used (cf. TIME).
- If a variable on the right-hand side of `=` is stated without databank, Gekko may look for it in the list of open databanks (if databank search is active, cf. MODE).
- Looping: with a list like for instance `#m = a, b;`, you may use `y{#m} = 1/(1 + x{#m});` to calculate `ya` from `xa`, and `yb` from `xb`.

Types of series

Normal series

Normal series look like the following example:

```
x = 100;
```

In that case, 100 is assigned to each observation in the global time period (cf. TIME). Different values for each observation can be assigned like this:

```
time 2021 2023;
x = 100, 110, 90;
y = (10*10, 113-3, 88+2);  //list with expressions must use parentheses
```

The right-hand side of the `x` series in this example is a comma-separated sequence of simple numbers. Beware that if you want to use a sequence of expressions, you must use a list definition (and lists are enclosed in parentheses, cf. the `y` series above).

You may indicate a local time period in the `<>`-option field:

```
time 2020 2030;
x <2021 2023 label='Gekko-variable'> = 100, 110, 90;
series <2021 2023 label='Gekko-variable'> x = 100, 110, 90; //the
same: old syntax
```

The local time period overrules the global period. If the three values corresponded to quarters for a quarterly series `x`, the statement `x!q = 100, 110, 90;` could be used. Alternatively, one could change the global frequency first like this: `option freq q; x = 100, 110, 90;`. In that case, you do not need to use the frequency indicator `x!q` explicitly, since `!q` is added implicitly to `x` in all places where the frequency is not stated. Note that the example above sets the label of `x` to 'Gekko-variable' (cf. also DOC).

The right-hand side of a series variable can be any legal Gekko expression that evaluates to a series, or anything that evaluates to a list of values of a suitable length. For list values, you may repeat them using `rep`, for instance `y = 1, 2 rep 2, 3;` is equal to `y = 1, 2, 2, 3;`. The last value in a list may be indicated with `rep *` which will repeat the item a suitable number of times, if the left-hand side is a series. For instance: `y <2021 2025> = 1, 2, 3 rep *;`, where the series will get values 1, 2, 3, 3, 3 over the period 2021-25.

Series names may be composed with {}-curlies, representing characters. For instance:

```
time 2010 2012;
a = 100; b = 200; xa = 1; xb = 2;
%i = 'b';
#i = ('a', 'b'); //or: #i = a, b;
prt <n> {%i}, {#i}, x{%i}, x{#i}; //the elements of #i have 'x'
prepended
```

Result (the four PRT arguments are shown in different colors):

```
                    b               a              b             xb
           xa             xb
  2010       200.0000       100.0000        200.0000         2.0000
       1.0000         2.0000
  2011       200.0000       100.0000        200.0000         2.0000
       1.0000         2.0000
  2012       200.0000       100.0000        200.0000         2.0000
       1.0000         2.0000
```

Array-series

For an easier introductory guide on array-timeseries, see this page. The dimensions of an array-series need to be stated when it is constructed. Afterwards, indexes are used to refer to its elements:

```
x = series(2);          //two dimensions
x[a, b] = 100;          //or: x['a', 'b'] = 100;
x[a, o] = 200;          //or: x['a', 'o'] = 200;
```

As seen, you may use the shorter `x[a, b]` instead of the more strict `x['a', 'b']`, when the elements are simple names, for instance not containing blanks or special symbols.

When dealing with timeseries given in some logical structure apart from time (for instance input-output cells), name composition is often used, for instance using the name convention `xab` and `xao` instead of `x[a, b]` and `x[a, o]`. Using array-series, there are convenient summing functions like `sum(#j, x[a, #j])`, summing up the second dimension of the array-series `x` (for instance, with `#j = ('b', 'o')`, the index `x[a, #j]` will correspond to `x[a, b], x[a, o]`). The same kind of logic can also be implemented with name-conventions, for instance `sum(#i, xa{#j})`, where `xa{#j}` will correspond to `xab, xao`. Still, array-series can be very practical in order to organize timeseries in some non-time structure/dimensions, and an array-subseries like for instance `x[a, b]` can be used in the same way as a normal timeseries `xab`. Also, with array-series there is no risk of name-collisions. For instance, `x[ab, c]` is clearly different from `x[a, bc]`, whereas a simple naming convention will produce the same name, `xabc`. This can be remedied with, for instance, underscores (`x_ab_c` vs. `x_a_bc`), but in that case why not just use array-series?

Elements that are simple numbers represented as strings may have values added or subtracted, for instance `x[#a+1]`, where `#a` could be a list of strings representing ages, like `('18', '19', ..., '80')`.

You may perform simple mathematical operations on array-series without indexes, for instance `p * x` in the above example, being equivalent to `p[#i, #j] * x[#i, #j]`. Such possibilities (array-series algebra) will be augmented.

Operators and left-side functions

The following tables presents the different operators:

Type	Operator	Example	Result	Note
Absolute	`^=`	`x ^= 1200;`	`x = x[-1] + 1200`	Same as `<d>` or `dif(x) = 1200;`. See also the

				<dyn> option.
Relative	%=	x %= 3.5;	x = x[-1]*(1+3.5/100)	Same as <p> or pch(x) = 3.5;. See also the <dyn> option.
Absolute	+=	x += 1200;	x = x + 1200	Same as <m>. You can also use -= to subtract values.
Relative	*=	x *= 1.03;	x = x*1.03	Similar to <q>. You can also use /= to divide with values.
Change in relative	#=	x #= 2.1;	x = x[-1]*(x0/x0[-1] + 2.1/100)	Same as <mp>. See also the <dyn> option.

In the formula regarding the # operator, x0 is the original timeseries, and x is the new one. Alternatively, the so-called 'short' operators may be used:

Type	Option	Example	Result	Note
Absolute	<d>	x <d>= 1200;	x <dyn> = x[-1] + 1200	Same as ^=. or dif(x) = 1200;. See also the <dyn> option.
Relative	<p>	x <p>= 3.5;	x <dyn> = x[-1]*(1+3.5/100)	Same as %= or pch(x) = 3.5;. See also the <dyn> option.
Absolute	<m>	x <m>= 1200;	x = x + 1200	Same as +=. You can also use -= to subtract values.
Relative	<q>	x <q>= 3;	x = x*(1+3/100)	Similar to *=. You can also use /= to divide with values.
Change in relative	<mp>	x <mp>= 2.1;	x <dyn> = x[-1]*(x0/x0[-1] + 2.1/100)	Same as #=. See also the <dyn> option.
Log	<l>	x <l>= 5;	x = exp(5)	Same as log(x) = 5;.
Relative	<dl>	x <dl>= 0.035;	x <dyn> = x[-1]*exp(0.035)	Same as dlog(x) = 0.035;.

Left-side functions:

Type	Option	Example	Result	Note

Absolute	dif()
	diff()
		dif(x) = 1200;	x <dyn> = x[-1] + 1200	Same as ^= or <d>=. See also the <dyn> option.

				You may use diff() as synonym.
Relative	`pch()`	`pch(x) = 3.5;`	`x <dyn> = x[-1] *(1+3.5/100)`	Same as `%=` or `<p>=`. See also the <dyn> option.
Log	`log()`	`log(x) = 5;`	`x = exp(5)`	Same as `<l>=`.
Relative	`dlog()`	`dlog(x) = 0.035;`	`x <dyn> = x[-1] *exp(0.035)`	Same as `<dl>=`.

Examples, normal series

Create a deflated price index (not an existing variable):

```
time 2010 2013;
create p1, p, rp1; //only necessary in sim-mode
p1 = 1.00, 1.12, 1.15, 1.14;
p = 1.00, 1.02, 1.04, 1.06;
rp1 = p1/p;
prt rp1;
```

Create a series with a given growth rate:

```
create x;             //only necessary in sim-mode
time 2011 2013;
x <2010 2010> = 1;    //uses a local time period
x %= 2.5;             //uses x is set to grow with 2.5 percent
annually
x <p>= 2.5;           //same as above, alternative syntax
pch(x) = 2.5;         //same as above, alternative syntax
prt x;                //grows with 2.5% p.a.
```

Change compared to the reference bank:

```
create x;                           //only necessary in sim-mode
time 2011 2013;
x = 1, 2, 3;
clone;                              //Ref bank made as copy of Work
bank
x <q> = 10;                         //or: x *= 1.10;
prt <n r m q> x;                    //level, ref-value, difference,
%difference
prt <n> x, @x, x-@x, 100*(x-@x)/@x; //same info, done manually
```

In the last PRT, `@x` is short for `ref:x`, that is, `x` from the Ref databank.

To set for instance a growth rate equal to another growth rate, you can use the <p> operator:

```
y <p> = pch(x);  //or: y %= pch(x), or: pch(y) = pch(x), or: y =
y[-1] * x/x[-1]
```

To change only one period, you may use:

```
tg[2020] = %v;  //%v is a scalar value
```

This will only set the 2020-value, and will work regardless of what the global sample might be. Used like this, at the same time stating a local period inside the <>-option field is not legal (or meaningful). Note that when using SERIES with []-brackets like this, a scalar value (or expression) is expected on the right-hand side of the equation. The above statement is functionally equivalent to the following:

```
tg <2020 2020> = %v
```

The $-operator can be used to "control" expressions, like an implicit IF-statement. For instance:

```
reset;
#m = a, b; %v = 10;
y1 = 3 $ ('b' in #m and %v == 10);
y2 $ ('b' in #m and %v == 10) = 3;
```

In the `y1`-statement, `y` will be 3 if the condition is true (if `'b'` is a member of `#m` and `%v` has the value 10), and else `y` will obtain the value 0. In the `y2`-statement, the whole statement is skipped if the condition is false, and in that case `y` will not exist at all. Note this conceptual difference regarding $-condition to the left of or to the right of `=`.

The $-operator can be used to switch between values inside a period, for instance:

```
reset;
time 2001 2005;
x = 10, 10, 11, 12, 10;
y1 = 110 $ (x == 10) + 111 $ (x <> 10);
y2 = iif(x, '==', 10, 110, 111);
```

The `y1`-statement illustrates the use of the $-operator for switching, and `y1` will contain the numbers 110, 110, 111, 111, 110 (the 10's are replaced with 110, and all other values are replaced with 111). The last `y2`-statement illustrates how to perform the same operation using the iif() function. The operation could alternatively be performed with FOR and IF statements, looping explicitly over each period, but using the $-operator or the iif() function is much more convenient here.

Adding 1000 to a series `jx` can be done with the `+` operator, or the `<m>` option:

```
jx <2010 2010> += 1000;
jx <2010 2010 m> = 1000;   //same result
```

Instead of updating with raw numbers, you may use scalar variables instead (in this case, you have to use parentheses to indicate the list, because the elements are not simple numbers):

```
%f1 = 0.02;
tg <2010 2012> += (%f1, 2*%f1, 0.01);
```

Using a list #m:

```
time 2010 2012;
#m = x1, x2;                  //or: ('x1', 'x2')
{#m} = 100, 80, 110;
{#m} <2010 2012> *= 1.02;   //x1 and x2 become 2% larger, could also
use <q> option
prt {#m};
```

Note that 1.02 is implicitly used for all three periods (you do not need to write `(1.02, 1.02, 1.02)`). Note also the {}-curlies in `{#m} = (100, 80, 110);`. Without the curlies, #m would become a list of the three values 100, 80, 110, which is not the intention. In `<2010 2012> {#m} *= 1.02;`, without the curlies, the expression would fail, since a list does not implement the `*=` operator. Finally, in `prt {#m};`, without the curlies, Gekko would print the strings 'x1' and 'x2', not the series `x1` and `x2`.

If you use `<keep=p>`, Gekko will keep the same growth rate in the data, after the time period where the variable is changed.

```
y <2007 2007 m keep=p> = 0.01;
```

This way, `y` has 0.01 added in 2007 (because of the `<m>` operator), and in all the subsequent years of data, the old growth rate in y is preserved (which is what the keep option does). Note that `keep=p` updates the series *outside* of the indicated period.

In sim-mode, you must first create a non-existing variable, but if the variable name starts with 'xx', it is automatically created:

```
xxvar = 27;  //works in sim-mode without prior create
```

If convenient, you may also use wildcard lists:

```
{'j*'} <2010 2010> = 0;
```

This sets all variables in the Work databank beginning with `j` to 0, for the given period.

You may set timeseries in other databanks than Work, for instance:

```
bank1:x = 100;
```

This will set the variable `x` to 100 in the bank `bank1` (cf. the OPEN statement), provided that the bank is unlocked. If you need to change timeseries in the reference databank, you may use the @-indicator for convenience:

```
@fy *= 1.03;
<q> @fy = 3;   //same
```

This will increase the variable `fy` in the Ref databank with 3% over the global sample period.

Examples, array-series

Array-timeseries comply rather tightly with GAMS syntax, to interface more naturally with GAMS files (gdx). But array-timeseries have many other uses, for instance when downloading multi-dimensional data, or reading data from px-files (PC-Axis), cf. the IMPORT statement.

An array-series can be thought of as a super-series, containing sub-series in one or more dimensions, where these sub-series are accessed with (lists of) simple names. For instance, `x` may be a one-dimensional array-series, containing the sub-series `x[a]` and `x[b]`. These sub-series are like any other normal timeseries, just stored inside the array-series. In this sense, `x` can be thought of as a kind of special map, allowing multiple dimensions, and designed for series access. In older versions of Gekko (prior to 2.3.1), such dimensions would typically be handled by means of naming conventions, for instance using normal series `x_a`, and `x_b` instead of `x[a]` and `x[b]`.

You may use single quotes for element access, so `x[a]` = `x['a']`, `x[b]` = `x['b']`, etc. Using quotes is the strict form, and using quotes, the element names may include any characters, for instance `x['ab ? x22']`.

The following is an example of the use of array-series. In the example, `#i` and `#j` are lists of strings containing the sets of names spanning the dimensions, in this case a 3 x 3 structure `[#i, #j]` like this:

```
[a, a]  [a, b]  [a, o]
[b, a]  [b, b]  [b, o]
[o, a]  [o, b]  [o, o]
```

The last part of the example below illustrates how to use default sets (via the map `#default`). In order for default sets to work, the array-series must contain domain information.

```
#i = a, b, o;                             //or: ('a', 'b', 'o')
#j = a, b, o;
#j0 = a, o;
x = series(2);                            //two dimension
p = series(2);                            //two dimensions
x[#i, #j] = 100;                          //all elements = 100
p[#i, #j] = 2;                            //all elements = 2
prt <n> x;                                //prints all elements of
the array-series x, the <n> avoids printing percentage growth
prt <n> p * x;                            //same as p[#i, #j] *
x[#i, #j], simple array-series algebra is possible
prt <n> x[a, #j];                         //or: x['a', #j], prints
the elements with 'a' in the first dimension
prt <n> x[#i, #j];                        //prints all elements,
similar to prt x;
y = sum((#i, #j), x[#i, #j]);             //the sum of all
elements, y = 900
z = sum(#j, x[a, #j] $ (#j in #j0));      //the sum of those #j
that are in #j0 (that is, middle column 'b' is skipped), z = 200
x.setdomains(('#i', '#j'));               //domains set, necessary
for #default logic
p.setdomains(('#i', '#j'));               //domains set, necessary
for #default logic
#default = map();                         //#default is a map type
#default.#j = #j0;                        //chose elements of #j to
print
prt x;                                    //will omit printing
middle column 'b' in set #j
prt <split> x;                            //splits the output
prt x.getelements();                      //elements belonging to
each of the 2 dimensions
prt x.subseries('elements');              //element combinations
for each of the 9 sub-series
```

The summing up with sum() is sometimes called a 'roll-up operation', aggregating rows/columns, whereas for instance `x[a, #j]` would be a so-called 'slice operation'. The in-built functions `getelements()` and `subseries()` can be practical if you need to know more about the elements of the sub-series of a given array-series `x` (for instance, the sub-series `x[a, b]` has element `a` in the first dimension, and element `b` in the second dimension).

You may $-condition on a variable that reflects sparsity in two or more dimensions, too (you may for instance have 193 countries, where only a subset of all combinations trade with each other). The following example illustrates two-dimensional sparsity:

```
x = series(2);
x[a, 10] = 100; x[a, 11] = 101; x[b, 10] = 102; x[b, 11] = 103;
e = series(2);
e[a, 10] = 0; e[a, 11] = 1; e[b, 10] = 1; e[b, 11] = 0; //sparsity
#e = (('a', '11'), ('b', '10')); //sparsity
#i = ('a', 'b');
#j = ('10', '11');
y1 = sum((#i, #j) $ (e[#i, #j]), x[#i, #j]); //203
y2 = sum((#i, #j) $ (#e[#i, #j]), x[#i, #j]); //203
```

In `y1`, the condition is on the array-series `e`, whereas in `y2` it is on the nested list `#e` (in GAMS, a nested list would correspond to a multidimensional set). You may alternatively move the dollar condition to the right, for instance `y1 = sum((#i, #j), x[#i, #j] $ (e[#i, #j]));`. Note that array-series are a bit more lenient regarding the user of integers, which are automatically interpreted as their corresponding strings. Also beware regarding the array-series example that there are settings that makes it possible to omit stating the `0` combinations for `e`, if these are numerous (`option series array calc missing = zero`), cf. this page.

Regarding domains, it is easy to remove a single element from a list of strings with for instance `#i.remove(%s)`, where `%s` is a string. To remove several elements from a list of strings, you may use `#i - #j`. Hence these $-conditionals can be used for easy removal/skipping of elements:

```
z = sum(#j, x['a', #j] $ (#j in #j.remove('b')));
z = sum(#j, x['a', #j] $ (#j in #j - ('b', 'o')));
```

To print an array-series `x1`, use either:

```
disp x1;                                         //shows info
regarding dimensions, elements, etc.
disp x1[a, b];                                   //shows info for
the the sub-series
prt x1;                                          //will print out
all elements
prt x1[#i, #j];                                  //prints out the
elements in lists #i and #j (combined)
```

If you need non-existing array-timeseries elements to be implicitly understood as having value 0, you can use `option series array calc missing = zero;`. In that case, you may use for instance `sum(#j, x1[#i, #j])`, even if some of the combinations (sub-series) of `#i` and `#j` do not exist in `x1`.

In general, you may print or plot an array-series without indicating the dimensions. You can assign lists to array-series dimensions and afterwards control which elements are printed/plotted via a special map with the name `#default`. This can be practical if you typically only want to see some of the elements of an array-series, but not all.

```
#s = ('e1', 'e2');
a = series(1);                  //array-series with 1 dimension
a[#s] = 100;                    //sets a[e1] = a[e2] = 100
a.setdomains(('#s',));          //assigns #s to dimension 1 of the
array-series
p <n> a;                        //prints a[e1] and a[e2]
#default = map();               //defines map #default and puts it
in the global databank
#default.#s = ('e1',);          //or in one line: #default = (#s =
('e1',))
p <n> a;                        //now, because of #default, only
a[e1] is printed
```

It can often be practical to put the `#default` map into the Global databank (that is: `global:#default = ...`), so that it is generally available irrespective of potential OPEN or READ statements. The `#default` map shown above will restrict all printing/plotting of array-series that have `#s` assigned to a dimension as its domain.

If you need to put "normal" timeseries into an array-series or vice versa (or if you are interfacing with multidimensional GAMS/gdx data), you can use the following example:

```
reset;
x1 = 1; x2 = 2;            //create some data here
index * to #m;             //put all series names into the list #m
x = series(1);             //define an array-series with 1 dimension
x[#m] = {#m};              //put the normal series into the array-
series
prt x;                     //print the 2 elements of x, try also
"disp x;"
delete x1, x2, #m;         //delete everything except x
#m = x.getelements()[1];   //get the elements of dimension #1 of x
{#m} = x[#m];              //recreate the normal series from the
array-series
prt x1, x2;                //print the normal series
```

Alias names

It is possible to assign one variable name to another via a special list with the name `#alias`. This can be practical if, for instance, the users are used to one kind of variable names, but are for instance using a model with another kind of variable names.

```
option interface alias = yes;              //this option must be set
global:#alias = #(listfile alias);         //reads alias.lst from
file
c = series(1); c[a] = 100; c[b] = 200;
y = 300;
prt x1, x2,  x3;
```

The `#alias` list could look like the following file:

```
--------------- alias.lst --------------------
x1; c[a]
x2; c[b]
x3; y
----------------------------------------------
```

This file is read as a list of lists, equivalent to `#alias = (('x1', 'c[s]'), ('x2', 'c[b]'), ('x3', 'y'));`. The print prints out `x1`, `x2`, and `x3` as 100, 200, and 300, respectively, even though the 'real' values are stored inside `c[a]`, `c[b]`, and `y`.

Details, x = x[-1] + ... type, and <dyn> option

In Gekko 3.0, series operations are handled more vector-like than in Gekko 2.x. This affects the use of lags in expressions with lagged dependent/endogenous variable:

```
time 2021 2024;
x = 100;
time 2022 2024;
x = x[-1] + 1;  //ERROR: will fail with a dynamics error
x <dyn = no> = x[-1] + 1;  //result: 101, 101, 101
x <dyn> = x[-1] + 1;  //result: 101, 102, 103
```

The last statement is probably easy enough to understand, but the second-last may seem strange. See much more on this, including illustrations, on the dynamic statements page. There are three easy ways of getting accumulations to work without using the `<dyn>` tag:

```
time 2021 2024;
x = 100;
time 2022 2024;
x ^= 1;      //result: 101, 102, 103
x <d>= 1;    //result: 101, 102, 103
dif(x) = 1; //result: 101, 102, 103
```

Still, in some cases it my be practical to be able to use the formulation `x = x[-1] + 1` and have it accumulate, for instance if the equation originates from a model equation. For such cases, either use the local option `<dyn>`, or alternatively use a BLOCK like `block series dyn = yes; ... ; end;`. Example:

```
time 2021 2024;
x = 100;
time 2022 2024;
x <dyn> = x[-1] + 1;      //result: 101, 102, 103
block series dyn = yes;
  x = x[-1] + 1;          //result: 101, 102, 103
end;
```

To guard against errors of this type where the user forgets to use `<dyn>` or a `block series dyn` on an expression like `x = x[-1] + 1`, from version 3.1.7 and onwards, Gekko will check this for the presence of lagged dependent/endogenous variables on the right-hand side and issue a 'dynamics error' if an expression like `x = x[-1] + 1` or similar is used without considering the dynamics question. As stated above, there is much more info on this on the dynamic statements page.

Note

See the page with syntax diagrams if the basics of names, expressions, etc. is confusing.

In addition to operators `+=` and `*=`, you can also use their inverse counterparts: `-=` and `/=`. So `x -= 2;` is the same as `x = x - 2;`, and `x /= 2;` is the same as `x = x / 2;`. This is standard in most computer languages. But please note that `x ^= 2;` is *not* the same as `x = x ^ 2;`, that is, x in the second power.

If any of the right-hand side variables are not found (searching depends upon mode), the statement will exit with an error, unless you set `option series array calc missing = ...;` or `option series normal calc missing = ...;`. If some of the variables have missing values (shown as 'M' when printing), the left-hand side will become missing as well (for the periods affected).

You may use m() to indicate a missing value, for instance `y = m();`.

Regarding $-conditions to the right of `=`, these always evaluate to 0 if the condition is false. On the other hand, regarding a $-condition to the left of `=`, the whole statement is skipped if the condition is false. Note however that such skipping does not apply intra-series regarding $ to the left of `=`. So in a statement like `y $ (x > 100) = 2 * x;` the y series will attain the value 0 in the periods where `x <= 100`, and the value `2 * x` otherwise. If you need `y` to remain untouched when `x <= 100`, you may instead use `y = iif(x, '>', 100, 2 * x, y);`, cf. the iif() function.

Regarding variable types and the Gekko type system, see the VAR section. In this appendix, variable assignment rules, including variable types, is explained in more detail.

A Gekko array-series can be though of as a nested dictionary/map, cf. this:

- Gekko: `m = series(2); m[a, x] = 1; m[a, y] = 2; m[b, x] = 3; m[b, y] = 4; prt m[a, x];`
- R: nested named list, `m <- list(a = list(x = 1, y = 2), b = list(x = 3, y = 4)); print(m$a$x); print(m[['a']][['x']]);`. The elements here are numbers and not sub-series, but the idea is similar.
- Python: nested dict.

Related options

OPTION freq = a; [a|q|m|d|u]
OPTION databank create auto = no; [yes|no]
OPTION series array calc missing = error; [error|m|zero]
BLOCK series dyn = no; [yes|no]. This option can only be set using a block
OPTION series dyn check = yes; [yes|no]
OPTION series normal calc missing = error; [error|m|zero]

Related functions

For array-series: getdomains(), subseries().

Related statements

CREATE, DOC, PRT, VAL, EXPORT<gcm>, EXPORT<flat>

4.3.77 SHEET

You can transfer timeseries or matrices to Excel by means of the SHEET statement. SHEET has the same syntax as the PRT, PLOT and CLIP statements, including the use of operators.

Guide: Excel printing
For an easier introductory guide on Excel printing, see this page.

You may also use SHEET to import data from individual cells via SHEET<import>. The sheet cells can be converted to timeseries, but can alternatively be loaded as a list, map or matrix for further processing (see examples).

Per default, SHEET uses an internal 'engine' to read and write Excel files. This engine does not depend upon Excel being installed. In order to read the older .xls format, you may use OPTION sheet engine = excel (cf. OPTION).

Excel note: if you encounter "dates" with integer numbers larger than 20000, this may be because Excel shows the dates as numbers rather than dates. You may try to change the format of the date cells: right-click, "Format cells", "Date".

For export of timeseries, SHEET uses the same internal component as PRT, so regarding operators and other details, also see the PRT help page.

Syntax

```
sheet < period IMPORT  operator   TITLE=...  STAMP=...   SHEET=...
 CELL=...  DATES=...  NAMES=...   COLORS=...  ROWS  COLS  APPEND=...
 LIST  MAP  MATRIX  MISSING  COLLAPSE=...  DATEFORMAT=...  DATETYPE=...
BANK=...  REF=...  MISSING=...  > variables  FILE=...  ;
```

period	(Optional). Local period, for instance `2010 2020`, `2010q1 2020q4` or `%per1 %per2+1`.
IMPORT	(Optional). Use `sheet<import>` to import matrix-like data from `xlsx`, `csv` or `prn` files, cf. example at the end of this help topic (if you need to import data that is arranged as timeseries in labelled rows/columns, see IMPORT). You may indicate the file type like for instance `sheet<import matrix csv>` (default is `xlsx`). The `prn` type is essentially a file with blank-separated numbers.

	SERIES (default): in this case, the variables must be a comma-separated list (or a list like `{#m}`). With this option, you may use `SHEET= (select the sheet)`, `CELL= (point to the starting cell)`, `ROWS`/`COLS` (select the orientation of the data/timeseries), and `FILE` (the filename). After this, Gekko will read the data into the variables. If the orientation is row-wise (which is default), Gekko will use the *n* x *k* cells starting at the CELL location, where *n* is the number of variables, and *k* is the number of time periods. See example under 'Examples' below. **LIST, MAP or MATRIX**: You can state one collection name like for instance `#m`. Additionally, you may use `SHEET=...`, `CELL=...`, `ROWS`/`COLS` (select the orientation of the data, `COLS` is transposed), and `FILE` (the filename). After this, Gekko will read the data into the given collection: • List: The data is loaded as a list of rows, where each row is a sub-list of elements representing the columns. The cells can be of any type, including null (empty). For instance, `#m[2][3]` will represent row 2, column 3 of the sheet (that is, the cell `C2`). See example under 'Examples' below. • Map: The data is loaded as a map, where the keys represent the cells. The cells can be of any type, including null (empty). For instance, `#m['%c2']` or `#m.%c2` will represent the cell `C2` (stored as the scalar `%c2`). • Matrix: The data is loaded as a matrix, where all the cells must be of value type. If you use the `MISSING` option, any empty cells will be filled with missing values (M), otherwise they are filled with 0's. See example under 'Examples' below. • List and map: note that string cells will be stripped (that is, blanks at beginning and end are removed). • List and map are [New in 3.0.6].
operator	(Optional). 'Long': `abs`, `dif`, `pch`, `gdif`, or 'short': `n`, `d`, `p`, `dp`, `m`, `q`, `mp`, `l`, `dl`, `r`, `rd`, `rp`, `rdp`, `rl`, `rdl`, `a`, `ad`, `an`, `ap`, `adp`, `al`, `adl`, `i`. Read more about operators under PRT.
TITLE	(Optional). A title for the sheet. You can use HEADING as alias.

STAMP	(Optional). If 'yes', a time stamp is inserted at the top.
variables	Variable(s) or expressions(s) to be printed (several variables can be printed at once using, var1, var2 ....). A matrix or matrix expression may be stated, too (only one at a time).
FILE	(Optional). SHEET will optionally create an Excel file silently without opening Excel (the filename will be [filename].xls or [filename]xlsx, and is put into the Gekko working folder) Filenames may contain an absolute path like `c:\projects\gekko\bank.gbk`, or a relative path `\gekko\bank.gbk`. Filenames containing blanks and special characters should be put inside quotes. Regarding reading of files, files in libraries can be referred to with colon (for instance `lib1:bank.gbk`), and "zip paths" are allowed too (for instance `c:\projects\data.zip\bank.gbk`). See more on filenames here.
SHEET	The name of the sheet for your data, for instance `'Data1'`. Gekcel: not supported, instead select the sheet with VBA code like this: `Worksheets("Data1").Activate`.
CELL	The cell where data starts, for instance `'C4'`, default is `'A1'`. Gekcel: not supported, instead use VBA to offset cells.
DATES	[yes|no]: If `yes`, dates are shown (is `yes` per default)
NAMES	[yes|no]: If `yes`, names are shown (is `yes` per default)
COLORS	[yes|no]: If `yes`, colors are shown (is `yes` per default). Gekcel: not supported.
ROWS	[yes|no]: If `yes`, the timeseries are printed in rows (default), use the COLS option to transpose.
COLS	[yes|no]: If `yes`, the timeseries are printed in columns (transposed).

APPEND	[yes|no|ifexist]: If `yes`, the table is appended to an existing Excel workbook. If `no`, a new workbook is always created (default). If `ifexist`, the table is appended to an existing Excel Workbook if the file exists, otherwise a new workbook is created. Gekcel: not supported. [`ifexist` is new in Gekko 3.1.9].
MATRIX	[yes|no]: Used with SHEET<import> to import a matrix, see example below. Gekcel: not supported.
MISSING	[yes|no]: Used with SHEET<import matrix>. Cells with no content are set to missing instead of 0.
COLLAPSE=	(Optional). This option will collapse quarterly or monthly data into annual averages or totals. Use `sheet<collapse>`, `sheet<collapse=avg>` or `sheet<collapse=total>`. You may set the collapse globally, cf. `option sheet collapse = [avg|total|none];`. Beware: this only applies for `option sheet freq = pretty;` (which is not default).
DATEFORMAT= DATETYPE=	(Optional). These options control the date format for .xlsx and .csv files, and for the use of SHEET in Gekcel. `DATEFORMAT` can be either `'gekko'` (default, for instance `2020q3` or `2020m11`) or a format string like `'yyyy-mm-dd'`, and the latter may contain a `first` or `last` indicator, for instance `'yyyy-mm-dd last'`, which indicates for quarterly, monthly or weekly data that the *last* day of the quarter/month/week is used. `DATETYPE` can be either `'text'` or `'excel'`. In the former case, the dates are understood as text strings (for instance `'2020q3'` or `'2020-09-30'` for a quarterly date), and in the latter case (not relevant for .csv files), the date is understood as an Excel date, which basically counts the days since January 1, 1900. This number would correspond to 44104 for the date 2020-09-31, and can be shown in Excel in different ways depending upon date format settings, language settings, etc., but the internal number itself is unambiguous. When using SHEET in Gekcel, using `DATETYPE='excel'` is advised for consistency. [New in 3.0.5].
BANK	(Optional). A bankname where variables are looked up. For instance `sheet <bank = b1> x;` is equivalent to `sheet b1:x;`. See also `<REF = ...>`.

	These options can be convenient instead of opening and closing banks.
REF	(Optional). A bankname where reference variables are looked up. For instance `prt <bank = b1 ref = b2 m> x;`" uses banks `b1` and `b2` for the multiplier. See also `<BANK = ...>`. These options can be convenient instead of opening and losing banks.
MISSING=	(Optional). With `<missing = ignore>`, SHEET will deal with missing array sub-series and missing data values like GAMS, treating them as zero for sums and mathematical expressions, or skipping the printing of a sub-series if it does not exist. The following options are set locally and reverted afterwards: `option series array print missing = skip; option series array calc missing = zero; option series data missing = zero`. See also the appendix page on missings.

- If no period is given inside the <...> angle brackets, the global period is used (cf. TIME).
- If a variable without databank indication is not found in the first-position databank, Gekko will look for it in other open databanks if databank search is active (cf. MODE).

You may use a 'operator' to indicate which kind of data transformation you would like on your variables, for instance `sheet<d>`, `sheet<q>`, `sheet<pch>`. As in the PRT statement, you may also use element-specific operators (for instance `sheet unemp, gdp<p>;`). See the PRT statement regarding the use of operators.

Examples

An example could be:

```
sheet x1, x2;
```

Shows the two variables in Excel (if Excel is installed). Or more advanced:

```
sheet x1 'GDP', x2 'Unemployment' FILE = scenarioA;
```

This produces the file scenarioA.xlsx silently. SHEET produces a table in Excel, with variables running downwards and periods running rightwards. Missing values are converted to missing values in Excel (#N/A). SHEET should work regardless of Excel macro settings, decimal separator etc., on Excel 2003 and upwards.

To illustrate the options, consider this example:

```
sheet <m  SHEET='Raw'  CELL='d1'  DATES=no  NAMES=no COLORS=no
COLS=yes  APPEND=yes> fm/fy 'Imports', fe/fy 'Exports'
FILE=adam.xlsx ;
```

This will print an absolute multiplier (operator m) of the two expressions (with labels), to the Excel workbook adam.xlsx (appending to the pre-existing file). The data will be put into the sheet Raw, in cell D1, without dates, labels and colors, and with the data running downwards in columns.

To illustrate how to transfer raw cell data in an out of Excel, consider this example:

```
reset; time 2001 2002;
xx1 = 1001, 1002;
xx3 = 3001, 3002;
sheet <2001 2002 sheet='test' cell='C5' dates=no names=no
colors=no> xx1, xx3 file=testing;
reset; time 2001 2002;
sheet <2001 2002 import sheet='test' cell='C5'> xx1, xx3
file=testing.xlsx;
prt xx1, xx3;
```

The first SHEET statement will produce the file testing.xlsx, with the sheet test inside, where the data is starting at the cell C5. Note that you need DATES=no and NAMES=no to only get the raw data. The data looks like this (starting at cell C5):

```
1001    1002
3001    3002
```

The last SHEET statement imports data from the sheet test from testing.xlsx, and puts the cells back into the variables (timeseries) xx1 and xx3. To import the Excel data from the previous example into a matrix instead, you may use this:

```
sheet <import matrix sheet='test' cell='C5'> #m file=testing.xlsx;
prt #m;
```

You may use SHEET to export a matrix or matrix expression to Excel, for instance sheet <title='Shares' sheet='2020'> #m/1000 file=m.xlx;. Alternatively, but with less options, you can use EXPORT: export <xlsx> #m file = m.xlsx;.

If you need to perform custom transformations of an Excel spreadsheet, you may load the cells as a list or map, for further processing. Consider this spreadsheet (`data.xlsx`, download here).

Name	Share	jan-20	feb-20	mar-20
Total	100	100.4	100.4	100.4
Total imports	45	104.2	103.1	101.4
011 Meat	10	100.6	99.9	101.7
022 Milk	5	97.4	100.1	98.9
112 Beverages	20	100.2	100.1	100.3
121 Tobacco	10	101.0	97.6	97.2
Total exports	55	99.6	99.6	99.5
011 Meat	20	99.7	98.6	94.6
022 Milk	10	100.0	98.9	99.2
112 Beverages	15	101.0	101.1	101.1
121 Tobacco	10	100.4	101.5	101.3

The dates shown are Excel dates, representing January, February, and March 2020, respectively (Excel stores these internally as the numbers 43831, 43862, and 43891 = days since January 1, 1900). If the frequency is set to monthly, Gekko will convert these Excel dates to the Gekko dates `2020m1`, `2020m2`, and `2020m3`. We wish to extract the rows with three-digit codes as timeseries with suitable names, for instance "011 Meat" should becomes `pm011` if found under imports, else `pe011`, and in addition we with to extract the fixed shares as for instance `s_pm011`.

In the above sheet, all rows have the same number of columns, but in contrast to a matrix, this is not guaranteed. The following program loops through the rows and extracts the data (including series labels):

```
option freq m; //so that Excel dates are read as months
time 2020 2020;
sheet<import list> #m file=data.xlsx;
%rows = #m.length(); //number of rows
%cols = #m[1].length(); //number of cols
%t1 = date(#m[1][3]); //start date
%t2 = date(#m[1][%cols]); //end date
%ie = 1; //imports or exports
for val %i = 1 to %rows;  //loop the rows
  if(#m[%i][1].index('total exports') == 1); %ie = 2;
end;  //exports type
  if(#m[%i][1].length() >= 3); //name with three chars or more
    %code = #m[%i][1][1..3]; //%code = first three characters
    if(%code.isnumeric() == 1); //if these chars are digits
      #numbers = #m[%i][3..]; //fetch the row cells into a list
      %namestart = 'pm';
      %label = 'Imports, ';
      if(%ie == 2);
        %namestart = 'pe';
        %label = 'Exports, ';
      end;
      %name = %namestart + %code; //name like 'pm011'
      {%name} <%t1 %t2 label = %label + #m[%i][1]> =
#numbers; //put the data into a series pm011 = ...
      s_{%name} = timeless(#m[%i][2]); //shares, using timeless
series
    end;
  end;
end;

disp pm011;
```

Instead of `#m[%i][%j]`, you may alternatively use `#m[%i, %j]`, but beware that a range like `#m[%i1..%i2, %j]` is not the same as `#m[%i1..%i2][%j]`, cf. the explanations here. A map containing the cells could also have been used, but in this case, a nested list is easier. Instead of timeless series like `s_pm011`, values like `%s_pm011` could have been used. Result:

```

                        2020m1          2020m2          2020m3
pm011                 100.5700         99.9485        101.7121
pm022                  97.3908        100.1147         98.9306
pm112                 100.2375        100.1348        100.2966
pm121                 100.9756         97.6316         97.1969
pe011                  99.7357         98.5820         94.6168
pe022                  99.9656         98.8769         99.1864
pe112                 100.9782        101.1213        101.1087
pe121                 100.4187        101.4881        101.3085
s_pm011                10.0000         10.0000         10.0000
s_pm022                 5.0000          5.0000          5.0000
s_pm112                20.0000         20.0000         20.0000
s_pm121                10.0000         10.0000         10.0000
s_pe011                20.0000         20.0000         20.0000
s_pe022                10.0000         10.0000         10.0000
s_pe112                15.0000         15.0000         15.0000
s_pe121                10.0000         10.0000         10.0000
```

Note

The EXPORT (or WRITE) statements can also output series as an Excel workbook, but cannot append to an existing spreadsheet. SHEET, however, has more options to control the workbook. Gekko will produce a macro/vba-enabled spreadsheet, if the file extension is .xlsm.

In Excel 2007 and newer, you can click on a cell inside the table and select 'Insert' and 'Line' from the 'Charts' ribbon, and a chart will be produced (with the correct legend, labels etc.).

Related options

OPTION print index date = '';
OPTION print index value = 100;
OPTION sheet collapse = none; [avg|total|none]; //show aggregates for quarters and months
OPTION sheet cols = no;
OPTION sheet engine = internal; //use 'excel' for the older .xls format
OPTION sheet freq = simple; [pretty|simple]; //for quarters and months
OPTION sheet mulprt abs = yes;
OPTION sheet mulprt gdif = no;
OPTION sheet mulprt lev = no;
OPTION sheet mulprt pch = no;
OPTION sheet mulprt v = no;
OPTION sheet prt abs = yes;
OPTION sheet prt dif = no;
OPTION sheet prt gdif = no;
OPTION sheet prt pch = no;
OPTION sheet rows = yes;
OPTION interface excel modernlook = yes; [yes|no]
OPTION interface excel language = danish; [danish|[empty]]

Related statements

CLIP, PRT, PLOT, EXPORT

4.3.78 SIGN

This statement prints out information regarding the model signature in the model file (.frm), and the 'true' hash code corresponding to the model file (and whether they are identical). You can use the SIGN statement to obtain the hash code for signing a new (or changed) model. (If the model is unsigned, click the link 'more' to obtain a comment line with signature that can be copy-pasted into the .frm file.)

The hashcode is a kind of check-sum or fingerprint regarding .frm files. The signatures are technically so-called MD5 hashes, and can be put into .frm file as commentaries (for example: `// Signature: fp88RzyZfJNaoTi3I4X3Ww`). This string of 22 characters and digits (note: the hash code is case-sensitive!) identifies a specific model file, so altering the model file will result in a different hash code. The motivation behind the signatures is two-fold: (a) To be able to make sure that an official model version has not accidently been changed, and (b) The signatures are used to identify models for caching (faster loading). When calculating the signature (hash code), empty lines and comment lines are ignored (except for comments containing model block identifiers `###`), so you may insert empty lines or comments any way you like in the .frm file and preserve the signature (any variable list after the `VARLIST;` or `VARLIST$` tag will be ignored in the hashcode, too). But changing the equations (FRML) in any way will result in a new hash code. The hash code is technically 128 bits, and this means that the probability of two different model files having the same hash code is 2^(-128) = 2.9E-39 (that is, effectively zero).

Syntax

sign;

Examples

To obtain a signature for a (unsigned) model loaded with the MODEL statement, type:

```
sign;
```

You will get an output similar to this:

```
No signature was found in model file (more)
 - Signature in model file       : [not found]
 - True model file hash code     : fp88RzyZfJNaoTi3I4X3Ww
```

Try clicking the 'more' link to obtain a line similar to this:

```
// Signature: fp88RzyZfJNaoTi3I4X3Ww
```

This line can be copy-pasted into the .frm file (typically at the top), which signs the mode. After this, you will be told that the model signature is OK when loading the (unaltered) model with the MODEL statement.

Note

You may put other meta-information into the model file (.frm). As of now, `Info`, `Date`, `Freq` and `Signature` fields are supported. For instance:

```
    // Info: Model used for forecasting 2012-2030
    // Date: 7-11-2012 15:37:00
    // Freq: q
    // Signature: fp88RzyZfJNaoTi3I4X3Ww
```

Gekko will complain if this format deviates, for instance the `Info` field is to be written with capital 'I', with no blank before the colon, and one blank after the colon. This rigorousness regarding form is to make it easy to spot the information in different .frm files. The `Info` and `Date` fields will be displayed when loading the model (MODEL statement).

Related statements

MODEL, SIM

4.3.79 SIM

SIM solves the model (cf. MODEL) dynamically, i.e. lagged endogenous variables use simulated values. The simulated values are placed in the first-position bank, thereby overwriting previous values of the endogenous variables. The default solving method is Gauss-Seidel. If Gauss-Seidel poses problems, for instance because of bad values in some of the variables, you may try setting `option solve failsafe = yes;`. This will pinpoint the exact time period and variable that first produces an invalid value. Failsafe has a small speed overhead. For harder problems, you may need to use the Newton method.

You may solve goals/means by indicating these with the ENDO/EXO statements. The Newton method is used in that case, but please note that you need to use `sim<fix>` in order for such goals/means to bind. When goals are removed later on (UNFIX statement), the solve method reverts to Gauss-Seidel.

SIM may also solve a model with leaded endogenous variables. The Fair-Taylor ('fair') method is used in that case, or you may use the more powerful Newton-Fair-Taylor (called 'nfair' in Gekko). See the `option solve forward method = ...`.

Gekko will try to calculate some reasonable starting values for the endogenous variables, by means of looking at lagged values regarding these. If you wish to use the current (non-lagged) values as starting values, you should use `option solve data init = no;`. For static models (for instance CGE models) that are simulated over a single time period, `init = no` should be used.

Syntax

sim < period FIX STATIC RES AFTER >;

period	(Optional). Local period, for instance `2010 2020`, `2010q1 2020q4` or `%per1 %per2+1`.
AFTER	`sim<after>` calculates three kinds of variables: (a) those designated with a formula beginning with 'Y', (b) all auto-generated variables of J- and Z-type, and (c) all table variables (variables defined after the `AFTER;` or `AFTER$` line in a model file). Since Y-type variables are typically also J- and Z-variables, one can think of `sim<after>` as a way to compute all reversed J- and Z-variables independently of the SIM statement (the SIM statement computes these after simulation), in addition to computing table variables. See the MODEL statement for more on J- and Z-variables.

	On a well-specified model, reading a historical databank, and issuing a `sim<after>` statement on the historical period should ideally make it possible to simulate on the historical period and replicate the endogenous variables.
FIX	Tells Gekko to enforce any goals/means stated by the ENDO/EXO statements. It is mandatory to use `sim<fix>` in such cases (if a normal SIM is used, the goals/means are ignored).
STATIC	Changes in endogenous variables are not transferred from period to period via lags. See also the `option solve static = yes`.
RES	The `sim<res>` statement (residual check) performs a one-step-ahead static single-equation simulation of the entire model, i.e. the result of one equation does not affect other equations, nor does results of previous periods affect following periods. For **Gekko models**, the left-hand side variables are computed. The difference between actual historical values and a static single equation solutions are the equation residuals. Residuals are used to measure how well the model equations forecasts historically. `sim<res>` is widely used for testing historical data against the model, or for testing a model against a historical databank. Gekko offers functionality to print results to files for subsequent inspection. See the menu item 'Utilities' --> 'Check residuals...' (output is grouped according to formula codes and/or model sections, and can be ordered). For **GAMS models**, there is no strict sense of left-hand and right-hand variables. Therefore, series/array-series are computed with names corresponding to the equation names. You may add a prefix to these names with `<prefix = ...>`.
PREFIX =	(Optional). For a GAMS scalar model, when using for instance `sim<res prefix = 'res_'>` will add `res_` to the equation names when calculating the residuals. So if an equation name is for instance `E_exports(agr, 2027)`, corresponding to the year 2027, the array-series `res_E_exports[agr]` will contain that residual (in the year 2027).

- If no period is given inside the <...> angle brackets, the global period is used (cf. TIME).

Examples

To increase all exogenous prices by 1% in 2011-2020, and inspect the effects on private consumption (`pcp`) and wages (`lna`):

```
read lang11;
#pm = pm01, pm2, pm3r, pm59, pm7b, pm7y, pms, pmt;
#pe = pee2, pee59, peet, peesq;
time 2011 2020;
{#pm} *= 1.01;
{#pe} *= 1.01;
sim;
mulprt pcp, lna;
```

The result shows the difference between the values after simulation (first-position bank) and values in the reference databank (`lang11.gbk`) for the variables `pcp` and `lna`. Here, it is assumed that `lang11` already is simulated over the period 2011-20.

If you need to solve goals/means, you can use the ENDO and EXO statements to change status of exogenous and endogenous variables, and perform a SIM afterwards. In that case, the solve method will be automatically changed to the Newton algorithm (and changed back if the goals/means are removed).

There are many options regarding SIM: please see under OPTION (in the `option solve ...` section). Or type `option solve ?;`. A single-equation static simulation can be performed with the `sim<res>` statement (for instance to check historical residuals).

If you want to change to the Newton algorithm manually, you can use `option solve method = newton;`. It typically runs a bit slower, but is generally much more robust (and precise). With `option newton robust = yes`, the Newton method activates a remedy against illegal starting values, like the logarithm to a negative number etc.

Static simulation (one period does not affect the next period) can be obtained by setting `option static = yes`.

Related options

See the OPTION help page for details.

OPTION solve data create auto = yes; [yes|no]
OPTION solve data ignoremissing = no; [yes|no]
OPTION solve data init = yes; [yes|no]
OPTION solve data init growth = yes; [yes|no]
OPTION solve data init growth min = -0.02;
OPTION solve data init growth max = 0.06;
OPTION solve failsafe = no; [yes|no]
OPTION solve forward dump = no; [yes|no]
OPTION solve forward fair conv = conv1; [conv1|conv2]

```
OPTION solve forward fair conv1 rel = 0.0001;
OPTION solve forward fair conv1 abs = 0.0001;
OPTION solve forward fair conv2 trel = 0.0001;
OPTION solve forward fair conv2 tabs = 1.0;
OPTION solve forward fair damp = 0.0;
OPTION solve forward method = fair; [fair| nfair | none]
OPTION solve forward fair itermax = 200;
OPTION solve forward fair itermin = 10;
OPTION solve forward method = fair; [fair | nfair];
OPTION solve forward nfair conv = conv1; [conv1|conv2]
OPTION solve forward nfair conv1 rel = 0.001;
OPTION solve forward nfair conv1 abs = 0.001;
OPTION solve forward nfair conv2 trel = 0.001;
OPTION solve forward nfair conv2 tabs = 1.0;
OPTION solve forward nfair damp = 0.0;
OPTION solve forward nfair itermax = 200;
OPTION solve forward nfair itermin = 0;
OPTION solve forward nfair updatefreq = 200;
OPTION solve forward stacked horizon = 5;
OPTION solve forward terminal  = const; [const|growth|none]
OPTION solve forward terminal feed = internal; [internal|external]
OPTION solve gauss conv = conv1;
OPTION solve gauss conv1 rel = 0.0001;
OPTION solve gauss conv1 abs = 0.0001;
OPTION solve gauss conv2 trel = 0.0001;
OPTION solve gauss conv2 tabs = 1.0;
OPTION solve gauss conv ignorevars = yes; [yes|no]
OPTION solve gauss damp = 0.5;
OPTION solve gauss dump = no; [yes|no]
OPTION solve gauss itermax = 200;
OPTION solve gauss itermin = 10;
OPTION solve gauss reorder = no; [yes|no]
OPTION solve method = gauss; [gauss|newton]
OPTION solve newton backtrack = yes; [yes|no]
OPTION solve newton conv abs = 0.0001;
OPTION solve newton invert = lu; [lu|iter]
OPTION solve newton itermax = 200;
OPTION solve newton robust = yes; [yes|no]
OPTION solve newton updatefreq = 15;
OPTION solve print details = no; [yes|no]
OPTION solve print iter = no; [yes|no]
OPTION solve static = no; [yes|no]
```

Related statements

MODEL, EXO, ENDO, READ, WRITE, CLONE, MULPRT

4.3.80 SMOOTH

SMOOTH replaces missing values ("holes") inside a timeseries with values generated by means of a particular (user-chosen) method. The statement ignores the global time period, and a local time period cannot be set. Only holes *between* non-holes are smoothed, so the statement only deals with holes that occur after the timeseries' first non-missing value and before the timeseries' last non-missing value (use the `fromSeries()` function with `'dataStart'`/`'dataEnd'` arguments if you need to know start/end dates of a timeseries).

Instead of the SMOOTH statement, you may alternatively use the similar `smooth()` function, for instance `plot x.smooth();` (see under functions).

Syntax

```
smooth var1  = var2  type;
```

var1	The new corrected variable
var2	The variable that contains missings/holes
type	The type of smoothing, choose between: • LINEAR. Linear interpolation (default). • GEOMETRIC. Geometric interpolation. • REPEAT. Repeats last known observation. • SPLINE. Cubic splines. • OVERLAY. Insert another series into the holes. Note: default is LINEAR. You can alter the default with `option smooth method = ... ;` (cf. OPTION). Note that in Gekko 2.x, SPLINE was used as default.

The methods are as follows:

LINEAR	Use linear interpolation (adds a fixed amount for each period in the hole(s)).
GEOMETRIC	Use geometric interpolation (multiplies with a fixed amount for each period in the hole(s)).
REPEAT	Set the values to the last known value before the hole(s).

SPLINE	Uses cubic splines to fill the hole(s).
OVERLAY	Uses another timeseries to fill the hole(s).

- If no period is given inside the <...> angle brackets, the global period is used (cf. TIME).
- If a variable on the right-hand side of `=` is stated without databank, Gekko may look for it in the list of open databanks (if databank search is active, cf. MODE).
- Looping: with a list like for instance `#m = a, b;`, you may use `for string %i = #m; smooth y{%i} = x{%i}; end;` to smooth `xa` into `ya`, and `xb` into `yb`.

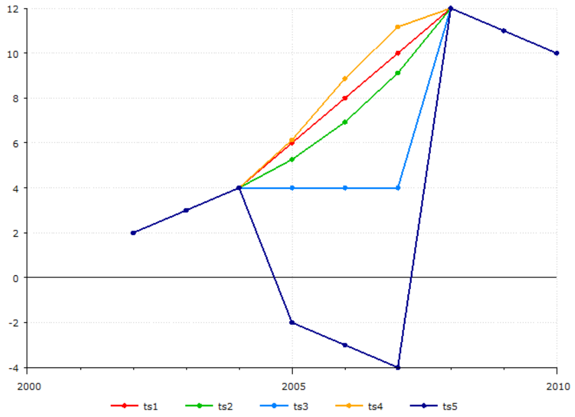
Example

For instance:

```
reset;
time 2002 2010;
ts <2002 2004> = 2, 3, 4;
ts <2008 2010> = 12, 11, 10;
tsb <2004 2008> = -1, -2, -3, -4, -5;
smooth ts1 = ts LINEAR;
smooth ts2 = ts GEOMETRIC;
smooth ts3 = ts REPEAT;
smooth ts4 = ts SPLINE;
smooth ts5 = ts OVERLAY tsb;
prt <n> ts1, ts2, ts3, ts4, ts5;  //or use plot instead of prt
```

As you can see, the timeseries `ts` has a hole in the middle, namely the observations 2005-2007 (inclusive). Using the SMOOTH statement, these three observations are filled out. Below, the four different interpolation methods are shown (interpolated values in red):

```
                   ts            ts1            ts2            ts3
           ts4            ts5
  2002         2.0000         2.0000         2.0000         2.0000
        2.0000         2.0000
  2003         3.0000         3.0000         3.0000         3.0000
        3.0000         3.0000
  2004         4.0000         4.0000         4.0000         4.0000
        4.0000         4.0000
  2005              M         6.0000         5.2643         4.0000
        6.1349        -2.0000
  2006              M         8.0000         6.9282         4.0000
        8.8696        -3.0000
  2007              M        10.0000         9.1180         4.0000
       11.1694        -4.0000
  2008        12.0000        12.0000        12.0000        12.0000
       12.0000        12.0000
  2009          11.0000         11.0000         11.0000         11.0000
        11.0000         11.0000
  2010          10.0000         10.0000         10.0000         10.0000
        10.0000         10.0000
```



Regarding the GEOMETRIC (green) method, note that the growth rate of `ts2` is constant (31.61%) in the three interpolated years. In this particular case, SPLINE (yellow) perhaps provides the most realistic hole-filling, since it takes the curvature of the `ts` timeseries into consideration.

Note

See the hpfilter() function regarding the smoothing of timeseries without holes.

Related options

OPTION smooth method = linear; [linear|geometric|repeat|spline|overlay]

Related statements

SPLICE

4.3.81 SPLICE

SPLICE is used to combine two or more timeseries into one. You may choose which one of these timeseries is "primary" (keeps its values): default is the last timeseries stated. The SPLICE statement splices all available observations, and hence is not called with a time period. If you need to truncate the period of the resulting timeseries, use a subsequent TRUNCATE, or alternatively use the in-built function `splice()`.

You may splice on a single period or a time interval. If no time period(s) is provided, all overlapping observations are used for splicing.

Syntax

```
splice <FIRST LAST N=... TYPE=...> ... ;
splice y =  x1          x2           x3 ... ;
splice y =  x1   t1     x2   t2      x3 ... ;
splice y =  x1   t1 t2  x2   t3 t4   x3 ... ;
```

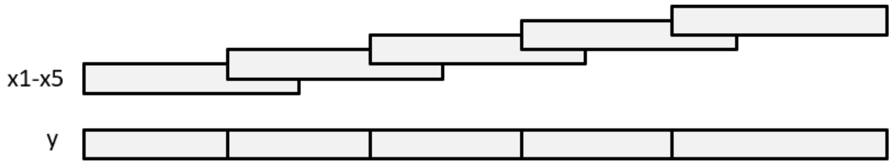
y	The new timeseries made from the variables on the right-hand side of `=`.
x1, x2, ...	The timeseries that are to be spliced into the left-hand side timeseries.
t1, t2, ...	(Optional). Time periods. If on the right-hand side two timeseries are separated by one date, this date is used as the splicing date. If there are two dates, the indicated time interval is used for splicing. A statement like `splice y = x1 2010q2 x2;` is essentially short-hand for `splice y = x1 2010q2 2010q2 x2;`. If no time period is stated between two of the right-hand side series, Gekko tries to splice using as many common observations as possible.
FIRST	(Optional). Uses the first timeseries as "primary" (keeps its values). Using `<first>` is equivalent to `<n=1>`. Using the last timeseries is default.
LAST	(Optional). Since `<last>` is already default, this option only has cosmetic use.

N=	(Optional). Sets the timeseries number (1, 2, ... , n) to use as "primary" (keeps its values). Using `<n=1>` is equivalent to using `<first>`. Using the last timeseries is default.
OBS=	(Optional). Use `<obs=n>` where `n` is an integer >= 0 to limit the length of the overlap (`n` == 0 is interpreted as infinite: that is, no effect). For instance, `<obs=4>` would mean that Gekko uses **at most** 4 overlapping observations when splicing. Think of it as removing observations/data in the non-primary series until the overlap has the desired max size. This can also be set as a global option with `option splice obs = ... ;`.
TYPE=	(Optional). Choose type of smoothing: `rel1`, `rel2`, `rel2` or `abs`. Below the types are described (example with three overlapping observations): • **rel1**: (Default). Ratio of averages used for a relative adjustment. The formula used for the relative correction factor `R = ((y1 + y2 + y3)/3) / ((x1 + x2 + x3)/3)`. • **rel2**: Average ratio used for a relative adjustment. The formula used for the relative correction factor is `R = (y1/x1 + y2/x2 + y3/x3)/3`. • **rel3**: All values are first transformed with `log()`, then a SPLICE with `<type=abs>` is computed, and finally all values are transformed back with `exp()`. Because of the `log()`, negative values will not work. The type is quite similar to `rel2`, except in essence using geometric rather than arithmetic averages. • **abs**: Average difference is used for an absolute adjustment. This is advised for timeseries that can be 0 or negative, and where the levels of the timeseries are comparable. The formula used for the absolute correction factor is `D = (y1 + y2 + y3)/3 - (x1 + x2 + x3)/3`. Note: if you are using just one overlapping period, `rel1`, `rel2` and `rel3` are equivalent.

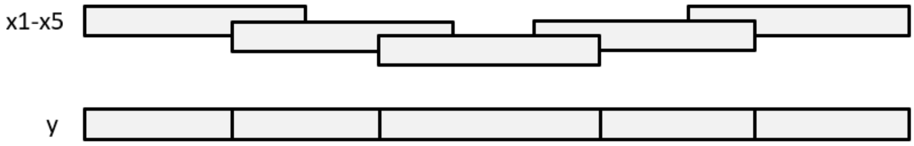
- If a variable on the right-hand side of `=` is stated without databank, Gekko may look for it in the list of open databanks (if databank search is active, cf. MODE).
- Looping: with a list like for instance `#m = a, b;`, you may use `for string %i = #m; splice y{%i} = x1{%i} x2{%i}; end;` to splice `x1a` and `x2a` into `ya`, and `x1b` and `x2b` into `yb`.

Methodology

Consider five overlapping timeseries `x1`-`x5`, as shown below. If you use `splice y = x1 x2 x3 x4 x5;`, Gekko will per default first use the levels of `x5` (the last series) as "primary" `y` values. It will then move to the left to `x4` and "glue" these values on `y` (using the overlapping observations to adjust). This is repeated left-wise for `x3`, `x2`, and finally `x1`. The resulting `y` series is shown, too (note how the internal period delimiters in `y` reflect start periods of `x2`-`x5`).



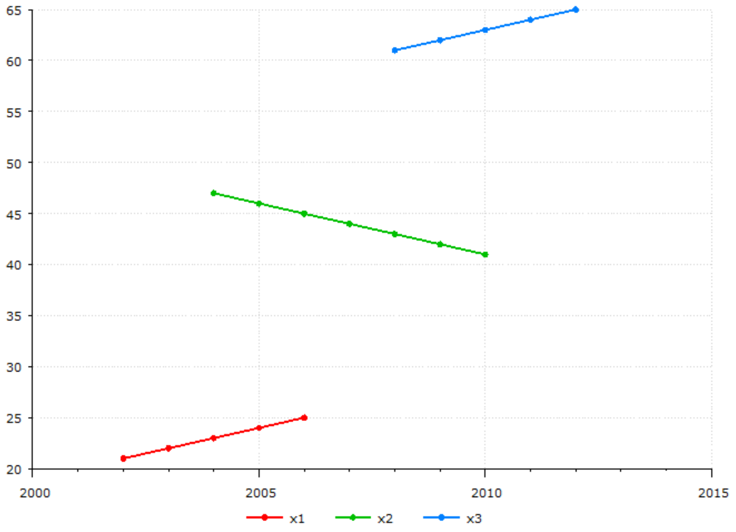
If, instead, `splice <n=3> y = x1 x2 x3 x4 x5;` is used, `x3` is used as "primary" `y` values. Then left-wise from `x3`, first `x2` and then `x1` are glued on. Finally, right-wise from `x3`, first `x4` and then `x5` are glued on.



Note that the internal period delimiters in `y` are now different (and the levels will usually be so, too). The delimiters now reflect `x2` and `x3` start periods on the left, and `x3` and `x4` end periods on the right.

Example

Consider these three overlapping timeseries `x1`, `x2` and `x3`.



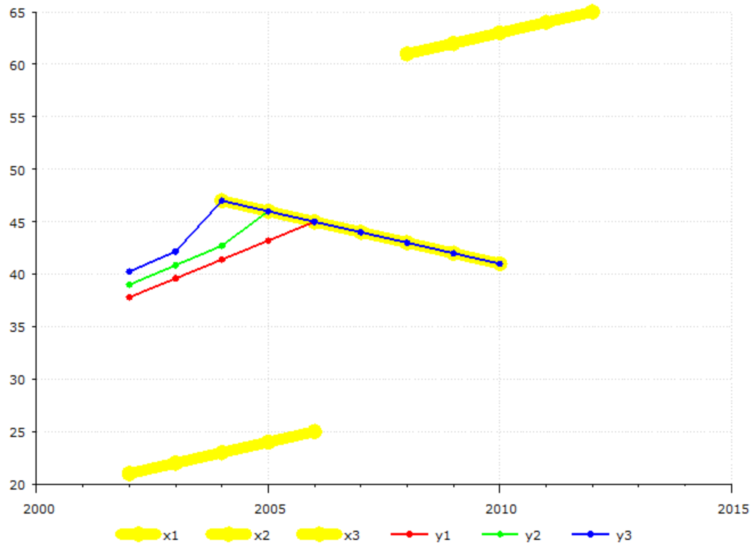
These series are very dissimilar and not exactly obvious candidates for splicing, but we will use them for illustrative purposes, cf. the following example:

```
reset;
time 2002 2012;
x1 <2002 2006> = 21, 22, 23, 24, 25;
x2 <2004 2010> = 41, 42, 43, 44, 45, 46, 47;
x3 <2008 2012> = 61, 62, 63, 64, 65;
splice          y1 = x1 2006      x2;
splice          y2 = x1 2005 2006 x2;
splice          y3 = x1           x2;    //all common observations
splice          y4 = x1           x2 x3; //multi-splice
splice <n=2>    y5 = x1           x2 x3; //second series is
"primary"
splice <first> y6 = x1            x2 x3; //first series is "primary"
prt <n> y1, y2, y3, x1, x2;
prt <n> y4, y5, y6, x1, x2, x3;
```

For `y1`, the year 2006 is used as splice year, whereas for `y2`, the period 2005-2006 is used. For `y3`, all common observations are used (2004-2006). Default is that the levels of the last series are kept.

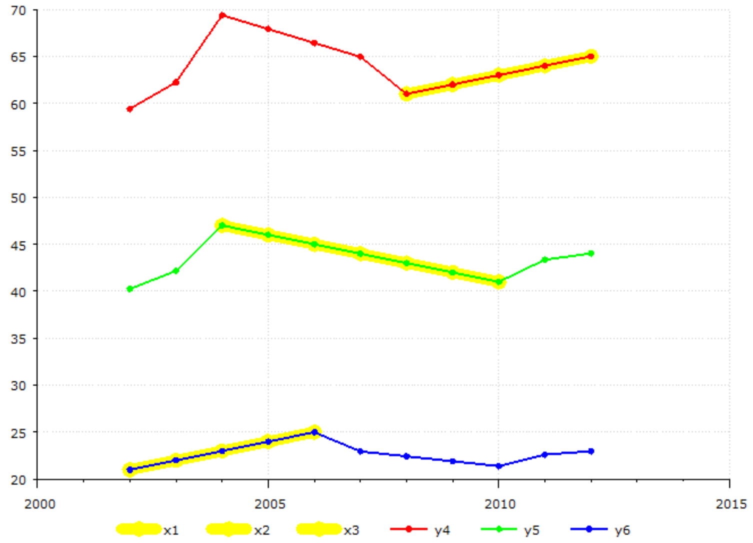
Levels:

```
                  y1              y2              y3              x1
          x2
 2002         36.1200         36.4286         36.7500         21.0000
            M
 2003         37.8400         38.1633         38.5000         22.0000
            M
 2004         39.5600         39.8980         41.0000         23.0000
     41.0000
 2005         41.2800         42.0000         42.0000         24.0000
     42.0000
 2006         43.0000         43.0000         43.0000         25.0000
     43.0000
 2007         44.0000         44.0000         44.0000               M
     44.0000
 2008         45.0000         45.0000         45.0000               M
     45.0000
 2009         46.0000         46.0000         46.0000               M
     46.0000
 2010         47.0000         47.0000         47.0000               M
     47.0000
```



The above figure illustrates how the attachment is done (remember that `y1` splices on 2006, `y2` splices on 2005-2006, and `y3` splices on 2004-2006). For `y1`, `x1` attaches cleanly to `x2` in 2006. For `y2` and `y3`, the attachement is done over a period range (2005-2006 and 2004-2006, respectively). For instance, in 2002 and 2003, the blue `y3` series is lower than it would be if `y3` was attached on the year 2004 only (and this is because `x1` is growing and `x2` is decreasing).

The multi-spliced series `y4`, `y5` and `y6` look like this:



Again, there are some "funny" movements right before/after the attachment years, which has to do with overlapping observations being used and the fact the the `x1`-`x3` series are so dissimilar. Also note that relative splicing is used here, not absolute.

Note

You may alternatively use the `splice()` function, for instance `y1 = splice(x1, 2006, x2);` But beware that this function respects the local/global time period when putting values into `y1`, so the `splice()` values will not be put into `y1` outside of the local/global time period. But apart from this truncating of the function output values, the function works just like SPLICE.

Related options

OPTION splice obs = 0;

Related statements

SMOOTH

4.3.82 STOP

The STOP statement will abort from all called program files/procedures/functions, without executing anything more after the STOP (this is not the case regarding RETURN). Therefore, STOP can be practical for debugging, making it possible to inspect variables at that particular point of execution. After a STOP, Gekko will show the "call stack", showing where the STOP statement was encountered.

To stop/abort a program while it is running, you may use the red stop button in the user interface.

Syntax

```
stop ;
```

Example

To see how STOP works, try creating the two program files `file1.gcm` and `file2.gcm` as shown below. Then call the first one of these from the statement prompt with `RUN file1;`. After this, Gekko will report that the program has been stopped in line 2 of `file2.gcm`, and that `file2.gcm` is called from `file1.gcm`. Next, if you issue a `prt %i;`, the value will be 1001, because the third lines of `file1.gcm` and `file2.gcm` are never executed. With a RETURN instead of a STOP in `file2.gcm`, the value of `%i` would be 1101, because the third line of `file1.gcm` would still have been executed.

```
------------- file1.gcm --------------
%i = 1000;
RUN file2;
%i = %i + 100;
--------------------------------------

------------- file2.gcm --------------
%i = %i + 1;
stop;
%i = %i + 10;
--------------------------------------

RUN file1;
prt %i;
```

Note

- The similar RETURN statement does not return from all program files/procedures/functions.
- The EXIT statement effectively issues a STOP, and then afterwards closes the Gekko application.

Related statements

RETURN, EXIT

4.3.83 STRING

The STRING statement is used to assign a string to a scalar variable of string type. String names always start with the symbol `%`, like the other scalar types val and date. Using the STRING keyword is no longer mandatory in Gekko 3.0.

A string can be used to refer to a variable via the {}-curlies. For instance, if `%s = 'b2:x!q'`, the expression `{%s}` will refer to the variable `b2:x!q` (a quarterly series with name `x`, taken from the `b2` bank). See also the syntax diagrams.

Strings are often used as list elements, for instance `#m = ('a', 'b', 'c')`. In that case, `prt #m;` will print these three strings, whereas `prt {#m};` will print the series `a`, `b`, and `c`.

When inserting or concatenating strings, it is best practice to use so-called string interpolation, using {}-curlies. If `%a = 'cat'` and `%c = 'black'`, it is more readable to use `'The {%a} is {%c}'` than `'The ' + %a + ' is ' + %b`. This also complies with name-composition, and hence a composed name like `a{%b}c` is easy to transform into a string, just add quotes: `'a{%b}c'`.

Syntax

```
%s = expression;
string %s = expression;
string ?; //print string scalars
```

It is no longer legal to use for instance `string s = 'abc';`, omitting the '%'. Also, using `{i}` as short for `{%i}` is no longer legal either.

Strings can be added together (concatenated) with the `+` operator.

A string (or list of strings) representing variable names may be manipulated by means of Gekko's inbuilt functions to handle these. Variable names here include bank, frequency, indexes, etc., and examples of such functions could be setBank(), removeBank(), replaceBank(), setFreq(), removeFreq(), setNamePrefix(), etc. There are many more of such functions, see the functions section, under ‘Bank/name/frequency/index manipulations’.

For instance, if you have a list `#m = ('x', 'y');`, you may use `prt {#m};` to print out `x` and `y`, `prt {#m.setBank('b')}`; to print out `b:x` and `b:y`, or `prt {#m.setFreq('q')};` to print out `x!q` and `y!q` (here, `prt b:{#m};` and `prt {#m}!q;` will work, too).

There are quite a few string functions, where strings can be combined in different fashions.

String combining functions

Function name	*Description*	*Examples*

[x]-index	Index: returns the character at position x.
	Returns: string
		`%s = 'abcd';`
		`prt %s[2];  //'b'`

[x1..x2]-index	Index: returns the range of characters from position x1 to x2 (both inclusive). You may omit x1 or x2.
	Returns: string
		`%s = 'abcd';`
		`prt %s[2..3];  //'bc'`

concat(s1, s2)	Appends the two strings: same as s1 + s2.
	Returns: string
		`%s = concat('He', 'llo');`
		`Result: 'Hello'.`

endswith(s1, s2)	Returns 1 if the string s1 starts with the string s2, else 0. The comparison is case-insensitive.
	Returns: val
		`%v = endswith('abcde', 'cde');`
		`Returns: 1`

includes(s1, s2)	Returns 1 if the string s2 is found inside (as a part of) string 1, else 0 is returned. The search is case-insensitive. Note that the `contains()` function is not useful regarding such in-string searches.
	Returns: val
	[New in 3.1.17].
		`%s = 'abCd';`
		`prt %s.includes('bc');  //1`

index(s1, s2)	Searches for the first occurrence of string s2 in string s1 and returns the position. It returns 0 if the string is not found. The search is case-insensitive
	Returns: val
		`%v = index('onetwothreetwo', 'two');`
		`Returns: 4.`
		`%v = index('oneTWO', 'two');`
		`Returns: 4.`

isAlpha(s)	Returns 1 if all the characters are letters (alphabet). [New in 3.0.5].	`%v = isAlpha('aBc');`
		`Returns: 1`

isLower(s)	Returns 1 if the string contains no uppercase characters. [New in 3.0.5].	`%v = isLower('abc12');`
		`Returns: 1`

isNumeric(s)	Returns 1 if all the characters are of numeric value. [New in 3.0.5].	`%v = isNumeric('123');`
		`Returns: 1`

isUpper(s)	Returns 1 if the string contains no lowercase characters. [New in 3.0.5].	`%v = isUpper('ABC12');`
		`Returns: 1`

length(s)	The length of the string (number of characters). You may use len() instead of length(). Returns: val	`%v = %s.length();`

lower(s)	The string in lower-case letters.
	Returns: string
		`%s = lower('aBcD');`
		`Result: 'abcd'.`

nl()	A system newline/linebreak. Computerwise, `nl()` is equal to the C# string `"\n"`, but beware that you cannot use the Gekko string `'\n'` instead of `nl()`, because the former is equal to the C# string `"\\n"`.
	Returns: string
		`%s = 'a' + nl() + 'b'; //string with newline`
		`tell %s;`
		`#m = %s.split(nl()); //list with 2 elements`
		`p #m;`

prefix(s1, s2)	If s1 is a string, it has the string s2 prefixed (prepended). Returns: string	`%s1 = %s2.prefix('a');`

replace(s1, s2, s3)
replace(s1, s2, s3, max)
	In the string s1, the function replaces all occurrences of s2 with s3. Replacement is case-insensitive.
	
	If max > 0, the replacement is performed at most max times.
		`%s = replace(%s1, %s2);`
		`//or: replace(%s, %s1, %s2)`

	Returns: string	

split(s1, s2)
split(s1, s2, removeempty)
split(s1, s2, removeempty, strip)
	Splits the string s1 by means of the delimiter s2. Empty elements are removed per default, and the resulting strings are stripped (blanks are removed from the start and end of the strings). The last two options are `1` and `1`, if omitted. [New in 3.0.6]	`%s = 'a, b,c,,d, , e';`
		`#m1 = %s.split(',');`
		`//--> ('a', 'b', 'c', 'd', 'e')`
		`#m2 = %s.split(',', 1, 1);`
		`//--> ('a', 'b', 'c', 'd', 'e');`
		`#m3 = %s.split(',', 0, 1);');`
		
		`//--> ('a', 'b', 'c', '', 'd', '', 'e')`
		`#m4 = %s.split(',', 1, 0);');`
		
		`//--> ('a', ' b', 'c',  'd', ' ', ' e')`
		`#m5 = %s.split(',', 0, 0);');`
		
		`//--> ('a', ' b', 'c', '', 'd', ' ', ' e')`
		
		`%s = 'a' + nl() + 'b'; //string with newline`
		`#m = %s.split(nl(), 0); //--> ('a', 'b')`
		`//Note: the 0 argument keeps empty lines`

startswith(s1, s2)	Returns 1 if the string s1 starts with the string s2, else 0. The comparison is case-insensitive.
	Returns: val
		`%s = 'abcde';`
		`%v = %s.startswith('abc');`
		`Returns: 1`

strip(s)	Removes blank characters from the start and end of the string. Returns: string	`%s1 = %s2.strip();  //or: strip(%s1)`
stripstart(s)	Removes blank characters from the start of the string. Returns: string	`%s1 = %s2.striptart();  //or: stripstart(%s1, %s2)`
stripend(s)	Removes blank characters from the end of the string. Returns: string	`%s1 = %s2.stripend();  //or: stripend(%s1, %s2)`

substring(s, start,	The piece of the string between character number	`%s = %s1.substring(3, 2);`
		`//or: substring(%s1, 3, 2)`

length)	start and length (these must be integer values).
	You can alternatively use a 'slice', using []-notation, see example.
	Returns: string
		`%s = %s1[3 .. 5];`
		`//a slice from pos 3 to 5 (both inclusive)`

suffix(s1, s2)	If s1 is a string, it has the string s2 suffixed (appended) Returns: string	`%s1 = %s2.suffix('a');`
upper(s)	The string with upper-case letters. Returns: string	`%s = upper('aBcD'); Result: 'ABCD'.`

In addition, there are some functions that fetch different kinds of meta-data (as strings) from the system or databanks, see under functions.

Examples

Concatenation:

```
%s1 = 'ab';
%s2 = 'cd';
%s3 = %s1 + %s2;     //result: 'abcd'
```

You may wish to use strings to control file names.

```
%path = 'folderA' ;
%bank = 'prognosis' ;
read c:\{%path}\{%bank};
```

Gekko supports automatic in-substitution of any expression inside {}-curlies. For instance:

```
%s1 = 'b';
%s2 = 'a' + %s1 + 'c';    //result: 'abc'
%s3 = 'a{%s1}c';          //result: 'abc', easier to type and read
```

In general, in such cases, it is better and more readable to use {}-curlies, instead of concatenating with `+`. Using {}-curlies both for name-composition (like `a{%s1}b`) and string substitution (like `'a{%s1}b'`) makes it easy to move composed names in and out of strings, because the syntax is the same.

The tilde (`~`) can be used to avoid in-substitution of {}-curlies. Tilde can also be used to allow single quotes inside a string:

```
%s1 = 'blue';
%s2 = 'the' + %s1 + 'car';   //result: 'the blue car'
%s3 = 'the {%s1} car';       //result: 'the blue car'
%s4 = 'the ~{%s1} car';      //result: 'the {%s1} car'
%s5 = 'the ~'blue~' car';    //result: 'the 'blue' car'
```

You may put any valid Gekko expression inside the {}-braces, as long as it can be evaluated to a string:

```
time 2020 2022;
x = 55000000;
%date = 2021;
%s1 = 'value in {%date} is {x[%date]/1e6}
M';                     //result: 'value in 2021 is 55 M'
%s2 = 'value in ' + %date + ' is ' + x[%date]/1e6 + '
M';         //same, but more cumbersome to write and read
```

Strings are often used in loops, to loop over variables, see FOR.

If you need to convert a string to a date or value, you must convert it explicitly with the date() and val() conversion functions, for instance:

```
%s1 = '2010';
%s2 = '123.45';
%d = date(%s1);
%v = val(%s1);
```

Note: whole lists of strings can be converted into lists of dates or values with the dates() and vals() functions.

TELL examples

Below are some examples regarding the use of strings in the TELL statement (TELL prints the string on the screen):

```
%s1 = 'Value in ';
%d1 = 2010;
%s2 = ' is: ';
%v1 = 113.45;
```

```
%v2 = 10;
%s3 = %s1 + %d1 + %s2 + (%v1 + %v2);
tell %s3;
tell 'Value in ' + %d1 + ' is: ' + (%v1 + %v2);
tell 'Value in {%d1} is: {%v1 + %v2}';
```

This will print `Value in 2010 is: 123.45` three times, so the three last TELLs are equivalent. When adding the scalars, it should be noted that scalar dates and scalar values are automatically converted to strings when added to a string with the `+` operator. So there is no need to use `%s3 = %s1 + string(%d1) + %s2 + string(%v1);`.

If you need to write curly braces like `{%d1}` literally, preprend the symbol `~` to indicate that Gekko should not try to in-substitute. For example:

```
tell 'The scalar name is ~{%d1}';
```

This will print 'The scalar name is {%d1}' on the screen, and not try to evaluate the inside of the {}-curlies. If you need to format values, you can either use the `format()` function, or use global formatting of {}-curlies via `option string interpolate format val = ...` . For instance. See more regarding format() function in the Gekko functions section (the code '6:0.00' means a 6 character wide field, where the number has exactly two decimals).

```
%v11 = 1/3; %v12 = 1/4; %v21 = 1/5; %v22 = 1/6;
tell '{format(%v11, '6:0.00')},{format(%v12, '6:0.00')}';
tell '{format(%v21, '6:0.00')},{format(%v22, '6:0.00')}';
//since the formatting is the same, you can use an option:
option string interpolate format val = '6:0.00';
tell '{%v11},{%v12}';
tell '{%v21},{%v22}';

//   result:
//   0.33,  0.25
//   0.20,  0.17
```

Note

See the page with syntax diagrams if the basics of names, expressions, etc. is confusing.

If you need to convert a VAL or DATE scalar to a STRING type, use the string() conversion function.

Gekko 3.0 will no longer in-substitute scalars outside {}-curlies, for instance the string `'name is %name'` will not work as intended (use `'name is {%name}'`).

Also, in Gekko 3.0 you can no longer use `{s}` instead of `{%s}` to refer to the name corresponding to a scalar string. Obviously, using for instance `x{i}{j}` instead of `x{%i}{%j}` inside a loop is easier on the eyes, but there are several drawbacks. See the end of this page regarding the drawbacks.

See also the format() function and OPTION string interpolate format val = ... ; regarding {...}-formatting of values inside strings.

Regarding variable types and the Gekko type system, see the VAR section. In this appendix, variable assignment rules, including variable types, is explained in more detail.

Related options

OPTION string interpolate format val = "";

Related statements

DATE, VAL

4.3.84 SYS

The SYS statement provides access to the system shell (in Windows sometimes called the DOS prompt). Gekko will wait until the SYS statement finishes before it continues.

Syntax

```
sys <MUTE> statements  WORKING = foldername;
```

MUTE	(Optional). With this option set, the system shell runs silently in Gekko. Alternatively, the system shell output (and error messages) is shown in the Gekko main window.
WORKING	(Optional). If set, the Windows system will use this folder as "starting directory" for the system process. For example, `SYS 'dir' working = c:\tools;` will show the contents of the `c:\tools` folder. If WORKING is not indicated, the Gekko working folder is used. [New in 3.1.16].
statements	A string with windows system shell commands. You may use SYS without arguments. In that case, the system shell opens up as a separate window.

Example

In general, the SYS statement returns the output of the system shell, for instance:

```
sys 'dir';  //shows the list of files in the working folder
```

You can use the SYS statement for file managing:

```
sys 'copy file1.txt file2.txt';
```

or for more complex tasks or jobs

```
sys 'start /wait gnuplot p1.prg';
```

If you need to insert scalar values, just use `{}`-parentheses. For instance:

```
%name1 = 'file1';
%name2 = 'file2';
sys 'copy {%name}.txt {%name2}.txt';
```

To open up the system shell in a separate window for interactive use, call SYS without arguments.

```
sys;
```

Note

Note the use of single quotes when using SYS.

- To call R via SYS, use for instance `sys '""C:\Program Files\R\R-3.6.2\bin\rscript.exe" --no-save "C:\Gekko\Files\test.r""';`
- To call Python vis SYS, use for instance `sys '""c:\Program Files\Python36_64\python.exe" -u "C:\Gekko\Files\test.py""';`

The use of `"`-quotes makes it possible to use blanks etc. in the file paths. When called like this, both R and Python will return screen output to Gekko while the R/Python program runs.

4.3.85 TABLE

The TABLE statement is used to call tables (.gtb files) designed in a special XML format. This format is designed for ease of use regarding most types of tables with the time dimension running outwards, and different variables running downwards.

Guide: tables
For an easier introductory guide on Gekko tables, see this page.

Table output can be either text or html. The latter is default since it often provides more readable output. When a table (text or html) has been printed, you may use the 'Copy' button to copy the cells to the clipboard, for successive pasting into e.g. a spreadsheet like Excel. Copying the table this way includes full precision of all numbers, but formatting will be lost.

For html tables shown in the 'Menu' tab in the Gekko window, the user may click the link 'Transform options' to transform the data (for instance, percentage growth).

Tables can also be called from menus (see MENU).

Note that you may use scalar-variables (for instance STRINGs) inside the xml elements. If you want to insert a scalar into a text field, enclose the scalar in {}-curlies, for instance `<txt>Table regarding the year {%year}</txt>`.

Note: You may use the in-built XML Notepad editor to edit the .gtb file, cf. the XEDIT statement.

Syntax

table < *period* *operator* HTML WINDOW=main > *tablename* ;

period	(Optional). Time period.
operator	(Optional). Can be `m` for multiplier or `r` for Ref databank. The `m` code works as if two tables were subtracted, one made with no operator ('Work'), and one made with r operator (Ref). No other codes are possible at the moment (operator `n` can also be used, but is the same as omitting an operator).
HTML	(Optional). If `html` is indicated, the table will be printed in html format, instead of in text format. It will be shown in the 'Menu' tab (unless the WINDOW=main option is active).

WINDOW =	(Optional). If `WINDOW=main` is indicated, output will be put into the main window, even if the table is of html type. In that case, html codes will be printed on the screen. This can be used together with `pipe<html>`, in order to insert a html table inside a html file. You may use TELL to insert other html lines into the html file.
tablename	Name of the table to be printed (extension .gtb can be omitted)

- If no period is given inside the <...> angle brackets, the global period is used (cf. TIME).

The Gekko table system can be used in two ways: either via a simple XML syntax to describe the appearance of the tables, or via a more matrix-like syntax where the tables' elements are assigned one by one (like filling in a matrix). For most tables with time running horizontally and variables vertically, the XML syntax is much easier to use.

Example:

```
Table printed: 30-04-2018 14:53:30
-------------------------------------------------------------------
---------------------
| Table 1. Supply balance, Million DKK, current prices
                    |
--------------------
+------------------------------------------------------------------
-
|                   |      2005       2006       2007       2008
    2009       2010 |
--------------------
+------------------------------------------------------------------
-
| GDP             Y |   1554520    1605530    1676300    1734800
 1796410    1849360 |
| Imports         M |    674748     716544     748044     771898
  802542     806695 |
| Total (Y+M)   Yst |   2229270    2322080    2424340    2506700
 2598950    2656050 |
--------------------
+------------------------------------------------------------------
-
```

In this table (shown in text format), there is a column with descriptions ('GDP' etc.), followed by a column with variable names ('Y' etc.). The table can easily be constructed by means of XML syntax. The XML tables use syntax that resemble the way tables are made in HTML syntax, specifically the use of the `colspan` attribute to merge cells horizontally. Data-columns (for instance the period 2005-10 above) are conceptually treated as a single (expandable) column, and formatting can be set on many levels.

To start out with a concrete example, the above table was constructed by means of the following code:

```
<?xml version="1.0" encoding="Windows-1252"?>
<gekkotable>
  <tableversion>1.0</tableversion>
  <table varformat="f9.0">
    <cols>
      <colborder/>
      <col txtalign="left"/>
      <col txtalign="right"/>
      <colborder/>
      <col type="expand" txtalign="center"/>
      <colborder/>
    </cols>
    <rows>
      <rowborder/>
      <row>
        <txt colspan="3">Table 1. Supply balance, Million DKK,
current prices</txt>
      </row>
      <rowborder/>
      <row>
        <txt/>
        <txt/>
        <date/>
      </row>
      <rowborder/>
      <row>
        <txt>GDP</txt>
        <txt>$</txt>
        <var>Y</var>
      </row>
      <row>
        <txt>Imports</txt>
        <txt>$</txt>
        <var>M</var>
      </row>
      <row>
        <txt>Samlet tilgang (Y+M)</txt>
        <txt>$</txt>
        <var>Yst</var>
      </row>
      <rowborder/>
    </rows>
  </table>
</gekkotable>
```

Inside the `<table>` tag (where the real defining of the table begins), the columns are first defined inside the `<cols>` tag. You may visualize the above table as the following 3-column table:

```
  ==========  | ====== | ========
    COL1      |  COL2  |   COL3
  ==========  | ====== | ========
              |        |
-----------------------------------
| Heading spanning 3 columns....  |
-----------------------+-----------
|                      |  Periods |
-----------------------+-----------
| Description1    var1 |  Data    |
| Description2    var2 |  Data    |
| Description2    var3 |  Data    |
-----------------------+-----------
```

In the XML defintion above, inside the `<cols>` tag, the columns are defined. First there is a `<colborder>`, corresponding to the left table border. Next there is a `<col>` tag, with default text alignment set to `left`, followed by a similar tag with text alignment set to `right`. These correspond to the "description" and "variable name" columns in the table. Next, there is a `<colborder>` to separate descriptions/variable names from the data, followed by a `<col>` of type `expand`. This column is expandable, and expands into as many columns as there are time periods. The `<cols>` definitions end with a `<colborder>`, corresponding to the right table border.

Regarding the `<rows>`, these are given one by one by means of `<row>` tags. Each `<row>` expects 3 items, since there are 3 columns defined in the `<cols>` section. The first item in `<rows>` is a `<rowborder>`, however, so these are added analogously to the `<colborders>` under the `<cols>` tag. The first `<rowborder>` corresponds to the top table border, and the next item is a `<row>` with a `<txt>` item inside. The `<txt>` spans all 3 colums, so the vertical border between column 2 and 3 will be hidden in this row. Next, there is a `<rowboarder>`, and then the date row begins. This row is filled with two empty `<txt>` items (think of these ase placeholders), and then a `<date>` tag corresponding to the third column. This puts dates corresponding to the time period into this section of the table. Afterwards, there is a `<rowborder>`.

Next is variables, and these are given by means of two `<txt>` tags and one `<var>` tag. The `<txt>` tags put text into the description and variable name fields, and the `<var>` tag identifies the variable (or expression) to print out. So we have these tags regarding the first variable row:

```
<txt>GDP</txt>
<txt>$</txt>
<var>Y</var>
```

The `$` is a special code implying that this piece of text is to be taken from the following `<var>` field. The text could alternatively have been given by means of `<txt>Y</txt>`, but this way of doing it would produce a lot of duplicate variable names in the file. To avoid duplication (and reduce the error rate when editing variable names), it is convenient to use the `$`-codes. The two next rows are added in a similar fashion, and finally a `<rowborder>` tag concludes the table.

It should be noted that there are three different item types: `<txt>`, `<var>` and `<date>`. These are formatted differently, and to format them you may use attributes corresponding to the type, for instance `varformat`, `varalign`, or `txtalign`. Such attributes are inherited, and can be put on different levels of the table, more specifically on the following tags (in that order):

- in `<table>` (applies to the whole table).
- in `<col>` (applies to the specific column>.
- in `<rowformat>` (applies to all following rows, until next `<rowformat>` overrides it).
- in `<row>` (applies to the specific row)
- or in the element itself (for instance in the `<txt>` or `<var>` tags).

For instance, in the example there is the following format given: `<table varformat="f9.0">` .This applies to the whole table (since there are no other `varformats` in the table), and it means that variables are to be written with a width of 9 characters, and without any decimals. Lower level formats override higher levels, so for instance a specific row might be formatted differently. For instance `<row varformat="f9.2">` could be used if that particular row were a percent change, where two decimals might be the most desirable format.

Regarding borders, parts of these can be hidden in different ways. The easiest way is to use column spanning, as in the example above. Since the table heading (`"Table 1. Supply balance"`) spans three columns, the inner vertical border after the second column is not shown. To remove all vertical borders for a particular row (or successive rows), you may use the tag `<colborderhide>` inside the `<rows>` tag. This will remove all vertical borders for the following rows, until a `<colbordershow>` is set. To remove only specific borders, you may use `<colboderhide>inner<colborderhide>`, or `<colboderhide>outer<colborderhide>` (this hides inner or outer borders). You may also indicate the borders to remove by means of a comma-separated list of integers: `<colboderhide>1,2<colborderhide>`. This will hide the first and second vertical borders.

List of different tags and attributes

In general, the *tags* (for instance `<txt>`) can be thought of as contaning data, whereas the *attributes* (for instance t`xtalign="left"`) can be thought of as containing formatting. So the tags, or more precisely the *elements* inside the tags (for instance the element `GDP` in `<txt>GDP</txt>`) can be thought of as the content of the table, whereas the attributes describe how the content is formatted/shown.

Tags:

- `<col>`: defines a column: set attribute type=`"expand"` for data columns. You may define a special kind of column showing only one (indicated) period.
- `<colborder>`: defines a vertical border separating columns
- `<colborderhide>`: hides vertical borders, content can be `"inner"`, `"outer"`, or list of integers
- `<colbordershow>`: shows vertical borders (if some of them have been hidden)

- `<cols>`: inside this tag, columns are defined one by one, including their type, and any borders separating them
- `<row>`: indicates a new row, content is defined inside the `<row>`
- `<rowborder>`: indicates a horizontal border
- `<rowformat>`: sets format that is to apply to all rows until next `<rowformat>`
- `<rows>`: inside this tag, rows are defined one by one, including horizontal borders etc.
- `<subcolborder>`: is used to show (grey) borders between columns inside a time period. This tag is set inside a `<col>` tag. Example: `<subcolborder period="%per1"/>` or `<subcolborder period="2030"/>`.

Attributes:

- `colspan` (example: `colspan="2"`), used inside a `<row>` tag to span/merge columns. Similar to the way cells are merged horizontally in HTML tables. Note that code like `<text colspan="3">AAA</txt> <txt>BBB</txt>` would make 'BBB' appear in the fourth column, so the `<text colspan="3">` tag really represents three `<txt>` tags.
- `datealign` (`left`/`middle`/`right`): used inside `<table>`, `<col>`, `<rowformat>`, `<row>`, or `<date>`. Horizontal alignment of dates in a `<date>` tag.
- `period` (example `period="#target_year"`). Used inside a `<col>` tag to indicate that only that particular period is to be shown, regardless of other time settings. Handy for a column with special kinds of indicators etc. The attribute is also used inside `<subcolborder>`, to indicate the period after which a gray line is to be inserted. Example: `<col period="#target_year" txtalign="center" varformat="f8.3"/>`.
- `txtalign` (`left`/`middle`/`right`): used inside `<table>`, `<col>`, `<rowformat>`, `<row>`, or `<txt>`. Horizontal alignment of text in a `<txt>` tag.
- `type` (example: `type="span"`): used inside a `<col>` tag to indicate that it is expandable (for instance data columns). Only columns of `"expand"` type are allowed to auto-expand into more columns, corresponding to the number of time periods the table is called with.
- `vardisplay` (example: `vardisplay="p"`): used inside `<table>`, `<col>`, `<rowformat>`, `<row>`, or `<var>`. Sets operator regarding <var> tag. These should be given as Gekko operators, for instance "p" for percent growth rate, "q" for multiplier percentage difference, "n" for levels etc (cf. the PRT/PLOT/SHEET statements).
- `varformat` (example: `varformat="f12.2"`): used inside `<table>`, `<col>`, `<rowformat>`, `<row>`, or `<var>`. Sets the print format for `<var>` tags. For instance, "f12.2" means floating point, width = 12 characters, and 2 decimals. Integers (no decimals) can be set as for instance "f12.0". If you need to round off to, for instance, the nearest hundreds, you may use negative decimals places, for instance varformat = "f9.-2". A number like 12345 would then be printed as 12300.
- `varscale` (example: `varscale="0.001"`): used inside `<table>`, `<col>`, `<rowformat>`, `<row>`, or `<var>`. All data in the given context is scaled/multiplied by the value given.

Editing the table

It is recommended that you use the in-built XML Notepad editor to edit the XML files, cf. the XEDIT statement (if used, choose 'View' --> 'Expand All' to unfold all XML nodes). You may also use Notepad (cf. the EDIT statement), but it is recommended to use a specific XML editor for editing the tables. Using a simple text editor like Notepad entails some potential problems. There will be no check that the XML syntax is correct, e.g. that a tag like `<row>` is always closed by means of a corresponding `</row>` tag. Also, the XML syntax represents some characters in a special way: notably the `<` , `>`, and `&` characters (these should be written `&lt;`, `&gt;`, and `&amp;`).

If the file is not in valid XML syntax, Gekko will complain that the file is invalid and abort.Internally, when executing a XML table, the XML is translated into "normal" Gekko code (heavily using table objects) which is then executed in the same way as a normal Gekko program file.

Example

Call the table table1.gtb, for the annual period 2012-2016:

```
table <2012 2016> table1;
```

To have it printed in HTML instead, and as multiplier differences (operator `m`), use:

```
table <2012 2016 m html> table1;
```

To edit the .gtb file using Notepad, you may use `edit table1.gtb;`, but it is preferable to use a dedicated xml editor, which can be called with XEDIT:

```
xedit table1.gtb;
```

Note

In order to copy-paste a html table to Excel, the most reliable way is to use the copy-button in the Gekko user interface. There is an option to paste decimal separators as commas instead of periods (`option interface clipboard decimalseparator`). Alternatively, you may right-click the html-table and paste from that menu, but that way you will lose non-shown decimals in the table cells.

In html tables, if you hover over a number, a label will appear with full decimals.

You can use a TIMEFILTER to omit periods for a more readable table.

If a variable is missing, and if `option table ignoremissingvars = yes` (default), the values will be shown as missings ('M'), instead of Gekko reporting an error.

Gekko will look for the table in first the working folder, and then in the folders denoted with `option folder table`, `option folder table1` and `option folder table2` (in that order). If the table is called from a .html menu, the URL path of the table call will be included too (after `option folder table2`). So if preferred for menu systems, you may put your table (.gtb) files next to your menu (.html) files.

The function frombank() can be used to decorate a table with filename/date of the underlying databank.

If you need to insert blank spaces in `<txt>`-fields in a html table (for instance to manually control the width of a column), you can use the special xml-code `&amp;nbsp;` (if written in a text editor like Notepad), for instance three blanks: `<txt>Gdp&amp;nbsp;&amp;nbsp;&amp;nbsp;</txt>`. If the code is written via XEDIT, you should write the code as `&nbsp;`. These special codes will only work for html tables: for txt tables they will show up as noise. For txt tables, you can just use normal blanks, for instance: `<txt>Gdp   </txt>`.

There is an automatic table conversion tool from the older PCIM table format to the new XML format. See the menu: 'Utilities' --> 'Converters' --> 'PCIM converters' --> 'Convert PCIM tables...'.

Related options

OPTION folder table = [empty];
OPTION folder table1 = [empty];
OPTION folder table2 = [empty];
OPTION interface table operators = yes; [yes|no]
OPTION table decimalseparator = period; [period|comma]
OPTION table html font = Arial;
OPTION table html fontsize = 72;
OPTION table html datawidth = 5.5;
OPTION table html firstcolwidth = 5.5;
OPTION table html secondcolwidth = 5.5;
OPTION table html specialminus = no; [yes|no]
OPTION table ignoremissingvars = yes; [yes|no]
OPTION table mdateformat = '';
OPTION table stamp = yes; [yes|no]
OPTION table thousandsseparator = no; [yes|no]
OPTION table type = html; [txt|html]

Related statements

MENU, PLOT, XEDIT, EDIT

4.3.86 TARGET

TARGET is used by the GOTO statement, to transfer execution to the point following right after the label.

Syntax

target name;

The label must be name-like, that is, alphanumeric characters including underscore (and not starting with a digit). You can not use scalars or expressions etc. as labels.

Examples

See the GOTO help file.

Related statements

GOTO

4.3.87 TELL

The TELL statement prints a text string in the output window. TELL will abort with an error if the argument is not a string, which can guard against unintended errors. PRT will also print scalar strings, but TELL is convenient for messages.

Syntax

tell < LINE NOCR MUTE > message ;

message	(Optional). The text to print (remember single quotes). Can be an expression that can be evaluated as a string.
LINE	(Optional). With `<line>`, an empty line is inserted before this message. This is similar to using `tell; tell 'message';`.
NOCR	(Optional). With `<nocr>`, no newline is inserted at the end of the string (no carriage return). This way, you can use several TELL's in succession on the same line.
MUTE	(Optional). Use `<mute=no>` to turn off general mute set with `option interface mute = yes;`. Use of `<mute>` or `<mute=yes>` will mute the message and is essentially like commenting out the TELL statement.

Examples

The TELL statement can be used to print text strings, for instance for messages:

```
tell;  //blank line
tell 'First line';
tell 'Second line';
```

Use `<nocr>` if you need to join text:

```
tell <nocr> 'The ';
tell <nocr> 'fox is ';
tell 'red.';
```

Use `<mute=no>` to overrule a general mute (typically used in program files), for instance:

```
option interface mute = yes;
open mybank;    //normally produces screen output
model mymodel; //normally produces screen output
tell <line mute=no> 'Databank opened and model loaded';
//
//Gekko keeps on muting from here
```

An alternative could be to put the OPEN and MODEL statements inside a BLOCK that turns on muting:

```
block interface mute = yes;
  open mybank;    //normally produces screen output
  model mymodel; //normally produces screen output
end;
tell <line> 'Databank opened and model loaded';
//
//Gekko does not mute from here
```

Below some examples, where scalars are inserted into the TELL statement in different ways:

```
%s1 = 'Value in ';
%d1 = 2010;
%s2 = ' is: ';
%v1 = 113.45;
%v2 = 10;
%s3 = %s1 + %d1 + %s2 + (%v1 + %v2);
tell %s3;
tell 'Value in ' + %d1 + ' is: ' + (%v1 + %v2);
tell 'Value in {%d1} is: {%v1 + %v2}';
```

This will print `Value in 2010 is: 123.45` three times, so the three last TELLs are equivalent. When adding the scalars, it should be noted that scalar dates and scalar values are automatically converted to strings when added to a string with the `+` operator. So there is no need to use `%s3 = %s1 + string(%d1) + %s2 + string(%v1);`.

If you need to write curly braces like `{%d1}` literally, preprend the symbol `~` to indicate that Gekko should not try to in-substitute. For example:

```
tell 'The scalar name is ~{%d1}';
```

This will print 'The scalar name is {%d1}' on the screen, and not try to evaluate the inside of the {}-curlies. If you need to format values, you can either use the `format()` function, or use global formatting of {}-curlies via `option string` `interpolate format val = ...` . For instance. See more regarding format() function in the Gekko functions section (the code '6:0.00' means a 6 character wide field, where the number has exactly two decimals).

```
%v11 = 1/3; %v12 = 1/4; %v21 = 1/5; %v22 = 1/6;
tell '{format(%v11, '6:0.00')},{format(%v12, '6:0.00')}';
tell '{format(%v21, '6:0.00')},{format(%v22, '6:0.00')}';
//since the formatting is the same, you can use an option:
option string interpolate format val = '6:0.00';
tell '{%v11},{%v12}';
tell '{%v21},{%v22}';

//   result:
//   0.33,   0.25
//   0.20,   0.17
```

Related statements

DISP, PRT

4.3.88 TIME

The TIME statement sets global time used for subsequent time series operations etc. The statement works for frequencies annual (`a`), quarterly (`q`), monthly (`m`), weekly (`w`), daily (`d`) and undated (`u`). The TIME statement is the only way to change global time. Dates have formats like `2020`, `2020q3`, `2020m3` or `2020m3d25`. Note that week numbering follows the ISO 8601 standard.

Syntax

```
time periods ;
time period ;
time ? ;
```

periods	*per1 per2*
per1	Start period, for instance 2010 or 2010q1
per2	End period, for instance 2012 or 2012q4
period	A single date that will be used as both start and end period.
?	Shows the current time settings (also shown at the bottom of the main window)

Example

If you only want to consider the year 2010, write:

```
time 2010 2010 ;
time 2010;  //equivalent
```

If you want to use the period 1980 to 2010, write:

```
time 1980 2010;
```

statements like SERIES, PRT etc. respect global time if no time period is indicated in the local <>-option field. In all the relevant statements (i.e. all statements operating on time series), a local time period can be set inside the <> brackets, operating on only the particular statement line.

Global frequency is altered this way (here to quarterly):

```
option freq q;
```

This sets the frequency to quarterly ('m' for monthly, 'a' for annual, 'u' for undated). If quarterly frequency is chosen, time periods are given as:

```
time 2000q2 2005q4;
```

That is, from second quarter of 2000 to fourth quarter of 2005 (both inclusive). Months work similarly, for instance:

```
time 2000m2 2005m12;
```

This syntax regarding quarters and months can be used wherever it is relevant, for instance in the PRT statement:

```
prt <2000m2 2005m12> fy, enl;
```

When the frequency is non-annual, you can still denote the time period by means of integers (i.e. years). In that case, the full span of the sub-period is assumed: for instance with quarterly frequency set, `prt <2010 2012>` would be equivalent to `prt <2010q1 2012q4>`, and `time 2010 2012` would be equivalent to `time 2010q1 2012q4`. Gekko will generally try to convert dates with "foreign" frequency into the current frequency. Note: if the frequency is quarterly, the following TIME statement

```
time 2010;
```

will span 2010q1 to 2010q4, that is, the statement covers all the subperiods.

Note

With annual frequency, if you write a number small enough, Gekko assumes that you intend to add 1900 to the number. So Gekko tranlates 95 into 1995 for instance. Actually, in this way, you may write 125 instead of 2025, although this way of writing is not recommended.

Timeseries may be converted from one frequency to another by means of the COLLAPSE and INTERPOLATE comands.

The TIMEFILTER statement may omit observations when printing etc., but can also be used to average the non-shown periods into the shown periods. This can be practical for timeperiods with many observations.

Local time periods like `prt<2010 2015>x;` do not allow the use of a single period, like `prt<2010>x;`. A series statement can omit the 'SERIES' but keep the local time period, for instance:

```
x <2020m1 2025m12> = 100;  //same as series <2020m1 2025m12> x =
100;
```

Related statements

TIMEFILTER, COLLAPSE, BLOCK, OPTION freq = ...

4.3.89 TIMEFILTER

TIMEFILTER does not work for other frequencies than annual in Gekko 3.0 yet. For PRT of annual series, <filter=avg> does not yet work.

The statement TIMEFILTER is used to indicate periods that are to be omitted in output from TABLE, DISP, PRT and MULPRT. The statements PLOT, SHEET and CLIP are not affected by the filter.

Syntax

timefilter *filterperiods* ;

filterperiods	*periods, periods, ...* (note the comma)
periods	*singleperiod* | *periodlist*
singleperiod	Single observation
periodlist	*singleperiod* `..` singleperiod `BY` *step* (if you prefer, you may use `TO` instead of `..`, and `STEP` instead of `BY`)
step	An optional stepsize (default step: 1). Must be integer >= 1. Omit `BY step` if not needed.

Examples

You may define periods like this:

```
timefilter  2010..2015, 2020..2030 by 5;
```

This results in the following:

```
Chosen periods: 2010, 2011, 2012, 2013, 2014, 2015, 2020, 2025,
2030
Hidden periods: 2016, 2017, 2018, 2019, 2021, 2022, 2023, 2024,
2026, 2027, 2028, 2029
```

So when you use for instance PRT, you will get all periods from 2010 up to and including 2015, and then the rest of the periods up to 2030 only shown every 5 years (2020, 2025 and 2030). A normal print will look like this:

```
prt fY;
```

```
                  fy       [%]
2010   1379471.0000       1.75
2011   1431367.5000       3.76
2012   1463109.6250       2.22
2013   1491416.2500       1.93
2014   1512247.3750       1.40
2015   1534174.6250       1.45
2020   1649068.8750       1.46
2025   1773862.6250       1.47
2030   1908568.7500       1.48
```

whereas an average-print will look like this:

```
prt <filter=avg> fY;
```

```
                       fY           %
2010        1379471.0000       1.75
2011        1431367.5000       3.76
2012        1463109.6250       2.22
2013        1491416.2500       1.93
2014        1512247.3750       1.40
2015        1534174.6250       1.45
2016-2020   1602281.9750       1.45
2021-2025   1723167.3250       1.47
2026-2030   1853869.9250       1.47
```

Here, the skipped periods are averaged into the shown periods. For the absolute level (the `fY` column), a simple average is used, whereas for the percentage column, a more complicated averaging of growth rates is performed, in order to yield consistent average growth rates for the aggregated periods

The filter is controlled via these general options:

- OPTION timefilter type = hide; [hide|avg]
- OPTION timefilter = no;

The last options shows whether filtering is to be applied or not, whereas the first options selects the type of filtering. These can be overridden in the PRT statement, for instance `prt<filter>`, `prt<nofilter>`, `prt<filter=hide>`, or `prt<filter=avg>`, so that different filtering can be performed quite easily in the PRT statement, without having to change the globals options.

Note

TIMEFILTER does not alter the TIME period settings. Note that this statement only affects print layout (so SIM, SERIES etc. are unaffected by TIMEFILTER settings). When simulating for instance, out-filtered periods would never be skipped: filtering only affects visual reporting.

You may use `TO` instead of `..` and `STEP` instead of `BY` when indicating *periods*, so `2010..2020 by 2` and `2010 to 2020 step 2` are equvialent (and the latter is more similar to the FOR loop over DATEs).

Related statements

TIME, PRT

4.3.90 TRACE2

TRACE2 shows data traces of timeseries statements in the data trace viewer. When the syntax and inner workings of data tracing has matured, TRACE2 will become just TRACE. The data trace viewer can also be started from the DISP command (click the link).

Data tracing is the ability to trace timeseries operations backwards in time, also known as *data lineage*. If you consider some output series *z*, produced from a series operation *f*() on an input series *y*, we may think of data tracing as the ability to store in the metadata of *z* how it was produced. Hence, the operation *z* = *f*(*y*) is stored in *z* (including the operation *y* = *g*(*x*), if *y* depends on some other series *x*, etc.) Besides "recording" code like for instance `z = y/4;` or `collapse y = x!q total;`, data tracing also stores information on which file and line the statement occurred in, time period, time stamp, etc. Therefore, in principle, an error in some observation in *z* ought to be traced back to *y* and further back to *x*.

Caveat: As mentioned, data tracing has not fully matured yet, and some teething troubles are to be expected. Data tracing runs independently of the Gekko calculations per se, and in particular there are (at the moment at least) no guarantees if a data trace says that a timeseries is constructed in this or that way over this or that period. Bugs may still be present, so for now use the data traces as an *indication*, showing how a timeseries is most probably constructed, from which `gcm` file, at what date, using which databank file, etc. This indication is very often very useful, but do not treat it as the absolute truth.

NOTE: When running the same series statements again and again iteratively (for instance, iterating over rows and columns in an input-output table until convergence), you may consider switching off data tracing with `option databank trace = no;`. For such solver-like scenarios, data tracing may slow down the calculations needlessly.

Syntax

trace2 ***series ;***

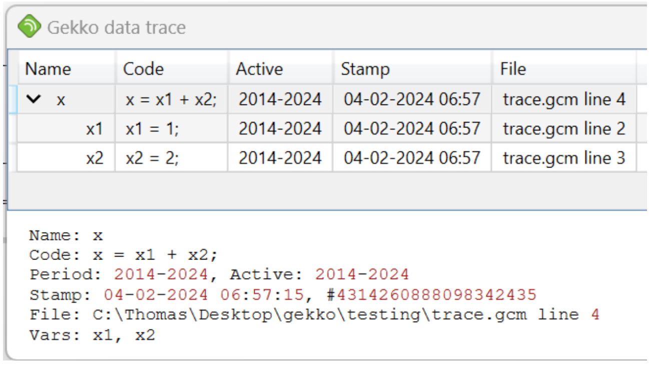
series	A Gekko series or array-series.

- If a variable is stated without databank, the databank is assumed to be the first-position databank.

Examples

Consider this `trace.gcm` program file:

```
reset;
x1 = 1;
x2 = 2;
x = x1 + x2;
disp x;
trace2 x;
```



Name	Code	Active	Stamp	File
x	x = x1 + x2;	2014-2024	04-02-2024 06:57	trace.gcm line 4
x1	x1 = 1;	2014-2024	04-02-2024 06:57	trace.gcm line 2
x2	x2 = 2;	2014-2024	04-02-2024 06:57	trace.gcm line 3

```
Name: x
Code: x = x1 + x2;
Period: 2014-2024, Active: 2014-2024
Stamp: 04-02-2024 06:57:15, #4314260888098342435
File: C:\Thomas\Desktop\gekko\testing\trace.gcm line 4
Vars: x1, x2
```

In the data trace viewer, you may click the ">" symbol to the left of `x` to unfold the `x1` and `x2` sub-traces. In the above example, when you print `x`, it will print as the value `3`. Data tracing can tell you that the value `3` originated from the statement `x = x1 + x2`, where `x1` originated from `x1 = 1` and `x2` originated from `x2 = 2` (`x1` and `x2` are "precedents" to `x`). This explains the value 3. In addition to this, the data trace system can, among other things, also tell the user *when* `x`, `x1` and `x2` were computed, in which Gekko program file (and line), and over which time periods. The `x` data traces regarding `x1` and `x2` are kept even if `x1` and `x2` are subsequently deleted (but `x` is not).

Data tracing works in and out of databanks, so you may trace series statements through Gekko databanks (.gbk). In Gekko >= 3.1.16, Gekko tracing is switched on per default. To switch it off, you may use this option at the top of your Gekko program:

```
option databank trace = no;
x1 = 1;
x2 = x1 + 2;
x3 = x1 + x2 + 3;
//no traces at all
```

To turn off tracing for only a few statement lines, you may use the following BLOCK statement:

```
block databank trace = no;
  x1 = 1;
  x2 = x1 + 1;

```

```
end;
x3 = x1 + x2 + 3;
//x1 and x2 will not have traces
```

To turn tracing off for at a single statement, you may use this local option:

```
x1 <option databank trace = no> = 1;
x2 = x1 + 1;
x3 = x1 + x2 + 3;
//x1 will not have traces
```

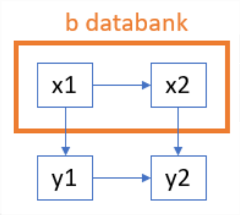
Data tracing costs some performance and makes Gekko databanks larger. The data trace time consumption is reported as a percentage after each Gekko job. See this appendix for much more about this, and how traces can be occasionally pruned off and stored in a central "archive trace databank".

Functions

The following functions can be used together with data traces:

Function name	*Description*	*Examples*

traceBank(b, type)
traceBank(x, b, type)
	After a databank containing traces is read/opened, the traceBank() function can find data connections between variables. Arguments are x for the Gekko (array-)timeseries, b is a string containing a databank name, and type is either `'precedents'`, `'precedents1'` , `'dependents'` or `'dependents1'`. The x argument may be a string like `'x[a]!q'` or a series object like `x[a]!q`. You may call the function without the x argument: in that case, all series names are considered.	`open<edit>b; clear b;`
		`x1 = 1;`
		`x2 = x1 + 2;`
		`close b;`
		`open b;`
		`y1 = 1 + b:x1;`
		`y2 = y1 + 2 + b:x2;`
		`prt tracebank('y2', 'b', 'precedents');`
		`prt tracebank('y2', 'b', 'precedents1');`
		`prt tracebank('x1', 'b', 'dependents');`
		`prt tracebank('x1', 'b', 'dependents1');`
		
		`//Output:`
		`//'x1!a', 'x2!a'`
		`//'x2!a'`
		`//'y1!a', 'y2!a'`
		`//'y1!a'`

	Note: for `'dependents'` type, it is recommended that x is a string name rather than a series object (because the series object may not even exist).
	The options `'precedents1'` and `'dependents1'` limits the effects to *direct* effects only. For instance, in the example, `y2` directly depends upon `x1` from the `b` databank, but not directly upon `x2`.
	Note: you may remove frequencies from the list elements with for instance `tracebank('y2', 'b', 'precedents').removefreq()`.
	[New in Gekko 3.1.22]
		
		`//Relationships b`
		`//timeseries. The`
		`represents the b`
		`//The series x1 and x2 reside in the b databank,`
		`//where x2 depends upon x1. The series y1 depends`
		`//upon b:x1, and the series y2 depends upon y1`
		`//and b:x2. The first tracebank() call shows that`
		`//y2 depends upon both b:x1 and b:x2, the former`
		`//of these indirectly via x2. The third`
		`//tracebank() shows that b:x1 affects both y1 and`
		`//y2, the latter of these indirectly via both y1`
		`//and b:x2.`

traceDelete2()
traceDelete2(x)
	Deletes all trace references in the first-position databank (or use x for an optional databank name). After this, a traceStats2() will show 0 traces.
	[New in Gekko 3.1.16]
		`tracedelete2();`
		`tracedelete2('Work');`

traceStats2()
traceStats2(x)
	For now, only a simple overview of the number of traces can be shown (x is an optional databank name). Depth = 0 means traces that are directly assigned to existing timeseries, whereas depth = *n* > 0 indicate traces that are not assigned to existing timeseries, but are instead assigned to other traces with depth = *n*-1. Traces with depth > 0 can be thought of as remnants	`tracestats2();`
		`tracestats2('Work');`

	of timeseries that do not (or do no longer) exist in the particular databank. [New in Gekko 3.1.16]	

Notes

Data tracing has some similarities to decomposition of model equations. In both cases, data values are tracked backwards to their origins. It is planned for data tracing to be able to send off a data trace like for instance `x = x1 + x2` to the DECOMP engine for further analysis.

When a new trace is about to be added to a series `x`, Gekko will first check if the last added trace is "practically identical" to the new one. If so, the new trace is omitted. "Practically identical" means that the code and the file + line number are the same, even though the entailed time period may differ. This avoids creating a web of trace dependencies in time-loops like the following `x <2010 2020> = 100; for val %t = 2010 to 2020; x[%t] = x[%t] + 1; end;`. Here (and in contrast to `x = 100; x = x + 1;`), `x` will get a trace corresponding to each year in the loop. But the 2020-trace will only point to the 2010-trace as a subtrace, not to the 2011-trace, the 2012-trace and so on.

Regarding data trace stamps, Gekko 3.1.16-3.1.19 used local time stamps, whereas Gekko >= 3.1.20 uses UTC time stamps. If Gekko >= 3.1.20 for instance reads a data trace written by Gekko 3.1.16-3.1.19, the trace data stamp may seem to be set 1-2 hours earlier than it really was (for Danish users). Since this is a question of time zones, the stamp *dates* are typically unaffected by the change (for Danish users).

Related options

OPTION databank trace = yes; [yes|no];

Related statements

DECOMP, DISP, PRT, SERIES

4.3.91 TRANSLATE

The statement translates command files (.cmd) from AREMOS or older versions of Gekko.

- Translate from AREMOS to Gekko 3.0. Details here.
- Translate from Gekko 2.0/2.2/2.4 to Gekko 3.0. Details here.
- Other translation: `<move>`, `<remove>` for Gekko 3.0 files.

Note: to translate from Gekko 1.8 to Gekko 2.0, this is possible using for instance the Gekko 2.4 or 3.0 stable release versions. The 1.8 to 2.0 translator is not available in Gekko 3.1.x.

Note that the translators are for command files (.cmd/.gcm), not model files (.frm).

The user is advised to compare the non-translated and translated files line by line, perhaps with a file comparison utility. The translators are by no means perfect, and should be thought of as translation guides.

Syntax

```
translate < AREMOS GEKKO20 MOVE REMOVE KEEPTYPES > filename ;
```

AREMOS	(Optional). Translates command files (.cmd) from AREMOS syntax. If you prefer to omit parentheses around lists of numbers in SERIES statements, you can use TRANSLATE<remove> to remove these afterwards.
GEKKO20	(Optional). Translates program files (.gcm) from Gekko 2.0/2.2/2.4 syntax to Gekko 3.0 syntax. NOTE: the translator is *not* intended for .frm files (model files). Use `TRANSLATE <gekko20 keeptypes> ... ;` to keep type keywords in assignment statements.
KEEPTYPES	(Optional). An option for the Gekko 2.0 translator only. You may optionally keep types when translating assignments, so that `VAL x = 100;` becomes `VAL %x = 100;` instead of just `%x = 100;`. Applies to SERIES, VAL, DATE, STRING, LIST and MATRIX.
MOVE	(Optional). A simple translator that moves the <>-field in assignments (typically SERIES statements). For instance: `<2010 2020> y = 100;` becomes `y <2010 2020> = 100;`. This translator does not touch anything else, so it is for existing 3.0 files where the user would like to move the option fields. It will also convert `series <2010 2020> y = 100;` into `series y <2010 2020> = 100;`, but not remove the SERIES keyword.

REMOVE	(Optional). A simple translator that removes superfluous ()-parenteses around a comma-separated list of numbers in assignments (typically SERIES statements). For instance: `y = (2, 3, 4);` becomes `y = 2, 3, 4;`. This translator does not touch anything else, so it is for existing 3.0 files where the user would like to remove such parentheses.
filename	Filenames may contain an absolute path like `c:\projects\gekko\bank.gbk`, or a relative path `\gekko\bank.gbk`. Filenames containing blanks and special characters should be put inside quotes. Regarding reading of files, files in libraries can be referred to with colon (for instance `lib1:bank.gbk`), and "zip paths" are allowed too (for instance `c:\projects\data.zip\bank.gbk`). See more on filenames here.

The translator to Gekko 3.0 will create a new file with "_translate" added to the file name.

Examples

To translate calc2.cmd (written in AREMOS) to calc2.gcm (Gekko 3.0 syntax), use

```
translate <aremos> calc2;
```

Translate p.gcm (written in Gekko 2.0/2.2/2.4) to p_translate.gcm (Gekko 3.0):

```
translate <gekko20> p;
```

The translators will do their best to come up with syntax suggestions to suit Gekko 3.0, but be warned that the resulting file may not even parse or run, or if it runs, it may even yield the wrong results! So please use with some care. That being said, the translators typically alleviate quite a lot of tedious editing of relatively easy code, so that the user can concentrate on translating the more tricky parts.

Please inspect the code thoroughly afterwards, preferably with a file comparing tool. (For instance in Total Commander: mark the two files, and use 'Files' --> 'Compare by content' to highlight the differences).

Related statements

RUN

4.3.92 TRUNCATE

TRUNCATE shortens a timeseries, so that the observations outside the given period are discarded. The statement will handle normal timeseries and array-timeseries, but will not truncate timeseries stored inside lists or maps.

Syntax

truncate < *period* > *variables* ;

period	(Optional). Local period, for instance `2010 2020`, `2010q1 2020q4` or `%per1 %per2+1`.
variables	A list of variables to truncate

- If no period is given inside the <...> angle brackets, the global period is used (cf. TIME).
- If a variable is stated without databank, the databank is assumed to be the first-position databank.
- Looping: with a list like for instance `#m = a, b;`, you may use `truncate x{#m};` to truncate `xa` and `xb`.

Example

To remove all data outside the sample 1990-2010 for all variables starting with 'fx' in the first-position databank:

```
truncate <1990 2010> fx*;
```

You may omit the period:

```
truncate p1, p2, p3;
```

In that case, the three variables are truncated according to the global time period.

Related statements

SERIES

4.3.93 UNFIX

UNFIX works in conjunction with SIM<fix>. It removes any goals/means set with the EXO and ENDO statements.

Syntax

unfix ;

Example

See the examples in the ENDO help file.

Note

Note that when ENDO and EXO are active, there is a target icon at the bottom right in the user interface, stating the number of goals and means set. After UNFIX, this icon disappears, since there are no more goals/means set for SIM<fix>.

Related options

OPTION model type = default; //default | gams

Related statements

ENDO, EXO, SIM

4.3.94 UNLOCK

UNLOCK is used to set an open databank editable.

Syntax

unlock databank ;

Example

```
open mybank;
unlock mybank;
```

This opens up mybank in the last position on the databank list (F2), and sets it editable (so that data inside can be changed).

Note

You may use OPEN<edit> to open an editable databank in the first position. `open<edit>mybank;` is actually short for `open<first>mybank; unlock mybank;`.

Related statements

LOCK, OPEN

4.3.95 VAL

The VAL statement is used to assign a numeric value to a scalar variable of value type. Value names always start with the symbol `%`, like the other scalar types date and string. Using the VAL keyword is no longer mandatory in Gekko 3.0.

> **Note**: the VAL type is sometimes called 'val' and sometimes 'value' in this documentation. These two are the exact same thing. Note however that you must use 'val' and not 'value' as type description in function and procedure definitions, and in for loops. Note also that Gekko has no integer type: just use a VAL type.

Value scalars can be used in expressions, for instance in series or in PRT/MULPRT/PLOT/SHEET/CLIP statements. An integer value may be used as an annual or undated date.

Syntax

```
%v = expression;
val %v = expression;
val ?; //print val scalars
```

It is no longer legal to use for instance `val v = 1.23;`, omitting the `%`.

Example

You may use values as a container for fixed floating point numbers for use in your program.

```
%v1 = 1.10;
%v2 = 1/(10 + %v1);
tell '{%v1}, {%v2}';
prt %v1, %v2;
tg = %v1 * tg;
tg <2010 2013> += (%v2, -%v2, 0.5*%v2, 0.01);  //must use
parentheses when not simple numbers
```

When printing values and series at the same time in a PRT statement, note that these values are held constant over any time period.

You can pick out individual timeseries observations with [] and put these into a value:

```
%gdp2020 = gdp[2020];
```

After this, the value `%gdp2020` stores the value of series `gdp` in 2020.

You may loop over value ranges, see FOR.

To convert dates or strings into values, you may use the val() function.

You may compose the value names if you need to, using {}-curlies:

```
for val %i = 1 to 3;
  %v{%i} = 100 * %i;  //defines %v1 = 100, %v2 = 200, %v3 = 300
end;
for val %i = 1 to 3;
  %v = %v{%i};
  tell 'Index {%i} has value {%v}';
end;
```

The result:

```
Index 1 has value 100
Index 2 has value 200
Index 3 has value 300
```

Here, the expression `%v{%i}` picks out the corresponding v-value. In general however, for such use column vectors (n x 1 matrices) or lists or values are recommended, cf. the identical example in the MATRIX section.

The following creates values from a list:

```
reset;
for string %i = a, b, c;  //or: for string %i = ('a', 'b', 'c');
  %{%i} = 100;
end;
mem;
```

This creates value scalars `%a`, `%b` and `%c`, all with value 100.

Note

See the page with syntax diagrams if the basics of names, expressions, etc. is confusing.

If you need to convert a date or string scalar to a avlue type, use the val() conversion function.

You may use m() to indicate a missing value.

See also the format() function and OPTION string interpolate format val = ... ; regarding {...}-formatting of values inside strings.

Regarding variable types and the Gekko type system, see the VAR section. In this appendix, variable assignment rules, including variable types, is explained in more detail.

Related options

OPTION string interpolate format val = "";

Related statements

DATE, STRING, SERIES, FOR, IF

4.3.96 VAR

Gekko statements of the type `... = ... ;` assign the right-hand side expression to the left-hand side variable, which is also called an assignment. Assignments can be done for different types of variables, and you can read more about the specific variable types and their syntax on these pages: SERIES, VAL, DATE, STRING, LIST, MAP and MATRIX.

The `var` keyword can always just be omitted, so it is in a sense superfluous in assignment statements. For example, these three statements are equivalent:

```
y = 100;
var y = 100;
series y = 100;
```

The first statement works because Gekko notices that the left-hand side variable has no type symbol `%` or `#`, so the variable must therefore be a timeseries (with constant value 100). The next statement just adds the superfluous `var` keyword. The third statement guarantees that `y` ends up being a timeseries, but we knew this already. Therefore,

If you use a type indicator, it will guarantee that the left-hand side of the expressions ends up being of the indicated type. For instance:

```
date %x = 2020;
```

In this case, the right-hand side is actually a numeric value (it would have been a date, if it had been stated as for instance `2020a`, or `2020q3`), and the `date` keyword indicates that Gekko should try to convert the right-hand side into a date, which is possible in this case. On the contrary,

```
var %x = 2020;
```

does not indicate any type on `%d`, so `%d` will end up being a value type. The same goes for the following:

```
%x = 2020;
```

In this case, `%d` also ends up being a value type, too. As a last note, the following statements are equivalent:

```
date %d = 2020;
%d = date(2020);
```

To conclude: in most cases, the type identifier (or the `var` keyword) can just be omitted in assignment statements (statements of the form `... = ... ;`, unless the user wants to be absolutely sure of the type of a given left-hand side variable. The following are examples of the compact way of assigning, without using type indicators or VAR:

```
x = 100;                         //will become a series
x <2020 2030> = 100;             //will become a series
%x = 'abc';                      //will become a string
%x = 2.5;                        //will become a value
%x = 2020;                       //will become a value: works fine
as annual date, too
#x = (1, 2*3-4);                 //will become a list of values
#x = 1, 2;                       //allowed for simple numbers (no
math expressions).
#x = ('a', 'b');                 //will become a list of strings,
"#x = a, b;" can be used too
#x = a, b;                       //allowed for simple strings
#x = (%x1 = 'abc', %x2 = 2.5);   //will become a map
#x = [1, 2];                     //will become a 1x2 matrix (row
vector)
```

In this appendix, variable assignment rules, including variable types, is explained in more detail.

Related statements

SERIES, VAL, DATE, STRING, LIST, MAP, MATRIX

4.3.97 WRITE

The statement writes variables to a Gekko databank file (.gbk). A .gbk file is a Gekko-specific binary databank format that stores series, values, dates, strings, lists, maps, and matrices.

Syntax

write < ***period*** **RESPECT TRACE=... >** ***filename*** **;**
write < ***period*** **RESPECT TRACE=... >** ***variables*** **FILE=*filename*** **;**

period	(Optional). Without a time period indicated, Gekko will write all the data for all observations. When a period is indicated, the written data(bank) is truncated.
RESPECT	(Optional). With this option, if no period is given, the global period is used.
TRACE=	(Optional). Use `yes` or `no`, `yes` is default. With `trace=no`, no data traces are written.
variables	Variables or lists (wildcards and bank indicators may be used), and items may be separated by commas. If no variables are given, the full first-position databank is written.
filename	Filenames may contain an absolute path like `c:\projects\gekko\bank.gbk`, or a relative path `\gekko\bank.gbk`. Filenames containing blanks and special characters should be put inside quotes. Regarding reading of files, files in libraries can be referred to with colon (for instance `lib1:bank.gbk`), and "zip paths" are allowed too (for instance `c:\projects\data.zip\bank.gbk`). See more on filenames here.

- If no period is given inside the <...> angle brackets, no time period is used.
- If a variable is stated without databank, the databank is assumed to be the first-position databank.

There is the following equivalence between WRITE and EXPORT: WRITE = EXPORT<all>, and the inverse: EXPORT = WRITE<respect>. If a local period is set, WRITE and EXPORT behave in the same way.

Examples

You may write the contents of the first-position databank like this:

```
write data;
```

This will produce the file `data.gbk`, containing the first-position databank. If you only want subset of the variables (or a subset of the time period), you may write for instance:

```
write<2040 2050> fy, fe, fm FILE=sim4050;
```

This produces the file `sim4050.gbk`, containing the three variables `fy, fe, fm` over the period 2040-50. If practical, you may also use wild-card lists:

```
write fx* file=fxfile;
```

This writes all variables starting with `fX` to the file `fxfile.gbk`. Actually `write ** file=databank;` is equivalent to `write databank;`, cf. the wildcards page regarding the double star `**` notation. To write a list of strings containing variable names, use `{}`-curlies:

```
#m = fy, fe, fm;     //or: #m = ('fy', 'fe', 'fm);
write <2040 2050> {#m} FILE=sim4050;
```

Without the braces, the list `#m` itself would have been written, not the three series.

Note

If `option folder = yes`, and `option folder bank` is set, the WRITE statement tries to write to that particular folder instead of the working folder.

If a model has been loaded, and all the endogenous variables of the model exist in the first-position databank, the WRITE statement will store info regarding the model, last simulation period etc. inside the .gbk file. After this, when reading the databank again, a link to this model info is provided. This can be practical when in doubt about when the variables in a given databank were simulated, the simulation period, the model name and signature, etc.

Gekko 3.x.x always writes a gbk databank in the 1.2 format version. See more about gbk formats under the READ statement, including how to exchange data between newer and older Gekko versions.

See the Gekko menu 'Options' --> 'Program dependency tracking' or use `option global dependency tracking = ...;` to activate dependency tracking, so that the use of external files (for instance program files, read/written databanks etc.) are shown as a list at the end of a Gekko session.

Related options

OPTION folder bank = [empty];

Related statements

READ, IMPORT, EXPORT, SHEET, MODEL, HDG

4.3.98 X12A

The X12A (X12-arima) performs seasonal adjustment on quarterly or monthly data. The component uses a well-known external component developed by the US Census Bureau, and is similar to the AREMOS command with the same name. (Later on, X13-arima-seats and Tramo-Seats might be provided in Gekko, perhaps via the JDemetra+ project).

Syntax

```
x12a < period  PARAM=...  BANK=... >  variables ;
```

period	(Optional). If not stated, Gekko will use the global time period.
variables	Variables and/or list(s). Wildcards may be used.
PARAM=	(Optional). A text string containing parameter values for X12A. If a parameter string is not provided, Gekko will use `'save=(d10, d11, saa) mode=mult sigmalim=(1.50,2.50) seasonalma=msr force=totals'` as default parameter. With the default parameter, if you are correcting the series `y`, the series `y_saa` is the result.
BANK=	(Optional). A databank name indicating where the timeseries are located.

- If no period is given inside the <...> angle brackets, the global period is used (cf. TIME).
- If a variable without databank indication is not found in the first-position databank, Gekko will look for it in other open databanks if databank search is active (cf. MODE).

Gekko appends the `save=(...)` types with underscore in the timeseries names, for instance `y_saa` for the `saa` type.

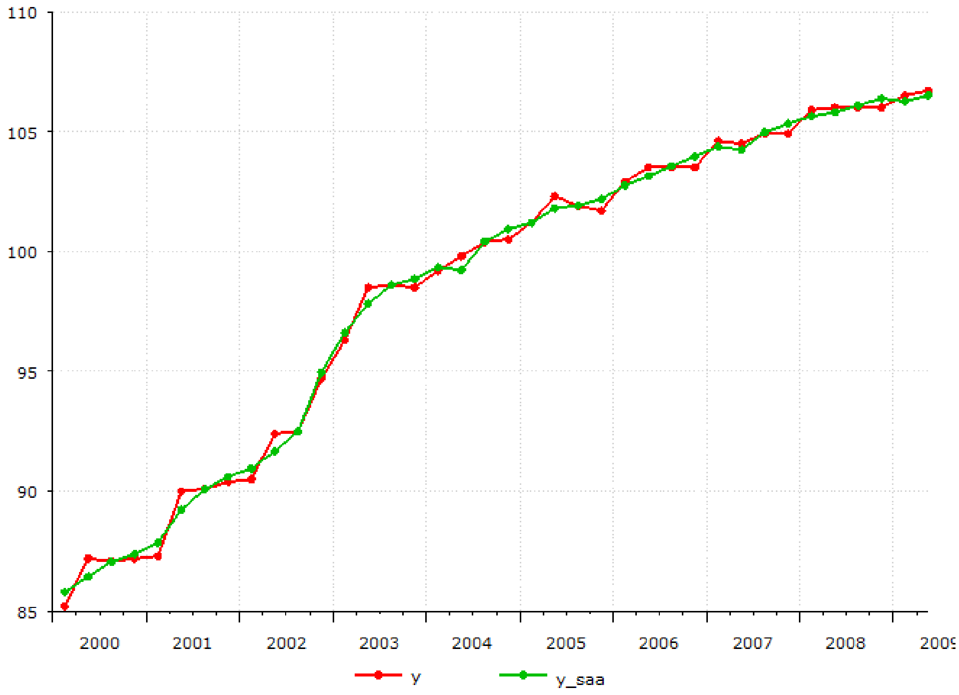
Example

You may try the following example:

```
reset;
option freq q;
create y;
```

```
time 2000q1 2009q2;
y = 85.2, 87.2, 87.1, 87.2, 87.3, 90, 90.1, 90.4, 90.5, 92.4, 92.5,
94.7, 96.3, 98.5, 98.6, 98.5, 99.2, 99.8, 100.4, 100.5, 101.2,
 102.3, 101.9, 101.7, 102.9, 103.5, 103.5, 103.5, 104.6, 104.5,
104.9, 104.9, 105.9, 106, 106, 106, 106.5, 106.7;
%p = 'save=(d10, d11, saa) mode=mult sigmalim=(1.50,2.50)
seasonalma=msr force=totals';
x12a <param = %p> y;  //the parameter could be omitted here,
because the default parameter is identical
plot y, y_saa;
```

Result:



Note the parameter `save = (d10, d11, saa)`. You can choose between `c17`, `d10`, `d11`, `d12`, `d13` and `saa`. The last one is only available if `force=totals` is set as parameter. Normally, the `_saa` series is considered the output of X12A.

Note

For much more information on X12-arima, see https://www.census.gov/ts/x12a/v03/x12adocV03.pdf from the US Census Bureau. See section 2.7 regarding program limits. It seems that there is a limit starting at around 60 years of data. For quarterly data, only up to 60 calendar years are supported. And for monthly data, only up to 59 years (708 months) are supported. To remedy this, you may split the period into overlapping sub-periods and use SPLICE to splice them.

Please note that all the parameters are located inside a text string.

If you need to inspect the results in more detail, please inspect the `tempX12aFile...` files in the temporary files folder (cf. the Gekko menu Help --> About...).

If you need more advanced seasonal correction, you may consider using the R interface (see R_RUN). R contains quite a lot of facilities for seasonal correction.

The example can be exactly reproduced in AREMOS with the following AREMOS command:

```
x12a y d10,d11,saa "mode=mult sigmalim=(1.50,2.50) seasonalma=msr
force=totals print=alltables";
```

Related statements

COLLAPSE, SMOOTH

4.3.99 XEDIT

The XEDIT statement uses the open-source XML Notepad xml editor to open up the designated file. The statement is practical for editing plot files (.gpt), or table files (.gtb). In general, a xml editor is much easier to use for editing xml files than a text editor.

Tip: use 'View' --> 'Expand All' to unfold all XML nodes for better overview. Use Ctrl+D to duplicate a xml node (including its children nodes). The nodes are easy to copy, delete and move areound in XML Notepad.

Syntax

xedit *filename* ;

filename	Filenames may contain an absolute path like `c:\projects\gekko\bank.gbk`, or a relative path `\gekko\bank.gbk`. Filenames containing blanks and special characters should be put inside quotes. Regarding reading of files, files in libraries can be referred to with colon (for instance `lib1:bank.gbk`), and "zip paths" are allowed too (for instance `c:\projects\data.zip\bank.gbk`). See more on filenames here. The extension .gpt is automatically added, if it is missing. If the filename is set to '*', you will be asked to choose the file in Windows Explorer.

Examples

You may use this to open up the file `p.gpt` from the working folder:

```
xedit p;
```

The .gpt extension is automatically inserted. You may select .gpt files like this:

```
xedit *;
```

This will open up a file dialog with .gpt files to choose from (you can choose other extensions, too).

To edit a table file called `t1.gtb`, you may use:

```
xedit t1.gtb;
```

Related statements

SYS, EDIT, PLOT, TABLE

4.4 Gekko functions

Gekko in-built functions can be used in expressions. The input parameters and the output type is described. The functions are divided into categories.

- Functions. Details on Gekko functions.

Note that user-defined functions are possible, too. See the FUNCTION statement.

4.4.1 Functions

Gekko has a number of in-built functions, listed below. Note that all Gekko functions implement so-called UFCS so that a function like for instance `f(x, y)` can generally be written as `x.f(y)`, and `f(x)` can generally be written as `x.f()`.

Mathematical functions:

Function name	*Description*	*Examples*
abs(*x*)	Returns the absolute value of x (series, val or matrix). Returns: series/value/matrix	`%v1 = abs(%v2);`

avg(x1, x2, ...)	Returns the average of x1, x2, ... etc. The input parameters may be series or value.
	Returns: series/value
		`y = avg(x1, x2, x3);`
		`%v = avg(%v1, %v2, %v3);`

avgt(x)
avgt(<t1 t2>, x)
	Returns the time-average of the observations of the timeseries x over the local/global time period (or over t1 to t2, if indicated)
	Returns: series
		`y = avgt(x);`
		`y = avgt(<2020 2025>, x);`
		`y = x.avgt(<2020 2025>);  //same as above`

ceiling(x)	Returns the the smallest integer which is greater than or equal to x (series, val or matrix). See also `int()`, `floor()` and `round()`. Returns: series/value/matrix	`prt ceiling(-2.2);`

dif(x) or diff(x)	Absolute time-difference of series x: can also be used on left side of '='. Does not work on value.
	Returns: series
		`y = dif(x);`
		`dif(y) = 100;`

difa(x) or diffa(x)	Year-over-year absolute time-difference. Same as `dif(x)`, but will use 4 lags	`y = difa(x);`

dify(x) or diffy(x)	for quarterly data, and 12 lags for monthly data. See also `<yoy>` option for PRT/PLOT etc. All four functions do the same. Returns: series	
difq(x) or diffq(x)	Quarterly difference. Same as `dif(x)`, but will use 3 lags for monthly data. You may use diffq() as alias. Returns: series	`y = difq(x);`

dlog(*x*)	Logarithmic time-difference of series x: can also be used on left side of '='. Does not work on value.
	Returns: series
		`y = dlog(x);`
		`dlog(y) = 0.02;`

dloga(*x*)
dlogy(*x*)
	Year-over-year logarithmic time-difference. Same as `dlog(x)`, but will use 4 lags for quarterly data, and 12 lags for monthly data. You may use dlogy() as alias. See also `<yoy>` option for PRT/PLOT etc.
	Returns: series
		`y = dloga(x);`

dlogq(*x*)	Quarter-over-quarter logarithmic time-difference. Same as `dlog(x)`, but will use 3 lags for monthly data Returns: series	`y = dlogq(x);`
eps()	Return a very small number (1e-300), to represent an *eps* value that is for practical purposes zero, but has special meaning and calculation rules (like missing values also have). For instance, the following return the *eps* value: `-eps()`, `eps()+eps()`, `eps()-eps()`, `eps()*eps()`, whereas the following return a missing	

	value: `eps()/eps()` and `log(eps())`. The exp function, `exp(eps())`, returns 1. You may use isEps() to check whether a value is eps.	

exp(*x*)	Returns the exponential value of x (series, value or matrix).
	Returns: series/value
		`y = exp(x);`
		`%v1 = exp(%v2);`

floor(x)	Returns the largest integer which is less than or equal to x (series, val or matrix). See also `int()`, `ceiling()` and `round()`. Returns: series/value/matrix	`prt floor(-2.2);`

iif(in1, op, in2, out1, out2)	Conditional, works like an if statement. Think of it like `if in1 op in2 then out1 else out2`, where op is a string containing the operator `==`, `<>`, `<`, `<=`, `>=`, `>`. The function can be used to avoid explicit time looping for timeseries. The op input must be a string, and the rest of the inputs must be of math type (there is another `iif()`-example here). If needed, you can use `==` (or `<>`) together with `m()` to test for the presence of missing values, see example.
	
	You may alternatively use $-conditionals, see examples under SERIES. See also the `replace()` and `isMiss()` functions for series.
	Returns: series/value
		`time 2010 2012;`
		`in1 = 1, 2, 3;`
		`in2 = 3, 2, 1;`
		`y1 = iif(in1, '<=', in2, 50, 100);`
		`//Result: y1 = 50, 50, 100.`
		
		`in3 = 100, m(), 300;`
		`y2 = iif(in3, '==', m(), 1000, in3);`
		`//Result: y2 = 100, 1000, 300`

int(x)	Returns the integer value of x (series, val or matrix), discarding the fractional part (after the `.`). See also `floor()`, `ceiling()` and `round()`. Returns: series/value/matrix	`prt int(-2.2);`

isEps(x)	The input parameter may be series or value, and the function returns 1 or 0 (or 1's and 0's) depending upon whether the input values are *eps* or non-*eps*, cf. the eps() function.	`time 2021 2023;`
		`x = 10, 20, 30;`
		`x[2022] = eps();`
		`prt isEps(x);`
		`//Will print 0, 1, 0`
		`//isEps() also works for scalars,`
		`//cf. the following:`
		`prt isEps(eps())`
		`//Will print 1`

isMiss(x)
isMiss(x, 'all')
	The input parameter may be series or value, and the function returns 1 or 0 (or 1's and 0's).
	
	If x is a series, the first variant of the function returns 1, else 0, over the period that contains data (cf. `fromSeries(x, 'dataStart')` and `fromSeries(x, 'dataEnd')`). With option `'all'`, the local or global period is used instead.
	
	An example is provided regarding how to replace missing values inside series, but note that this can also be done with the `iif()` or `replace()` functions.
	
	See also `allMiss()`.
	
	Returns: series/value
		`prt isMiss(1); prt isMiss(miss());`
		`time 2020 2023;`
		`z = 1, m(), 3, m();`
		`prt <n> z, z.isMiss(), z.isMiss('all');`
		`//Note that for 2023, the function returns a`
		`//missing value. The function only finds missing`
		`//values between non-missing values.`
		`//With 'all' option, missing values that are not`
		`//enclosed between non-missing values will be`
		`//set to 1, too.`
		`prt sumt(z.isMiss('all'));`
		`//Prints the number of missings over 2020-23.`
		
		`time 2021 2025;`
		`x = 1, 2, m(), 4, 5;`
		`y = 100;`
		`y = y $ (x.isMiss()) + x $ (not x.isMiss());`
		`//y becomes 1, 2, 100, 4, 5. See also iif() function.`
		
		`time 2021 2026;`
		`x = 1, m(), m(), 4, m(), 6;`
		`x $ (x.ismiss('all')) %= 0;`
		`//x becomes 1, 1, 1, 4, 4,`

		`6,`
		`//filling out missing "holes"`

lag(x, lag*)*	Lags series x a number of periods. Note the sign of the lag: `lag(x, 2) = x[-2]`. Can be used if x is an expression. Returns: series	`y = lag(x, 2);  //same as x[-2]`

log(*x*)	Returns the natural logarithmic value of x (series, value or matrix). Can also be used on the left side of '='.
	Returns: series/value/matrix
		`y = log(x);`
		`%v1 = log(%v2);`
		`log(y) = a * log(b);`

m() or miss()	Returns a missing value. Useful in some series or matrix expressions. Cf. also the m(..., ...) function for matrices.
	Returns: value
		`y <2020 2020> = m();`
		`#m = [1, 2; m(), 4];`

max(x1, x2, ...)	Finds the largest value. Arguments may be value, series or date.
	Returns: value/series/date.
	[New in 3.1.6]
		`prt max(2, 4, 1, 3);`
		`prt max(2000q1, 2000q4);`
		`time 2001 2003;`
		`x1 = 1, 2, 3;`
		`prt max(x1, 2);`

min(x1, x2, ...)	Finds the lowest value. Arguments may be value, series or date.
	Returns: value/series/date.
	[New in 3.1.6]
		`prt min(2, 4, 1, 3);`
		`prt min(2000q1, 2000q4);`
		`time 2001 2003;`
		`x1 = 1, 2, 3;`
		`prt min(x1, 2);`

mod(x1, x2)	Modulo, the remainder of the division x1/x2.
	Returns: value/series.
	[New in 3.1.12]
		`%x = mod(7, 4); //3`
		`%x = mod(8, 4); //0`
		`%x = mod(9, 4); //1`

movavg(x1, lags)	Moving average of series x1.
	Returns: series
		`y = movavg(x, 3);`
		`y = (x + x[-1] + x[-2])/3;  //same`

movsum(x1, lags)	Moving sum of series x1, cf. `movavg()`.
	Returns: series
		`y = movsum(x, 3);`
		`y = x + x[-1] + x[-2]; //same`

pch(*x*)	Percentage growth in series x: can also be used on left side of '='. Does not work on value.
	Returns: series
		`y = pch(x);`
		`pch(y) = 2;`

pcha(x)
pchy(x)
	Year-over-year growth. Same as `pch(x)`, but will use 4 lags for quarterly data, and 12 lags for monthly data. You may use pchy() as synonym. See also `<yoy>` option for PRT/PLOT etc.
	Returns: series
		`y = pcha(x);`

pchq(*x*)	Quarterly growth. Same as `pch(x)`, but will use 3 lags for monthly data. Returns: series	`y = pchq(x);`

pow(x, y)	The exponent must be a value or number, not a series. The function `pow(x, y)` is equal to `x**y` or `x^y`, that is, a in the y'th power. You may use `power(x, y)` as synonym.
	Returns: series/value
		`y = pow(x1, %x2);`
		`%v = pow(%x1, %x2);`

rnorm(mean, var)
rnorm(mean, vcov)
	Returns a random number from a normal distribution with mean and variance provided. If fed with a n x 1 matrix of averages, and a n x n covariance matrix, the function will return a n x 1 matrix of values. See also `rseed()` and `runif()`.
	Returns: value/matrix
		`%n = rnorm(0, 1);`
		`%n = rnorm(-100, 25**2);`
		`#n = rnorm(#mean, #vcovar);`

rseed(x)	Given value x, it sets a random seed for `runif()`	`rseed(12345);`

	and `rnorm()` functions. The function returns the seed as a value, but can be used without return value. Returns: value	
round(x, d)	Rounds x (series, val or matrix) to d decimal places. See also `int()`, `floor()` and `ceiling()`. Returns: series/value/matrix	`%v1 = round(%v2, 3);`
runif()	Returns a random number from a uniform distribution between 0 and 1. See also `rseed()` and `rnorm()`. Returns: value	`%v = runif();`

seq(start, end)	Returns a list of integer values or dates between start and end (both included). Start and end must be two values or two dates.
	Returns: list
		`#m = seq(1, 100);`
		`#t = seq(2001q1, 2005q4);`

sqrt(*x*)	Returns the square root of x (series, value or matrix).
	Returns: series/value/matrix
		`y = sqrt(x);`
		`%v1 = sqrt(%v2);`

sum(x1, x2, ...)
sum(list, x)
	Returns the sum of x1, x2, ... etc. The input parameters may be series or value.
	If the first argument is a list name (or a list of list names), the sum function will sum the second argument over these lists.
	Returns: series/value
		`y = sum(x1, x2, x3);`
		`%v = sum(%v1, %v2, %v3);`
		`y = sum(#j, x[a, #j]);`
		`y = sum((#i, #j), x[#i, #j]);`
		`y = sum(#j, xa{#j});`
		`y = sum((#i, #j), x{#i}{#j});`
		To sum a simple list `#j` of series names, you may use this:
		`y = sum({#j}); //shorter than sum(#j, {#j})`
		Note that for PRT/PLOT etc., you should use the more explicit `PRT sum(#j, {#j});` because

		PRT/PLOT auto-unfolds 'uncontrolled' lists into columns/lines.

sumt(x)
sumt(<t1 t2>, x)
	Returns the time-sum of the observations of the timeseries x over the local/global time period (or over t1 to t2, if indicated)
	Returns: series
		`y = sumt(x);`
		`y = sumt(<2020 2025>, x);`
		`y = x.sumt(<2020 2025>);  //same as above`

tanh(x)	Returns the hyperbolic tangens of x (series, value or matrix). Returns: series/value/matrix	`%x = tanh(0.5);`

tiny()	Return a very small number (1.7777e-301), which can be useful to guard against for instance 0's in fractions, cf. example.	`x1 = 1; x2 = 2;`
		`y = (x1 + tiny())/(x2 + tiny());`
		`//y1 will become 0.5`
		`x = 0; x2 = 0;`
		`y2 = (x1 + tiny())/(x2 + tiny());`
		`//y2 will become 1`
		`//If x1 is <> 0 and x2 == 0, y is near-infinity.`

Conversions

Function name	*Description*	*Examples*

date(x)	Tries to convert the scalar x to date type. See also under "date combining functions".
	Returns: date
		`%d = date(2000+15);`
		`Result: %d = 2015.`

dates(x)	Tries to convert each element of the list x into a date.
	Returns: list
		`#m1 = (2001, 2002, 2003);`
		`#m2 = dates(#m1);  //or: #m1.dates()`

data(x)	Converts a string x containing blank-separated numbers to a list of values.
	Returns: list of values
		`#m = data('1 2 3');`
		`x = data('1 2 3');`

format(x, code)	Formats the value/date/string `x` by means of the formatting `code`. The formatting code is as follows for values:
	
	`'[width]:[format]'`
	`'[width]:[format]=[culture]'`
	
	The width specifies that the string will be at least `[width]` characters wide. If the `[width]` is positive, the number is right-aligned within the field, and if it is negative, it is left-aligned.
	
	The `[format]` follows the conventions shown here or here, so you may either use a pattern like `0.000` or `0.###` (exactly three digits or at most three digits), or you may use a description like `F3` (floating point, three digits). So `12:0.000` or `12:F3` are both a 12 characters wide field, and a number with three decimals.
	
	The `[culture]` argument after an optional `=` symbol changes how thousands and decimal operators look like. For instance, `en-US` is English (US variant), `da-DK` is Danish, see the possible cultures here. So `format(1234567.8987, '0,0.00=da-DK')` will return `1.234.567,90`. Without culture, it would return `1,234,567.90`.
	
	A value format can be set globally with `option string interpolate`
		`%v = 12.3456;`
		`%d = 2020q1;`
		`%s = 'abc';`
		
		`tell;`
		`tell '123456789012' + '|';`
		`tell '------------' + '|';`
		`tell %v.format('0.000') + '|';`
		`tell %v.format('12:0.000') + '|';`
		`tell %d.format('12') + '|';`
		`tell %s.format('12') + '|';`
		`tell %v.format('-12:0.000') + '|';`
		`tell %d.format('-12') + '|';`
		`tell %s.format('-12') + '|';`
		`tell '------------' + '|';`
		
		`// 123456789012|`
		`// ------------|`
		`// 12.346|`
		`//       12.346|`
		`//       2020q1|`
		`//          abc|`
		`// 12.346      |`
		`// 2020q1      |`
		`// abc         |`
		`// ------------|`

	`format val = ... ;`. This can be practical when printing out for instance a table of values with the same value formatting. You may also format strings or dates: in that case only `[width]` can be used (positive or negative). In this way, table-like alignment is quite straightforward. Returns: string.	

string(*x*)	Tries to convert the scalar x to string type. See also under "string combining functions".
	If x is a list of strings, the string() function returns a comma-separated list of strings.
	Returns: string
		`%s = string(12) + string(34);`
		`Result: %s = '1234'.`

strings(x)	Tries to convert each element of the list x into a string.
	Returns: list
		`#m1 = (1, 2, 3);`
		`#m2 = strings(#m1); //or: #m1.strings()`

val(*x*)	Tries to convert the scalar x to value type.
	Returns: value
		`%v = val('12' + '34');`
		`Result: %v = 1234.`

vals(x)	Tries to convert each element of the list x into a value.
	Returns: list
		`#m1 = ('1', '2', '3');`
		`#m2 = vals(#m1); //or: #m1.vals()`

Date combining functions

Function name	*Description*	*Examples*

date(d, f, opt)	Converts the date d into a new date with frequency f	`%d = 2021q1;`
		`prt %d.date('a'); //2021`

date(d, f)	(string). The optional option can be omitted, or be `'start'` or `'end'`. Using `'start'` or `'end'` is only relevant when converting from a lower to a higher frequency.
	Beware that week numbers are special around New Year.
	Returns: date
		`prt %d.date('m'); //error!`
		`prt %d.date('m', 'start'); //2021m1`
		`prt %d.date('m', 'end');   //2021m3`
		`prt %d.date('w', 'start'); //2020w53 !!`
		`prt %d.date('w', 'end');   //2021w13`
		`//Note that the first week that fully`
		`//contains the first quarter of 2021 starts`
		`//in 2020!`

date(y, f, sub)
date(y, 'm', m, 'd', d)
	Constructs a new quarterly/monthly/weekly date from y (integer), frequency (string), and subperiod (integer). You may also construct a daily date with a similar syntax.
	Note: you may also use date(x), where x can be a value or a string, and Gekko will try to convert the argument into a date.
	Returns: date
		`%d = date(2020, 'q', 2);  //2020q2`
		`%d = date(2020, 'w', 42);  //2020w42`
		`%d = date(2020, 'm', 12, 'd', 24);  //2020m12d24`

fromExcelDate(v)	Converts an Excel date (the val `v`, counting the number of days since January 1, 1900) to a date with daily frequency. Returns: date (daily)	See examples regarding the toExcelDate() function.

getFreq(d)	Extracts the frequency of a date
	Returns: string
		`%d = 2020q2;`
		`prt %d.getfreq();  //'q'`

getDay(d)	Extracts the day number from a date. Will fail if the date is not daily.
	Returns: value (integer)
		`%d = 2020m3d25;`
		`prt %d.getday(); //25`

getSpecialDa	Finds a special day by year	`prt`

y(year, name)	and name, and returns its daily date. Mostly used for holidays. If you input a wrong name, Gekko will provide a link showing all possible special days. Current possible holiday names (English, and equivalent Danish names):
	
	New_Years_Day Nytaarsdag
	Leap_Day Skuddag
	Maundy_Thursday Skaertorsdag
	Good_Friday Langfredag
	Easter_Sunday Paaskedag
	Easter_Monday Anden_paaskedag
	Labour_Day Foerste_maj
	General_Prayer_Day Store_bededag
	Ascension_Day Kristi_himmelfartsdag
	Whit_Sunday Pinsedag
	Whit_Monday Anden_pinsedag
	Constitution_Day Grundlovsdag
	Christmas_Eve Juleaften
	Christmas_Day Foerste_juledag
	Boxing_Day Anden_juledag
	New_Years_Eve Nytaarsaften
	
	Note that `getSpecialDay(%year, 'Leap_Day')` may return a null value, else it returns February 29 in leap years). The leap day is not a holiday, but is nevertheless kept in this list of special days. See
		`2021.getSpecialDay('Easter_Sunday'); //2021m4d4`
		`prt 2021.getSpecialDay('Paaskedag');     //same day`
		
		`for(val %year = 2021 to 2025);`
		`  %day = %year.getSpecialDay('Christmas_Eve');`
		`  %s = %day.getWeekday('en');`
		`  tell 'In {%year}, Christmas Eve is on a {%s}';`
		`end;`
		`//Result:`
		`//In 2021, Christmas Eve is on a Friday`
		`//In 2022, Christmas Eve is on a Saturday`
		`//In 2023, Christmas Eve is on a Sunday`
		`//In 2024, Christmas Eve is on a Tuesday`
		`//In 2025, Christmas Eve is on a Wednesday`
		
		`prt 2020.getSpecialDay('Leap_Day');`
		`prt 2021.getSpecialDay('Leap_Day');`
		
		`%y = 2020;`
		`if(not %y.getSpecialDay('Leap_Day').isNull());`
		`  //code to handle if %y is a leap year`
		`end;`

	example.	

getMonth(d)
getMonth(d, lang)
	Extracts the month number from a date. More specific than getSubPer(), and will fail if the date is not monthly or daily.
	
	You may input a language ('en' = English, 'da' = Danish), in which case a string is returned.
	
	Returns: value (integer) or string
		%d = 2020m2;
		prt %d.getmonth(); //2
		prt %d.getmonth('en'); //'February'
		prt %d.getmonth('en').lower()[1..3]; //'feb'

getQuarter(d)	Extracts the quarter number from a date. More specific than getSubPer(), and will fail if the date is not quarterly. Returns: value (integer)	%d = 2020q2; prt %d.getquarter(); //2
getSubPer(d)	Extracts the sub-period from a date (1 if annual or undated, the quarter if quarterly, the month if monthly or daily, and the week if weekly). Returns: value (integer)	%d = 2020q2; prt %d.getsubper(); //2

getWeek(d)	Get the week number from a date of weekly frequency. Does not accept daily frequency as argument, but the conversion date('w') can be used as intermediary for this (see examples). Beware that the year may change when converting from daily to weekly frequency.
	Returns: value (integer)
	[New in 3.1.13]
		%d = 2020w20;
		p %d.getWeek(); //20
		p %d.getYear(); //2020
		
		%d = 2019m12d31;
		p %d.getWeek(); //error!
		p %d.date('w').getWeek(); //1
		p %d.date('w').getYear(); //2020!
		
		%d = 2021m1d1;
		p %d.date('w').getWeek(); //53
		p %d.date('w').getYear(); //2020!

getWeekday	Extracts the weekday	%d = 2020m3d25;

(d)
getWeekday(d, lang)
	number from a date. Will fail if the date is not daily. The numbers are as follows: Monday = 1, Tuesday = 2, Wednesday = 3, Thursday = 4, Friday = 5, Saturday = 6, Sunday = 7.
	You may input a language ('en' = English, 'da' = Danish), in which case a string is returned.
	Returns: value (integer) or string.
	[New in 3.1.12]
		`prt %d.getweekday(); //3`
		`prt %d.getweekday('en'); //Wednesday`
		`prt %d.getweekday('en').lower()[1..3]; //wed`
		`prt %d.getweekday('da'); //Onsdag`

getYear(d)	Extracts the year from a date.
	Returns: value (integer)
		`%d = 2020q2;`
		`prt %d.getyear(); //2020`

max(d1, d2, ...)	Finds the largest of any number of dates. (`max()` can also be used with value arguments). Returns: date [New in 3.1.6]	`prt max(2002q1, 2001q4);`
min(d1, d2, ...)	Finds the smallest of any number of dates. (`min()` can also be used with value arguments). Returns: date [New in 3.1.6]	`prt min(2002q1, 2001q4);`

observations(d1, d2)	Counts the number of observations (periods), with both start and end date included. The date difference using `-` is always the same as the number of observations minus 1.	`prt observations(2020q2, 2023q3); //14`
		`prt 2023q3 - 2020q2; //13`

toExcelDate(d)	Converts a daily date into an Excel date (counting the number of days since January 1, 1900). See also fromExcelDate(). Excel	`%v1 = toExcelDate(2019m11d12);`
		`%v2 = toExcelDate(2019m12d3);`
		`prt %v1, %v2; //43781 and 43802`
		`prt %v2 - %v1; //21 days`

	dates can be subtracted to obtain day spans. [New in 3.0.7] Returns: value.	(span) %d = fromExcelDate(%v1 + 100); prt %d; //100 days from 2019m11d12

truncate(d1, d2)	Finds overlap between two different time periods. The period d1 to d2 is compared with the global time period (if no local period is indicated), or with the local time period (if such a period is indicated in the <...> fields). Use a local time period if you need to find the overlap between two arbitrary time windows.
	
	Returns: a list of two elements, start and end date of the resulting period. If the two elements are both null, there is no overlap.
	
	[New in 3.1.6]
		time 2010 2020;
		#m = truncate(2000, 2030); //(2010, 2020)
		#m = truncate(2000, 2015); //(2010, 2015)
		#m = truncate(2000, 2005); //(null, null)
		time 1980 1990;
		#m = truncate(<2010 2020>, 2000, 2030); //(2010, 2020)
		#m = truncate(<2010 2020>, 2000, 2015); //(2010, 2015)
		#m = truncate(<2010 2020>, 2000, 2005); //(null, null)
		//
		// The overlapping period (z) can be visualized as
		// the overlap of these time windows/periods:
		//
		
		// period1 . x x x x x . . .
		// period2 . . . y y y y y .
		// truncate() . . . z z z . . .

String combining functions

Function name	*Description*	*Examples*
[x]-index	Index: returns the character at position x. Returns: string	%s = 'abcd'; prt %s[2]; //'b'
[x1..x2]-index	Index: returns the range of characters from position x1 to x2 (both inclusive). You may omit x1 or x2. Returns: string	%s = 'abcd'; prt %s[2..3]; //'bc'

concat(s1, s2)	Appends the two strings: same as s1 + s2.
	Returns: string
		`%s = concat('He', 'llo');`
		`Result: 'Hello'.`

endswith(s1, s2)	Returns 1 if the string s1 starts with the string s2, else 0. The comparison is case-insensitive.
	Returns: val
		`%v = endswith('abcde', 'cde');`
		`Returns: 1`

includes(s1, s2)	Returns 1 if the string s2 is found inside (as a part of) string 1, else 0 is returned. The search is case-insensitive. Note that the `contains()` function is not useful regarding such in-string searches.
	Returns: val
	[New in 3.1.17].
		`%s = 'abCd';`
		`prt %s.includes('bc');  //1`

index(s1, s2)	Searches for the first occurrence of string s2 in string s1 and returns the position. It returns 0 if the string is not found. The search is case-insensitive
	Returns: val
		`%v = index('onetwothreetwo', 'two');`
		`Returns: 4.`
		`%v = index('oneTWO', 'two');`
		`Returns: 4.`

isAlpha(s)	Returns 1 if all the characters are letters (alphabet). [New in 3.0.5].	`%v = isAlpha('aBc');`
		`Returns: 1`

isLower(s)	Returns 1 if the string contains no uppercase characters. [New in 3.0.5].	`%v = isLower('abc12');`
		`Returns: 1`

isNumeric(s)	Returns 1 if all the characters are of numeric value. [New in 3.0.5].	`%v = isNumeric('123');`
		`Returns: 1`

isUpper(s)	Returns 1 if the string contains no lowercase characters. [New in 3.0.5].	`%v = isUpper('ABC12');`
		`Returns: 1`

length(s)	The length of the string (number of characters).	`%v = %s.length();`

	You may use len() instead of length(). Returns: val	

lower(s)	The string in lower-case letters.
	Returns: string
		`%s = lower('aBcD');`
		`Result: 'abcd'.`

nl()	A system newline/linebreak. Computerwise, `nl()` is equal to the C# string `"\n"`, but beware that you cannot use the Gekko string `'\n'` instead of `nl()`, because the former is equal to the C# string `"\\n"`.
	Returns: string
		`%s = 'a' + nl() + 'b'; //string with newline`
		`tell %s;`
		`#m = %s.split(nl()); //list with 2 elements`
		`p #m;`

prefix(s1, s2)	If s1 is a string, it has the string s2 prefixed (prepended). Returns: string	`%s1 = %s2.prefix('a');`

replace(s1, s2, s3)
replace(s1, s2, s3, max)
	In the string s1, the function replaces all occurrences of s2 with s3. Replacement is case-insensitive.
	
	If max > 0, the replacement is performed at most max times.
	
	Returns: string
		`%s = replace(%s1, %s2);`
		`//or: replace(%s, %s1, %s2)`

split(s1, s2)
split(s1, s2, removeempty)
split(s1, s2, removeempty, strip)
	Splits the string s1 by means of the delimiter s2. Empty elements are removed per default, and the resulting strings are stripped (blanks are removed from the start and end of the strings). The last two options are `1` and `1`, if omitted. [New in 3.0.6]	`%s = 'a, b,c,,d, , e';`
		`#m1 = %s.split(',');`
		`//--> ('a', 'b', 'c', 'd', 'e')`
		`#m2 = %s.split(',', 1, 1);`
		`//--> ('a', 'b', 'c', 'd', 'e');`
		`#m3 = %s.split(',', 0, 1);');`
		`//--> ('a', 'b', 'c', '', 'd', '', 'e')`
		`#m4 = %s.split(',', 1, 0);');`
		`//--> ('a', ' b', 'c',  'd',`

		`' ', ' e')`
		`#m5 = %s.split(',', 0, 0);');`
		
		`//--> ('a', ' b', 'c', '', 'd', ' ', ' e')`
		
		`%s = 'a' + nl() + 'b'; //string with newline`
		`#m = %s.split(nl(), 0); //--> ('a', 'b')`
		`//Note: the 0 argument keeps empty lines`

startswith(s1, s2)	Returns 1 if the string s1 starts with the string s2, else 0. The comparison is case-insensitive.
	Returns: val
		`%s = 'abcde';`
		`%v = %s.startswith('abc');`
		`Returns: 1`

strip(s)	Removes blank characters from the start and end of the string. Returns: string	`%s1 = %s2.strip(); //or: strip(%s1)`
stripstart(s)	Removes blank characters from the start of the string. Returns: string	`%s1 = %s2.striptart(); //or: stripstart(%s1, %s2)`
stripend(s)	Removes blank characters from the end of the string. Returns: string	`%s1 = %s2.stripend(); //or: stripend(%s1, %s2)`

substring(s, start, length)	The piece of the string between character number start and length (these must be integer values).
	
	You can alternatively use a 'slice', using []-notation, see example.
	
	Returns: string
		`%s = %s1.substring(3, 2);`
		`//or: substring(%s1, 3, 2)`
		`%s = %s1[3 .. 5];`
		`//a slice from pos 3 to 5 (both inclusive)`

suffix(s1, s2)	If s1 is a string, it has the string s2 suffixed (appended) Returns: string	`%s1 = %s2.suffix('a');`
upper(s)	The string with upper-case	`%s = upper('aBcD'); Result:`

	letters. Returns: string	`'ABCD'.`

List functions:

Note that some of the functions assume that the lists are lists of strings. This will be fixed regarding values and dates.

Function name	*Description*	*Examples*
[x]-index	Index: picks out a single element. In contrast to R, this does not return a 1-element list containing the variable. If you need that, use for instance #m[3..3]. Returns: var	`#m[3];   //the third element`
[x1..x2]-index	Index: picks out a range of elements. You may omit x1 or x2. Returns: list	`#m[3..5];   //the third to fifth elements`

[x1, x2]-index	For a nested list of lists, `#m[3, 5]` will return the same element as `#m[3][5]`, so this is just convenience to make a nested list accessible like a matrix. See more here.
	Returns: variable
	
	[New in 3.0.6].
		`#m = ((1, 2), (3, 4));`
		`prt #m[2, 1], #m[2][1];   //same`

[x1..y1, x2..y2]-index
[x1..y1, x2]-index
[x1, x2..y2]-index
	For a nested list of lists, `#m[2..3, 2..4]` will select the given 'rows" and "columns", corresponding to selecting a submatrix from a matrix. Beware that in general, `#m[2..3, 2..4]` is completely different from `#m[2..3][2..4]`. See more here.	`//  1  2  3`
		`//  4  5  6`
		`//  7  8  9`
		`// 10 11 12`
		`#m = ((1, 2, 3), (4, 5, 6), (7, 8, 9), (10, 11, 12));`
		`prt #m[2, 2..3];`
		`prt #m[2][2..3]; //same as above`
		`prt #m[2..4, 2]; //matrix-like selection`
		`prt #m[2..4][2]; //different`

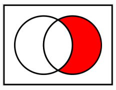
	Returns: list
	[New in 3.0.6].
		`from above!`
		`prt #m[2..4, 2..3]; //matrix-like selection`
		`prt #m[2..4][2..3]; //different from above!`

append(x1, x2)
append(x1, i, x2)
	Adds variable x2 as it is at the end of list x1. Note that if x2 is a list of for instance 3 items, only 1 element is added (the list itself). If you need to add the 3 elements individually, use extend().
	
	If used with i argument, x2 is inserted at index i, instead of at the end. See also extend().
	
	To prepend, use append(x1, 1, x2).
	
	Returns: list
		`#y = #x1.append(#x2);  //or: append(#x1, #x2)`
		`#y = #x1.append(2, #x2);  //insert at position 2`

contains(x1, x2)	Checks if the list of strings x1 contains the string x2. Returns 1 if true, 0 otherwise. You may alternatively use `x2 in x1`, see the last example. See also the count() and index() functions. The comparisons are case-insensitive.
	Returns: val
		`%v = #x1.contains(%s);`
		`if(#x1.contains(%s) == 1); tell 'yes'; end;`
		`if(%s in #x1); tell 'yes'; end;`

count(x1, x2)	Counts the number of times the string x2 is present in the list of strings x1. See also the contains() and index() functions. Note: to obtain the number of elements in a list, use the length() function. The comparisons are case-	`%v = #x1.count(%s);  //or: count(#x1, %s)`

	insensitive. Returns: val	
data(x)	Accepts a string of blank-separated values x and turns them into a list of values. This is handy for long sequences of blank-separated numbers, instead of manually setting the commas. Returns: list	`#m = data('1.0  2.0  1.5');`
dates(x)	Tries to convert each element of the list x to a date. Returns: list	`#y = dates(#x);`

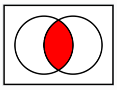
except(x1, x2)

	The except() function subtracts x2 from x1. You may alternatively use the operator -. Only works for lists of strings. See also intersect() and union().
	Was called difference() in Gekko 2.0. See also extend().
	Returns: list
		`#y = #x1.except(#x2);  //or: except(#x1, #x2)`
		`#y = #x1 - #x2;          //same`
		`#y -= #x1;  //subtract from itself`

extend(x1, x2)
extend(x1, i, x2)
	The arguments x1 and x2 must be lists. The function inserts the elements of list x2 one by one at the end of (or at position i in) the list x1. The resulting list may contain dublets.
	For two lists x1 and x2, you may alternatively use the + operator. See also except() and append().
	To pre-extend, use extend(x1, 1, x2).
		`#y = #x1.extend(#x2);  //or: extend(#x1, #x2)`
		`#y = #x1 + #x2; //same as above`
		`#y = #x1.extend(2, #x2);  //insert at position 2`
		`#y += #x1; //add to itself`

	Returns: list	

flatten(x)	For at list x, the function returns a flattened version of the list. For instance, the list `(1, (2, 3))` is transformed into a non-recursive list of non-list elements: `(1, 2, 3)`.
	Returns: list
		`#m1 = (1, (2, 3));`
		`#m2 = #m1.flatten(); //or: flatten(#m1).`

index(x1, x2)	Returns the index of the first occurrence of the string x2 in the list of strings x1. Returns 0 if x2 is not found in x1. See also the count() and contains() functions. The comparisons are case-insensitive. Returns: val	`%i = #x1.index(%s);  //or: index(#x1, %s)`

intersect(x1, x2)

	The intersect() function finds the common elements of the two list of strings x1 and x2. The resulting list will not contain dublets. You may alternatively use the operator `&&`. Only works for lists of strings. See also except() and union().
	Returns: list
		`#y = #x1.intersect(#x2);  //or: intersect(#x1, #x2)`
		`#y = #x1 && #x2;`

length(x)	Returns the number of elements in the list x. You may use len() instead of length().
	Returns: val
		`%v = #x.length();  //or: length(#x).`
		`%v = #x.len(); //the same`

list(x1, x2, ...)	Returns a list of the variables x1, x2, etc. The function is handy for lists with only 0 or 1 elements. See examples.
	Returns: list
		`#m = ();       //will fail`
		`#m = list();  //ok: empty list`
		`#m = (1, 2);  //easy`
		`#m = (1);      //will fail`
		`#m = (1,);     //is ok`
		`#m = list(1); //is ok`

lower(x)	Returns string elements in the list as lower-case. Returns: list	#y = #x1.lower(); //or: lower(#x1)

pop(x1, i)
pop(x1)
	Removes the element at position i in the list x1. Removes the last element if called with pop(x).
	Returns: list
		#y = #x1.pop(2); //or: pop(#x1, 2)
		#y = #x1.pop(); //last element
		#y = #x1.pop(1); //first element

preextend(x1, x2)	Same as extend(x1, 1, x2), putting the elements of x2 in the first position of x1.	#y = #x1.preextend(#x2); //insert at position 1
prefix(x1, x2)	If x1 is a list of strings, each element has the string x2 prefixed (prepended) Returns: list	#y = #x1.prefix(%s); //or: prefix(#x1, %s);
prepend(x1, x2)	Same as append(x1, 1, x2), putting x2 in the first position of x1.	#y = #x1.prepend(#x2); //insert at position 1

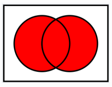
sort(x)
sort(x, 'natural')
	Returns a sorted list of strings, provided that x is a list of strings. Sorting is case-insensitive.
	When the 'natural' argument is used, strings containing numbers will sort naturally, for instance returning a8, a9, a10, instead of a10, a8, a9.
	Returns: list
	Natural sort: [New in 3.1.10].
		#y = #x.sort(); //or: sort(#x)
		#y = #x.sort('natural');

remove(x1, x2)	Removes any string x2 from the list of strings x1. See also the except() function.
	Returns: list
		#y = #x1.remove(%s); //or: remove(#x1, %s);
		#y = #x1 - %s; //also legal

replace(x1, x2, x3)
replaceinside(x1, x2, x3)
replaceinside
	replace(): In the list of strings x1, if this string element is the same as x2, x3 is inserted instead.	#y = #x1.replace(%x2, %x3); //or: replace(#x1, %x2, %x3)
		#y = #x1.replaceinside(%x2, %x3); //or: replace(#x1, %x2,

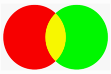
(x1, x2, x3, max)	replaceinside(): the string element has any occurences of x2 inside the string replaced with x3. The replacements may be limited via the max argument. Returns: list	`%x3, 'inside')`
reverse(x)	To be done	
split(x, s)	To be done	
strings(x)	Tries to convert each element of the list x to a string Returns: list	`#y = strings(#x);`
suffix(x1, x2)	If x1 is a list of strings, each element has the string x2 suffixed (appended) Returns: list	`#y = #x1.suffix(%s);  //or: suffix(#x1, %s);`

t(x)	For a nested list of lists, the t() function returns the transpose, similar to transposing a matrix. [New in 3.0.6].
	Returns: list (of lists)
		`#m = ((1, 2), (3, 4));`
		`p #m, t(#m);`

union(x1, x2)

	The union() function finds the union of the two lists. Alternatively use the operator `||`. The resulting list will not introduce dublets, in contrast to the similar `+` operator (simple concatenation). Only works for lists of strings. See also except() and intersect().
	Returns: list
		`#y = #x1.union(#x2);  //or: union(#x1, #x2)`
		`#y = #x1 || #x2;`

unique(x1)	Retains only those elements of list x1 that are unique (list of strings only). Returns: list	`#y = #x1.unique();  //or: unique(#x1)`

upper(x)	Returns string elements in the list as upper-case. Returns: list	`#y = #x1.upper();  //or: upper(#x1)`
vals(x)	Tries to convert each element of the list x to a value Returns: list	`#y = vals(#x);`

venn(x1, x2)	For two lists of strings, this function prints out lists corresponding to a Venn diagram.
	
	That is, intersection and two differences. For instance, `venn(#m1, #m2)` corresponds to printing out the following:
	• `#m1 && #m2`
	• `#m1 - #m2`
	• `#m2 - #m1`
	See also intercept(), except() and union().
	Returns: nothing.
		`#m1 = a, b, c, d, e;`
		`#m2 = c, d, e, f, g;`
		`venn(#m1, #m2);`
		`//Results:`
		`//The following 3 strings are in both lists:`
		`//'c', 'd', 'e'`
		`//The following 2 strings are in list #1, but not in list #2:`
		`//'a', 'b'`
		`//The following 2 strings are in list #2, but not in list #1:`
		`//'f', 'g'`

Bank/name/frequency/index manipulations

Function name	*Description*	*Examples*

addBank(x, bank)	If x does not have a bankname, a bankname is added. The input x may be string or list.
	Returns: string or list
		`%name = addBank('x!q', 'b2');`
		`Result: 'b2:x!q'`

addFreq(x, freq)	If x does not have a freq, a freq is added. The input x	`%name = addFreq('x', 'q');`
		`Result: 'x!q'`

	may be string or list. Returns: string or list	
getBank(x)	Returns the bank part of x. The input x may be series, string or list. Returns: string or list	%bank = getBank('b2:x!q'); Result: 'b2'
getFreq(x)	Returns the freq part of x. The input x may be series, string or list. Returns: string or list	%bank = getFreq('b2:x!q'); Result: 'q'
getFullName(bank, name, freq)	Returns the full name corresponding to the input, where bank, name and freq are strings. Returns: string	%name = getFullName('b2', 'x', 'q'); Result: 'b2:x!q'
getFullName(bank, name, freq, index)	Returns the full name corresponding to the input, where bank, name and freq are strings, and index is a list (of strings) Returns: string	%name = getFullName('b2', 'x', 'q', ('a', 'b')); Result: 'b2:x!q[a,b]'
getIndex(x)	Returns the index part of x. The input x may be string or list. Returns: list	#index = getIndex('b2:x!q[a, b]'); Result: ('a', 'b') If the input is a list, the output will be a list of lists.
getName(x)	Returns the name part of x. The input x may be series, string or list. Returns: string or list	%name = getName('b2:x!q'); Result: 'x'
getNameAndFreq(x)	Returns the name part of x. The input x may be series, string or list. Returns: string or list	%name = getNameAndFreq('b2:x!q'); Result: 'x!q'
removeBank(x)	Removes any bank in x. The input x may be string or list. Returns: string or list	%name = removeBank('b2:x!q'); Result: 'x!q'

removeBank(x, bank)	Removes any banks in x with the indicated bankname. The input x may be string or list.
	Returns: string or list
		%name = removeBank('b2:x!q', 'b2');
		Result: 'x!q'
		%name = removeBank('b3:x!q', 'b2');
		Result: 'b2:x!q'

removeFreq(x)	Removes any freq in x. The input x may be string or list. Returns: string or list	%name = removeFreq('b2:x!q'); Result: 'b2:x'

removeFreq(x, freq)	Removes any freq in x with the indicated freqname. The input x may be string or list.
	Returns: string or list
		%name = removeFreq('b2:x!q', 'q');
		Result: 'b2:x'
		%name = removeFreq('b3:x!q', 'm');
		Result: 'b2:x!q'

removeIndex(x)	Removes any index in x. The input x may be string or list. Returns: string or list	%name = removeIndex('b2:x!q[a, b]'); Result: 'b2:x!q'

replaceBank(x, b1, b2)	Replaces any banks in x having name b1 with name b2. The input x may be string or list.
	Returns: string or list
		%name = replaceBank('b2:x!q', 'b2', 'b3');
		Result: 'b3:x!q'
		%name = replaceBank('b2:x!q', 'b3', 'b4');
		Result: 'b2:x!q'

replaceFreq(x, f1, f2)	Replaces any freq in x having freq b1 with freq b2. The input x may be string or list.
	Returns: string or list
		%name = replaceFreq('b2:x!q', 'q', 'm');
		Result: 'b3:x!m'
		%name = replaceFreq('b2:x!q', 'm', 'a');
		Result: 'b3:x!q'

setBank(x, bank)	The indicated bankname is set, even if it exists already. The input x may be string or list. Returns: string or list	%name = setBank('b3:x!q', 'b2'); Result: 'b2:x!q'
setFreq(x, freq)	The indicated freq is set, even if it exists already. The input x may be string or list. Returns: string or list	%name = setFreq('b2:x!q', 'm'); Result: 'b2:x1!m'

setName(x, name)	The indicated name is set. The input x may be string or list.
	Returns: string or list
		`%name = setName('b2:x!q', 'y');`
		`Result: 'b2:y!m'`

setNamePrefix(x, p)	The name of x has prefix p added.The input x may be string or list.
	Returns: string or list
		`%name = setNamePrefix('b2:x!q', 'a');`
		`Result: 'b2:ax!m'`

setNameSuffix(x, s)	The name of x has suffix s added.The input x may be string or list.
	Returns: string or list
		`%name = setNameSuffix('b2:x!q', 'b');`
		`Result: 'b2:xb!m'`

Timeseries functions

Function name	*Description*	*Examples*

allMiss(x)	On a timeseries x, the function returns 1 if all observations are missing values ("no data at all"), else it returns 0. See also `isMiss()`, and `fromSeries('dataStart')`, and `fromSeries('dataStart')`.
	
	[New in Gekko 3.1.20].
		`time 2021 2023;`
		`x1 = m();`
		`prt x1.allMiss();  //1`
		`x1 <2022 2022> = 1;`
		`prt x1.allMiss();  //0`

arrayPack(name, lname)	This function finds all non-array ("normal") timeseries in the first-position databank and puts them into a single 1-dimensional array-series called `name`. In addition, a list `lname` is created, containing the names of the "packed" timeseries.
	
	Note that after the function is called, the first-position
		`read adambank.gbk;`
		`arraypack('adam', '#adamvars');`
		`write <gdx> adambank.gdx;`
		`//Creates the 1-dimensional array-series adam,`
		`//with elements corresponding to the normal timeseries`
		`//inside adambank.gbk. For instance, adambank.gbk may`
		`//contain the series fY, which is mirrored in the element`
		`//adam[fY]. Also, the list #adamvars will contain the`
		`//element 'fY' (and more).`

	databank only contains the array-series and the list. See also `arrayUnpack()`.
	[New in 3.1.16]
		`//When writing the gdx file, the adam variable will`
		`//have domain names 'adamvars' and 't'.`
		`//(You may use arraypack('adam', '*'), in which case`
		`//the Gekko list is not created).`

arrayUnpack (name)	This function takes the 1-dimensional array-series called `name` from the first-position databank and transforms ("unpacks") all the elements of the array-series into normal timeseries. The function deletes the array-series before it returns. See also `arrayPack()`.
	[New in 3.1.16]
		`read <gdx> adambk.gdx; //has adam variable inside`
		`arrayunpack('adam');`
		`//Now the first-position databank contains normal`
		`//timeseries corresponding to the elements of the`
		`//array-series adam.`

bankFlatten(t)
bankFlatten(bank, t)
	For the global (or locally given) period, all series and array-series are flattened. This means that over the global (or local) period, all values are set to their value in `t`. A bank name may be provided: if not, the first-position databank is used.
	[New in 3.1.16]
		`time 1960 2022;`
		`bankFlatten(2000);`
		`//All series and array-series are set to`
		`//their 2000 value, over the global period`
		`//1960-2022.`
		`bankFlatten(<1970 2010>, 2000);`
		`//same, but now flattened over 1970-2010.`
		`bankFlatten('Ref', 2000);`
		`//for the Ref databank`

bankReplace (x1, x2)
bankReplace (bank, x1, x2)
	For the global (or locally given) period, in all series and array-series, the value `x1` is replaced with `x2`. A bank name may be provided: if not, the first-position databank is used.
	[New in 3.1.16]
		`time 1960 2022;`
		`bankReplace(m(), 0);`
		`//All series and array-series have missing value replaced with 0, over the global period 1960-2022.`
		`bankReplace(<1970 2010>, m(), 0);`
		`//same, but now over 1970-2010.`
		`flatten('Ref', m(), 0);`
		`//for the Ref databank`

binary(x)	Converts a table-like nested list of strings into a corresponding array-series where the present elements (that is, the 'rows' of the nested list) are represented with value 1. The nested list must contain time info as the last element of each sublist. The function can be used to tranform a multidimensional GAMS set into a corresponding array-series. This can be convenient for printing, etc.
	[New in 3.1.14]
		`time 2001 2002;`
		`#m = (('a', 'b', '2001'), ('a', 'c', '2002'));`
		`x1 = #m.binary();`
		`x2 = #m.binary().replace(m(), 0); //set missing to 0`
		`prt #m; //try also prt <view> #m;`
		`prt <n> x1;  //1 or missing`
		`prt <n> x2;  //1 or 0`
		
		`//          x1[a, b]    x1[a, c]`
		`// 2001       1.0000           M`
		`// 2002            M      1.0000`
		`//`
		`//          x2[a, b]    x2[a, c]`
		`// 2001       1.0000      0.0000`
		`// 2002       0.0000      1.0000`

collapse(x)
collapse(x, method)
collapse(x, freq)
collapse(x, freq, method)
	Function variant of the COLLAPSE statement, converting from higher to lower frequency. Note possible differences compared to COLLAPSE regarding the use of databanks and time periods.
	
	You may input a frequency (freq), choose from `'a'`, `'q'`, `'m'`, `'w'`. If frequency is not indicated, the function will per default collapse from `d` to `m`, `w` to `m`, `m` to `q`, or `q` to `a`.
	
	The method argument can be `'total'`, `'avg'`, `'first'`, `'last'`, `'strict'` or `'flex'`. The last two of these indicate how missing values are handled. You may combine the first four and the last two of these with `-`, for
		`time 2020 2020;`
		`option freq q;`
		`x = 1, 3, 2, 6;`
		`p <n> x, x.collapse();`
		`//same: x.collapse('total')`
		`//same: x.collapse('a')`
		`//same: x.collapse('a', 'total')`
		
		`//Result:`
		`//                    x  x.collapse()`
		`// 2020`
		`// q1            1.0000`
		`// q2            3.0000`
		`// q3            2.0000`
		`// q4            6.0000`
		`// a                          12.0000`
		
		`//Nested collapse, d --> m --> q --> a`
		`time 2020 2025;`
		`option freq d;`
		`x!d = 1;`
		`p <n>`

	instance `'avg-flex'` to choose method `'avg'` combined with missing handling `'flex'`. Note that missing handling 'flex' is currently only available for collapse of daily (`!d`) series. Regarding these options, see the COLLAPSE statement.
	You may use `option collapse method = ... ;` and `option collapse missing d = ... ;` to control default values for the method and missing handling. See OPTION.
	Returns: series.
	[New in 3.1.11]
		`x`
		`!`
		`d`
		`.collapse().collapse().collaps`
		`e();`
		
		`//           x!d.collap`
		`//        se().collapse`
		`//        ().collapse()`
		`// 2020          366.0000`
		`// 2021          365.0000`
		`// 2022          365.0000`
		`// 2023          365.0000`
		`// 2024          366.0000`
		`// 2025          365.0000`
		
		`//x!d.collapse('a') could be used instead`

flatten(x, d)	Transforms 1 array-timeseries *x* into *n* normal timeseries with names reflecting the array-series dimensions and delimiter corresponding to the string argument *d*. For instance, an array-series `npop` defined over the two dimensions `#sex` and `#age` could contain the element (sub-series) `npop[m, 40]`, which `flatten()` would turn into the normal series `npop_m_40`.
	[New in 3.1.16]
		`npop = series(1);`
		`npop[m, 40] = 60000;`
		`npop[f, 40] = 62000;`
		`npop[m, 41] = 59000;`
		`npop[f, 41] = 61000;`
		`npop.flatten('_');`
		`delete npop;`
		`p <n dec=0 width=10> {'*'};`
		
		`//Output:`
		`//         npop_f_40    npop_f_41`
		`  npop_m_40    npop_m_41`
		`//2014          62000          61000`
		`       60000         59000`
		`//2015          62000          61000`
		`       60000         59000`
		`//...`

getDomains(x)	Returns a list of strings (with the number of elements corresponding to the number of dimensions) containing the domain for each dimension of the array-series x, for instance `'#i'` or `'*'` (the latter	`#d = getdomains(x); //or: #d = x.getdomains();`

	means no domain). See also `getFixType()`, `setFixType()`, `setDomains()`. Returns: list	

getElements(x)	Returns a nested list of strings, which for each dimension of the array-series shows which elements occur (in all sub-series). In the example on the right, the first dimension only contains `a`'s, whereas the second dimension contains `b` and `c`. The returned list has as many elements as the dimensionality of the array-series. See also `subseries()`.
	[New in 3.1.10]
		`x = series(2);`
		`x[a, b] = 1;`
		`x[a, c] = 2;`
		`prt x.getelements();`
		`// ('a',), ('b', 'c')`
		`prt x.getelements()[2];`
		`// 'b', 'c'`
		`prt x.getelements()[2][1];`
		`// 'b'`

getFixType(x)	Returns the "fix" type of an array-timeseries `x`, where the return value can be `'variable'` or `'parameter'`. Array-series imported from GAMS gdx can be of variable or parameter type. See also `setFixType()`, `getDomains()`, `setDomains()`. Returns: string. [New in 3.1.14]	`//See example under setFixType()`

hpfilter(x, lambda)
hpfilter(<t1 t2>, lambda)
hpfilter(x, lambda, log)
hpfilter(<t1 t2> x, lambda, log)
	Returns a HP-filtered version of series x. Lambda is normally 6.25 for annual, 1600 for quarterly, and 129600 for monthly series. An additional argument 0 or 1 may be added (1 if log-transforms are to be used inside the	`y = hpfilter(x, 6.25);`
		`y = hpfilter(x, 6.25, 1);  //log-transforms`
		`y = hpfilter(<1970 2015>, x, 6.25);`
		`y = x.hpfilter(<1970 2015>, 6.25); //alternative syntax`

	calculation). Time period may be indicated with t1-t2. Returns: series	

interpolate(x)
interpolate(x, method)
interpolate(x, freq)
interpolate(x, freq, method)
	Function variant of the INTERPOLATE statement, converting from lower to higher frequency. Note possible differences compared to INTERPOLATE regarding the use of databanks and time periods.
	
	Method must be `'total'` or `'avg'` (optionally combined with Olsette/Cholette/Denton, for instance `'total-olsette'` or `'avg-cholette'`).
	
	Destination freq must be `'q'`, `'m'`, `'w'`, or `'d'`. Per default, interpolate() will transform from `a` to `q`, `q` to `m`, `m` to `d` and `w` to `d`.
	
	Returns: series.
	
	[New in 3.1.11]
		`time 2020 2021;`
		`x = 1, 3;`
		`p <n> x, x.interpolate();`
		
		`//`
		`x.`
		`// x`
		`interpolate()`
		`// 2020`
		
		`// q1`
		`1.0000`
		`// q2`
		`1.0000`
		`// q3`
		`1.0000`
		`// q4`
		`1.0000`
		`// a 1.0000`
		
		`//`
		
		`// 2021`
		
		`// q1`
		`3.0000`
		`// q2`
		`3.0000`
		`// q3`
		`3.0000`
		`// q4`
		`3.0000`
		`// a 3.0000`

isArraySeries(x)	Returns 1 if the variable is an array-series, and 0 otherwise.
	
	[New in 3.1.14]
		`x = series(1);`
		`x[a] = 100;`
		`prt isArraySeries(x);`

isTimelessSeries(x)	Returns 1 if the variable is a timeless series, and 0 otherwise.
	
	[New in 3.1.14]
		`x = timeless(100);`
		`prt isTimelessSeries(x);`

laspchain (plist, qlist, t)
laspchain(<t1 t2>, plist, qlist, t)
laspchain(<t1 t2>, plist, qlist, t, option)
 	
laspchain (vals, valsLag, t)
laspchain(<t1 t2>, vals, valsLag, t)
	Laspeyres chain index for quantities. Can be used with either (a) two lists of strings, or (b) two timeseries. In both cases the result is a map containing two consistent aggregated series `p` and `q` (price and quantity). The price is set = 1 in the t period. A period can be indicated in the `<t1 t2>` field. The function is robust regarding missing values at the start or end of the time period.
	
	(a): Two lists of strings (names)
	
	The names inside the two lists indicate the series that are to be aggregated.
	
	See also the bottom of the LIST help page.
	
	(b): Two series (accumulation)
	
	A different way to use the function is to input one series (vals) with values in current prices, and one series (valsLag) with values in the previous period's prices. From this, Gekko will compute the aggregate price and quantity.
	
	If you need to fill out holes (missings) between non-missings, for instance in prices series, you can use the smooth() function or SMOOTH statement.
	
	Quarterly data
		`time 2017 2020;`
		`p1 = 1.02, 1.03, 1.04, 1.05;`
		`p2 = 1.12, 1.13, 1.14, 1.15;`
		`q1 = 2, 3, 4, 5;`
		`q2 = 12, 13, 14, 15;`
		`#p = p1, p2;`
		`#q = q1, q2;`
		`//(a): With lists of strings`
		`#a = laspchain(#p, #q, 2020);`
		`prt #a.p, #a.q;`
		`v = p1*q1 + p2*q2;`
		`vlag = p1[-1]*q1 + p2[-1]*q2;`
		`//(b): With series (accumulation)`
		`#b = laspchain(v, vlag, 2020);`
		`prt #b.p, #b.q;`
		
		`//With explicit time indication:`
		`//laspchain(<2017 2020>, #p, #q, 2020);`
		`//laspchain(<2017 2020>, v, vlag, 2020);`

	For quarterly data, using two lists of strings, option `'annualoverlap'` must be indicated. When quarterly data are input, the so-called annual overlap method is used, where the algorithm starts out collapsing the quarterly data into annual data, then computes a "normal" annual chain index for the aggregate price, and lastly uses this annual price index to compute the quarterly indexes. Returns: map	
laspfixed (plist, qlist, t) laspfixed(<t1 t2>, plist, qlist, t) laspfixed(<t1 t2>, plist, qlist, t, option)	Laspeyres fixed-price index. As `laspchain()`, but with fixed prices. Cannot be used with series input. Returns: map	`//See example regarding laspchain()`

percentile(x, %v)	Computes the %v percentile for the series x, for the global time period. Any missing values within that sample are ignored. Setting %v = 0.5 results in the median.
	Returns: val
		`%z = percentile(y, 0.25);`
		`%z = percentile(y, 0.50);`

rebase(x, t)
rebase(x, t, index)
	Function variant of the REBASE statement. Note possible differences compared to REBASE regarding the use of databanks and time periods.	`time 2020 2022;`
		`x = 10, 11, 12;`
		`prt <n> x, x.rebase(2021),`
		`x.rebase(2021, 1);`
		`//`
		`      x            x.re`
		`//                    x`
		`.rebase(2021)   base(2021, 1)`
		`// 2020          10.0000`
		`90.9091           0.9091`

	You cannot use start/end dates with this function, only a single date (alternatively, use REBASE statement).
	Returns: series.
	[New in 3.1.11]
		`// 2021          11.0000`
		`100.0000          1.0000`
		`// 2022          12.0000`
		`109.0909          1.0909`

rename(x, cfg)	For an array-series *x*: renames and reorders dimension elements, according to a *cfg* nested list. [New in 3.1.16]	TODO

reorder(x, ord)	For array-series *x*, this function reorders the dimensions according to the list *ord*. The list must contain the integers 1..*n*, where *n* is the dimensionality of the array-series.
	Note: for an array-series dimension of size *n*, if `reorder()` is fed with the list `(1, 2, 3, ... , n)`, the function returns an identical array-series.
	[New in 3.1.16]
		`x = series(3);`
		`x[a, c, e] = 1; x[a, c, f] =`
		`2;`
		`x[b, d, e] = 3; x[b, d, f] =`
		`4;`
		`y = x.reorder((2, 3, 1));`
		`prt y;`
		`//Will print subseries y[c, e,`
		`a], y[c, f, a],`
		`//y[d, e, b] and y[d, f, b].`
		`The first dimension of x`
		`//has been moved to the last`
		`dimension of y, and`
		`//the other two dimensions are`
		`moved forwards.`

replace(x, v1, v2)	For the series or array-series x, the function replaces the value v1 with the value v2, over the given sample.
	See also `iif()`, `isMiss()`, and the $-conditional.
	Returns: series
		`time 2001 2003;`
		`x = (1, m(), 3);  //m() is`
		`missing value`
		`//result is (1, 0, 3):`
		`y = x.replace(m(), 0);  //or:`
		`replace(x, m(), 0)`
		`//the replacement is done for`
		`the sample:`
		`z <2000 2004> = x.replace(m(),`
		`0);`
		`//result is (0, 1, 0, 3, 0)`

rotate(x, d)	Transforms the array-series x to a new array-series, where the time dimension and dimension number d swap places. For instance, the array-series pop may contain sub-series for each age group 0 to 100, that is, pop['0'], pop['1'], ... , pop[100']. These 101 sub-series are all defined over a time period, say 2020-2050. Then profile = rotate(pop, 1) will be a new array-series containing sub-series for each time period 2020 to 2050, that is, profile['2020'], profile['2021'], ... , profile['2050']. These 31 sub-series are all defined over an undated time period 0 to 100, corresponding to the age dimension. Hence, `plot <0u 100u> profile['2020'];` will plot the age profile of the population in the year 2020.
	See example under PLOT (age profiles section).
	Returns: series
		`//pop is a 1-dimensional array-series`
		`//with ages 0-100 in its dimension.`
		`//note: 'u' indicates undated frequency`
		
		`profile!u = rotate(pop, 1);`
		`#t = ('2020', '2030', '2040', '2050');`
		`plot <0u 100u> profile!u[#t];`

series(freq, n)	Constructs a series or array-series of the given frequency and with the given dimensions (n). You can skip some of these options, see examples.
	
	See also isArraySeries().
	
	Returns: series
		`x = series('q', 3);  //quarterly array-series, 3-dim`
		`x = series('q');  //quarterly normal series`
		`x = series(3);  //3-dim array-series with current frequency`
		`x = series();  //normal series with current frequency`
		
		`y = series(1);`
		`y['a'] = 100;  //or: y[a] = ...`

setDomains(x, d)	Uses a list of strings (d) to set the domains for each dimension. Use `'*'` to indicate a domain-less dimension (in GAMS called the universal set), but otherwise each element should start with symbol `#` (to indicate that it is a list). See also `getFixType()`, `setFixType()`, `getDomains()`.
	Returns: nothing
		`#d = ('#b', '#x');`
		`setdomains(x, #d);  //or: x.setdomains(#d);`
		`y.setdomains(('*', '#x'));`

setFixType(x, type)	Sets the "fix" type of an array-timeseries `x`, where type can be `'variable'` or `'parameter'`. When [exporting](#) an array-series to a GAMS gdx file, this difference is of significance. See also `getFixType()`, `getDomains()`, `setDomains()`.
	
	[New in 3.1.14]
		`x = series(1);`
		`x[a] = 100;`
		`p x.getFixType();`
		`//returns: 'variable', which is default`
		`write <gdx> data1; //variable in GAMS`
		`x.setFixType('parameter');`
		`p x.getFixType();`
		`write <gdx> data2; //parameter in GAMS`

smooth(x)
smooth(x, method)
smooth(x, y)
smooth(x, method, y)
	Function variant of the [SMOOTH](#) statement to fill holes (missing values). Note possible differences compared to SMOOTH regarding the use of databanks and time periods.
	
	Method must be `'linear'`, `'geometric'`, `'repeat'`, `'spline'`, or `'overlay'`. If method is not stated, `'linear'` is default. The y variable can be a series, value or 1x1 matrix (it is only used for 'overlay' smoothing).
	
	Returns: series.
		`z = x.smooth('linear');     //user linear smoothing`
		`z = x.smooth();             //short for the above`
		`z = x.smooth('overlay', y); //use y series to fill holes`
		`z = x.smooth(y);            //short for the above`
		`z = x.smooth(0);            //use 0 value as overlay`

	[New in 3.1.13]	

splice(x1, x2, ...)
splice(x1, t1, x2, ...)
splice(x1, t1, t2, x2, ...)
splice(type, x1, ...)
	Function variant of the SPLICE statement to splice two or more overlapping timeseries into one resulting timeseries. Note possible differences compared to SPLICE regarding the use of databanks and time periods.
	
	Type can be `'rel1'`, `'rel2'`, `'rel3'`, `'abs'` for the method, or `'first'`, `'last'`, `'1'`, `'2'`, `'3'`, etc. for indicating the "primary" series number (such numbers must be strings). You may combine the method and "primary" indication with `-`, for instance `'rel3-first'` or `'abs-2'`. Default is `'rel1-last'`. See many more details under SPLICE.
	
	Returns: series.
	
	[New in 3.1.15]
		`time 2001 2005;`
		`x1 <2001 2003> = 1, 2, 3;`
		`x2 <2003 2005> = 11, 12, 13;`
		`y = splice(x1, x2);`
		`prt y, x1, x2;`
		
		`//Regarding the left-hand side series,`
		`//note that splice() never inserts`
		`//values outside of the given`
		`//local or global time period.`
		`//So this time period is respected,`
		`//in contrast to the SPLICE statement.`

subseries(x, option)	Helper function for array-series.
	
	• option = `'elements'`: nested list of strings with element combinations for each sub-series.
	• option = `'names'`: list of sub-series names (as strings). You may remove the frequency part of these names with `removefreq()`, see example.
		`x = series(2);`
		`x[a, b] = 1;`
		`x[a, c] = 2;`
		`prt x.subseries('elements');`
		`// ('a', 'b'), ('a', 'c')`
		`prt x.subseries('names');`
		`// 'x!a[a, b]', 'x!a[a, c]'`
		`prt`
		`x`
		`.subseries`
		`('names').removefreq();`
		`// 'x[a, b]', 'x[a, c]'`
		`prt x.subseries('length');`
		`// 2`
		`prt x.subseries('dimensions');`
		`// 2`

	• option = `'length'`: the number of sub-series inside the array-series. • option = `'dimensions'`: the number of dimensions of the array-series. See also `getelements()`. [New in 3.1.10]	

timeless(freq, v)	Constructs a timeless series of the given frequency, to value v. You can skip some of these options, see examples.
	In many cases, you can just use a value scalar with the same functionality. But timeless series can be practical, for instance they can be used as array-series.
	See also isTimelessSeries().
	Returns: series
		`x = timeless('q', 3);  //quarterly timeless series, with value = 3.`
		`x = timeless('q');  //quarterly timeless series, no value set.`
		`x = timeless(3);  //timeless series with current frequency, with value = 3.`
		`x = timeless();  //timeless series with current frequency, no value set.`
		`y = series(1);`
		`y['a'] = timeless(100);  //or: y[a] = ...`

Time, databank and other environment info

Function name	*Description*	*Examples*

bankfilename(s)
bankfilename(s, p)
	OBSOLETE: use `fromBank(%s, 'filename')` or `fromBank(%s, 'fullpath')`.	

bankname(s)	Returns the name of the bank. Input can be the string 'first' or 'ref', or a val	`%b1 = bankname('first');`
		`%b0 = bankname('ref');`
		`%b2 = bankname(2);`

	designating the number in the databank list. Returns: string	

currentDateTime()	Returns current date and time.
	Returns: string
		`%s = currentDateTime();`
		`//Returns: '15-09-2014 12:34:58' (for instance).`
		`tell currentDateTime();`

currentDate()
currentDate2()
	Returns current date, either as a string, or as a date with daily frequency. [New in 3.1.1]
	• currentDate() returns: string
	• currentDate2() returns: daily date. You may convert to other frequencies with the `date()` function, cf. examples.
		`%s = currentDate();`
		`//Returns: '15-09-2014' (for instance).`
		`%d = currentDate2();          //daily`
		`%m = currentDate2().date('m'); //monthly`
		`%a = currentDate2().date('a'); //annual`

currentDay()	Returns the current day (between 1 and 31). [New in 3.0.5]. Returns: val	`%v = currentDay();`
currentFolder()	Returns the current working folder (cf. `option folder working = ...`). [New in 3.0.6]. Returns: string.	`%s = currentFolder();`

currentFreq()	Returns the current frequency, for instance `a`, `q`, `m`, `d` or `u`.
	Returns: string
		`%s = currentFreq();`
		`//Returns: 'a' (depending upon frequency setting, cf. option freq).`

currentHour()	Returns the current hour. [New in 3.0.5]. Returns: val	`%v = currentHour();`
currentMinute()	Returns the current minute. [New in 3.0.5]. Returns: val	`%v = currentMinute();`

currentMonth()	Returns the current month. [New in 3.0.5]. Returns: val	`%v = currentMonth();`

currentPerStart()	Returns the start of the global time period.
	Returns: date
		`%s = currentPerStart();`
		`//Returns: 2012q1 (for instance).`

currentPerEnd()	Returns the end of the global time period.
	Returns: date
		`%s = currentPerStart();`
		`//Returns: 2015q4 (for instance).`

currentSecond()	Returns the current second. [New in 3.0.5]. Returns: val	`%v = currentSecond();`

currentTime()	Returns current time.
	Returns: string
		`%s = currentTime();`
		`//Returns: '12:34:58' (for instance).`

currentYear()	Returns the current year. [New in 3.0.5]. Returns: val	`%v = currentYear();`

exist(s)	Returns 1 if the variable corresponding to the string *s* exists, else 0. **The input must be of string type.** For timeseries, you do not have to add frequency to the name (for instance `!q`), if the series is of current frequency. The function respects the `option databank search` setting (that is, in sim-mode it will only look in the first-position databank, if a databank name is not provided).
	
	The `exist()` function can also check if an array-subseries exists, cf. example.
	
	Returns: val
		`x = 1;`
		`prt exist('x');`
		`prt exist('work:x');`
		
		`y = series(2);`
		`y[a, a] = 1; y[a, b] = 2; y[b, b] = 4;`
		`prt exist('y[a, a]'); prt exist('y[b, a]');`
		`#i = y.getelements()[1];`
		`#j = y.getelements()[2];`
		`for string %i = #i;`
		`  for string %j = #j;`
		`    if(exist('y[{%i},{%j}]'));`
		`      y[%i,%j] *= 100;`
		`    end;`
		`  end;`
		`end;`
		`prt y;`

filteredperiods(d1, d2)	Returns the number of filtered periods between `d1` and `d2` (these are dates). Returns: val	`%v = filteredperiods(%d1, %d2);`
fromDatabank(x, type)	Accesses meta-information from the databank x (string). Type (string) can be: - `'filename'` (only file, returns string) - `'fullpath'` (file and path, returns string) - `'label'` (cf. HDG, returns string) - `'stamp'` (last time written, returns string) - `'stamp2'` (last time written, returns daily date) - `'format'` ("1.0"/"1.1" or "1.2", returns string) - `'count'` (number of variables, returns val) [New in 3.1.15].	- TODO: EXAMPLES

fromSeries(x, type)	Accesses meta-information from the timeseries x. Type can be
	- `'name'` (returns string)
	- `'bank'` (returns string)
	- `'freq'` (returns string)
	- `'label'` (returns string)
	- `'source'` (returns string)
	- `'units'` (returns string)
	- `'stamp'` (returns string)
	- `'stamp2'` (returns date)
	- `'dataStart'` or `'dataEnd'` (returns date: period with actual data). See also `isMiss()` and `allMiss()`.
	- `'dataStartTruncate'` or `'dataEndTruncate'` (ret
		`%s = ref:gdp.fromSeries('label'); //Returns 'Gross domestic product' (for instance).`
		`%d = gdp.fromSeries('dataStart'); //Returns 1980q1 (for instance). Same logic regarding 'dataEnd' argument.`
		`%s = gdp.fromSeries('freq'); //Returns 'q' (for instance).`
		
		`time 2011 2015;`
		`x = 1, 2, 3, 4, 5;`
		`time 2013 2017;`
		`prt x.fromSeries('dataStartTruncate'); //2013`
		`time 2011 2015;`
		`prt x.fromSeries(<2013 2017>, 'dataStartTruncate'); //2013`

	urns date: period with actual data, truncated with global or local time period). May return null/empty. [New in 3.1.6] The x argument can be either a series name or string.	

gekkoBitness()	Returns bitness of Gekko (either "32" or "64"). Returns: string. [New in 3.1.11]	`tell 'Gekko {gekkoVersion()} ({gekkoBitness()}-bit)';`
		`//Gekko 3.0.1 (64-bit) --> for instance`

gekkoInfo(x)	Returns a string with info on Gekko. Choose between:
	• 'short1' = Gekko-version
	• 'short2' = Gekko-version + bitness
	• 'short3' = Gekko-version + bitness + period
	• 'short4' = Gekko-version + bitness + working folder
	• 'short5' = Gekko-version + bitness + period + working folder
	Returns: string. [New in 3.1.11]
		`tell gekkoInfo('short1');`
		`//Gekko 3.0.1`

gekkoVersion()	Returns the Gekko version number (`'xx.yy.zz'`). See also `gekkoVersionDate()`. Returns: string	`tell 'Gekko ' + gekkoVersion();`
		`//Gekko 3.1.16     (for instance)`

gekkoVersionDate()	Returns the daily date corresponding to the `gekko.exe` file of the currently executing Gekko. Beware the file systems, version control systems, etc. may tamper with file dates (like the date of	`tell 'Gekko ' + gekkoVersion() + ' ' + gekkoVersionDate();`
		`//Gekko 3.1.16 2024m3d11 (for instance)`

	`gekko.exe`). See also `gekkoVersion()`. Returns: daily date	
getEndoExo()	Returns a list with names of those variables that start with 'endo_' or 'exo_'. This is used with GAMS models, when fixing equations. Returns: list of strings.	`#m = getEndoExo();`
isLibraryLoaded(x)	Returns 1 if the library with the name x (a string) is loaded, and 0 otherwise. Returns: val	`%v = islibraryloaded('lib1');`
isOpen(x)	Returns 1 if the databank with the name x (a string) is open, and 0 otherwise. Returns: val	`%v = isopen('mybank');`

root()
root('root')
root('gekko')
	Returns the name of the root folder as a string (without any closing backslash). The root folder is defined as a folder that contains a file with the name `root.ini` or `gekko.ini` (`root()` or `root('root')` will look for `root.ini`, whereas `root('gekko')` will look for `gekko.ini`).
	To find the root directory, Gekko first looks for a `root.ini`/`gekko.ini` file in the working folder. If not found, it looks in the parent folder of the working folder, and so on upwards. If no `root.ini`/`gekko.ini` file is found, or if more than one is found, Gekko aborts with an error. **NOTE**: the
		`open g:\update\data\scenario5\bank;  //hard-coded root`
		`open {root()}\data\scenario5\bank;   //using root()`
		`// In the latter OPEN, it is presupposed that the`
		`// folder g:\update contains a file with the`
		`// name root.ini. The contents of this file has`
		`// no significance, but could contain a remark that`
		`// the file should not be deleted or moved.`
		`// If the system of folders inside g:\update is`
		`// later on moved to a new location, for instance`
		`// to g:\update2, the latter OPEN statement still works`
		`// in the new location.`
		`// You could alternatively use relative paths to`
		`// the same effect, but then you would have to use`

	root.ini/gekko.ini file must also be consistent with the folder where the running gcm file is residing, else an error is issued (change the working folder if this is an issue). This is to avoid confusion.
	
	Using this root function makes it easier to refer to files inside a larger folder structure, without hard-coding the name of the root folder. This also makes it easier to copy or move a folder structure. See also rootify(). [New in 3.1.13].
		// paths like open ..\..\..\data\scenario5\bank;.
		
		//To use a gekko.ini file instead of
		//root.ini as root, use:
		open {root('gekko')}\data\scenario5\bank;
		
		//The contents of a root.ini file could be for instance:
		// Do not delete or move. Used by the Gekko root()
		// function to provide an "anchor" for relative paths.
		
		tell root(); //to see what the current root is

rootify(folder, rootpath)	This function can be used to replace path names with {root()} in a sysem of .gcm files. This can be useful if you are using a system with a lot of hardcoded paths that you want to make easier to copy or move. The folder argument designates the folder (including subfolders) for which rootify() is done. The rootpath argument is the path you want to have replaced with {root()}.
	
	For textfiles other than .gcm Gekko will also look at these and report if any contain the rootpath. Gekko will not alter file dates in rootify().
	
	See also root().
	
	[New in 3.1.19].
		//Example: consider a gcm file residing in the folder
		//g:\common\data:
		
		  read g:\common\data\b1.gbk;
		  x1 = x1 + 1;
		  write g:\common\data\b2.gbk;
		  tell 'created g:\common\data\b2.gbk';
		
		//After rotify('g:\common\data', 'g:\common\data')
		//the file looks like this:
		
		  read {root()}\b1.gbk;
		  x1 = x1 + 1;
		  write {root()}\b2.gbk;
		  tell 'created {root()}\b2.gbk';
		
		//For this to work, you need to place a
		//g:\common\data\root.ini (with arbitrary contents)

time()
time(<t1 t2>)
	Returns the current time period as a series where the dates are represented as values. Works with quarters and months, too. The function may for instance be practical for creating trend variables.
	
	If you need a quarterly or monthly time() to average-collapse into an annual time(), you should use `time()-0.5`.
	
	Returns: series
		`option freq q;`
		`time 2010q1 2011q4;`
		`p time();`
		`p time(<2010q3 2011q3>); //truncated`
		`//the first one prints 2010.125, 2010.375, 2010.625, 2010.875, ...`

Matrix functions:

Function name	*Description*	*Examples*
avgc(x)	Average over cols. Returns: matrix	`#m2 = avgc(#m1);`
avgr(x)	Average over rows Returns: matrix	`#m2 = avgr(#m1);`

chol(x)
chol(x, type)
	Cholesky decomposition of matrix x. Accepts type (string), either 'upper' or 'lower'.
	Returns: matrix
		`#m2 = chol(#m1, 'upper');`

cols(*x*)	Returns the number of colums of *x* Returns: val	`%v = cols(#m);`

design(x)	Returns a "design" matrix, equivalent to the same Gauss function. The function is practical for aggregating rows or columns.	`#a = [3; 1; 1; 3; 2];`
		`#m = design(#a);`
		`#x = ones(4, 5);`
		`p #m, #x * #d;`
		
		`#m`
		`1`
		`2 3`

	Input: is a n x 1 column matrix Returns: n x k matrix of 0's and 1's. The input numbers specify the columns in which the 1's should be placed. In the example, a 4 x 5 matrix #x of 1's is defined, and by means of the aggregation matrix #m, it is aggregated from size 4 x 5 to 4 x 3. In this case, the new column 1 is the old columns 2 and 3, the new column 2 is the old column 5, and the new column 3 is the old columns 1 and 4.	1 0.0000 0.0000 1.0000 2 1.0000 0.0000 0.0000 3 1.0000 0.0000 0.0000 4 0.0000 0.0000 1.0000 5 0.0000 1.0000 0.0000 #x * #d 1 2 3 1 2.0000 1.0000 2.0000 2 2.0000 1.0000 2.0000 3 2.0000 1.0000 2.0000 4 2.0000 1.0000 2.0000
det(x)	Determinant of a matrix. Returns: val	%v = det(#m);
diag(x)	Diagonal. If x is a n x n symmetric matrix, the method returns the diagonal as a n x 1 matrix. If x is a n x 1 column vector, the method returns a n x n matrix with this column vector on the diagonal (and zeroes elsewhere). Returns: matrix	#m2 = diag(#m1);
divide(x1, x2)	Element by element division of the two matrices. If x2 is a row vector, each x1 column will be divided with the corresponding value from the row vector. And if x2 is a column vector, each x1 row will be divided with the corresponding value from the column vector. Returns: matrix	#x = divide(#x1, #x2);

i(n)	Returns a n x n identity matrix. Returns: matrix	#m = i(10);
inv(x)	Inverse of matrix x Returns: matrix	#m2 = inv(#m1);
maxc(x)	Max over cols Returns: matrix	#m2 = maxc(#m1);
maxr(x)	Max over rows Returns: matrix	#m2 = maxr(#m1);
minc(x)	Min over cols Returns: matrix	#m2 = minc(#m1);
minr(x)	Min over rows Returns: matrix	#m2 = minr(#m1);
m(r, c) or miss(r, c)	Returns a n x k matrix filled with missing values. Cf. also m() function for values. Returns: matrix	#m = m(5, 10);
multiply(x1, x2)	Element by element multiplication of the two matrices. If x2 is a row vector, each x1 column will be multiplied with the corresponding value from the row vector. And if x2 is a column vector, each x1 row will be multiplied with the corresponding value from the column vector. Returns: matrix	#x = multiply(#x1, #x2);
ones(n, k)	Returns a n x k matrix filled with 1's Returns: matrix	#m = ones(5, 10);

pack(v1, v2, ...)
pack(<t1 t2>, v1, v2, ...)
	Using period t1-t2, the timeseries v1, v2, ... are packed into a n x k matrix, where n is the number of observations and k is the	#m = pack(<2020 2030>, x, y, z); Returns: a 11 x 3 matrix #m with the values.

	number of variables. If the period is omitted, the global time period is used. Returns: matrix	
rows(*x*)	Returns the number of rows of *x*. Returns: val	`%v = rows(#m);`
sumc(x)	Sum over cols Returns: matrix	`#m2 = sumc(#m1);`
sumr(x)	Sum over rows Returns: matrix	`#m2 = sumr(#m1);`
t(x)	Returns the transpose of a matrix. Returns: matrix	`#m2 = t(#m1);`
trace(x)	Returns the trace of a matrix. Returns: val	`%v = trace(#m);`

unpack(m)
unpack(<t1 t2>, m)
	The column matrix m (with only one column) is unpacked into a timeseries spanning the period t1-t2. If the period is omitted, the local/global time period is used.
	The unpack() function is not strictly necessary: you may alternatively assign a nx1 matrix directly to a series (see example).
	Returns: series
		`//This picks out the second column of #m (and all the rows).`
		`y = #m[.., 2].unpack(<2020 2030>);`
		`y <2020 2030> = #m[.., 2].unpack();  //same`
		`y <2020 2030> = #m[.., 2]; //also works`

zeros(n, k)	Returns a n x k matrix filled with 0's. Zeroes() can be used as alias. Returns: matrix	`#m = zeros(5, 10);`

Modelling

Function	*Description*	*Examples*

name		

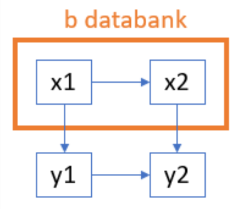
modelRawEqs()	Returns a list of strings corresponding to the equation names used by the loaded raw GAMS model. Note: the so-called GAMS scalar model is not used in this function.
	
	Returns: list of strings. [New in Gekko 3.1.16]
		`model <gms> mymodel.zip;`
		`sim <res>; //residuals`
		`#eqs = modelRawEqs();`
		`for string %s = #eqs;`
		`  p <n nomax> {%s};`
		`end;`
		`//prt {#eqs} may print too wide`

modelRawVars()	Returns a list of strings corresponding to the variable names used by the loaded raw GAMS model. Note: the so-called GAMS scalar model is not used in this function.
	
	Returns: list of strings. [New in Gekko 3.1.16]
		`#vars = modelRawVars();`
		`for string %s = #vars;`
		`  p <n nomax> {%s};`
		`end;`
		`//prt {#vars} may print too wide`

Data trace handling:

Function name	*Description*	*Examples*

traceBank(b, type)
traceBank(x, b, type)
	After a databank containing traces is read/opened, the traceBank() function can find data connections between variables. Arguments are x for the Gekko (array-)timeseries, b is a string containing a databank name, and type is either `'precedents'`, `'precedents1'` , `'dependents'` or `'dependents1'`. The x argument may be a string like `'x[a]!q'` or a series object like `x[a]!q`. You	`open<edit>b; clear b;`
		`x1 = 1;`
		`x2 = x1 + 2;`
		`close b;`
		`open b;`
		`y1 = 1 + b:x1;`
		`y2 = y1 + 2 + b:x2;`
		`prt tracebank('y2', 'b', 'precedents');`
		`prt tracebank('y2', 'b', 'precedents1');`
		`prt tracebank('x1', 'b', 'dependents');`
		`prt tracebank('x1', 'b', 'dependents1');`
		
		`//Output:`
		`//'x1!a', 'x2!a'`
		`//'x2!a'`
		`//'y1!a', 'y2!a'`

	may call the function without the x argument: in that case, all series names are considered.
	
	Note: for `'dependents'` type, it is recommended that x is a string name rather than a series object (because the series object may not even exist).
	
	The options `'precedents1'` and `'dependents1'` limits the effects to *direct* effects only. For instance, in the example, `y2` directly depends upon `x1` from the `b` databank, but not directly upon `x2`.
	
	Note: you may remove frequencies from the list elements with for instance `tracebank('y2', 'b', 'precedents').removefreq()`.
	
	[New in Gekko 3.1.22]
		`//'y1!a'`
		
		
		`//Relationships b`
		`//timeseries. The`
		`represents the b`
		`//The series x1 and x2 reside in the b databank,`
		`//where x2 depends upon x1. The series y1 depends`
		`//upon b:x1, and the series y2 depends upon y1`
		`//and b:x2. The first tracebank() call shows that`
		`//y2 depends upon both b:x1 and b:x2, the former`
		`//of these indirectly via x2. The third`
		`//tracebank() shows that b:x1 affects both y1 and`
		`//y2, the latter of these indirectly via both y1`
		`//and b:x2.`

traceDelete2()
traceDelete2(x)
	Deletes all trace references in the first-position databank (or use x for an optional databank name). After this, a traceStats2() will show 0 traces.
	
	[New in Gekko 3.1.16]
		`tracedelete2();`
		`tracedelete2('Work');`

traceStats2()
traceStats2(x)
	For now, only a simple overview of the number of traces can be shown (x is an optional databank name). Depth = 0 means traces that are directly assigned to existing timeseries, whereas depth = *n* > 0 indicate traces that are not assigned to	`tracestats2();`
		`tracestats2('Work');`

	existing timeseries, but are instead assigned to other traces with depth = *n*-1. Traces with depth > 0 can be thought of as remnants of timeseries that do not (or do no longer) exist in the particular databank. [New in Gekko 3.1.16]	

Miscellaneous functions:

Function name	*Description*	*Examples*

compareFolders(f1, f2)
compareFolders(f1, f2, filter)
compareFolders(f1, f2, filter, options)
	The function compares two folders f1 and f2 (including subfolders), a bit like COMPARE for files.
	With default settings, two files with the same name, in the same subfolder, with same size, and with same date (within 2 seconds) are considered equal. If dates differ, the files are compared by contents: byte for byte for binary files, and line for line for text files (differing whitespace at the start or end of a single text line will not count as a difference). The function outputs three files: comparefolders.txt with file info, comparefolders1.zip with differing common text files for easy comparison, and comparefolders2.zip with Gekko code to update folder1 or folder2.
	You may use a wildcard filter like for instance `'*.gcm, !*_bak.gcm'`,
		`comparefolders('c:\update\newdata', 'c:\update\olddata');`
		`comparefolders('c:\update\newdata', 'c:\update\olddata', '*.gcm, !*_bak.gcm');`
		`comparefolders('c:\update\newdata', 'c:\update\olddata', '', 'text strict');`

	which means all `.gcm` files except those ending with `_bak` (use prefix `!` for patterns to omit). The last argument can be `'text'` if you only want to compare text files (Gekko 'tastes' the files to see if they are binary or text), and/or `'strict'` to always compare files by content. Note that in comparefolders1.zip, folder delimiters are represented as `--`, and extension dot is represented as `,`. Sort the files by name when opening the .zip. You may drag and drop a folder from a file manager like Total Commander into the Gekko input window, to paste the folder path as a string. Total Commander is also convenient for comparing the files in comparefolders1.zip. [New in Gekko 3.1.16].	

existFile(x)	Returns value 1 if the file exists, 0 otherwise. If the file is stated without path, Gekko will also look in loaded libraries for it. See also readFile() and writeFile().
	[New in Gekko 3.1.9]
		`if(existfile('c:\data\scenario5.gbk'));`
		`  //do something;`
		`end;`

flush()	Removes all cache files (can also be done from the Gekko menu File --> Delete cache files...). This forces Gekko to re-read datafiles and models rather than using cached versions. At start up,	`flush();`

	Gekko performs flushing automatically every 14 days, or when the cache files take up more than 50 GB of file space. The function is intended for occasional manual use, and Gekko will refuse using it from a Gekko program. Using it too often will just slow down reading datafiles and models. [New in Gekko 3.1.14]	
gamsscalar(x)	This function can be called with `gamsscalar('pack')`, which creates a zip file with the GAMS scalar model, suitable for a MODEL statement. The function uses a settings file `gamsscalar.json`. The scalar model can be used for DECOMP and SIM<res>. More info on this soon.	`gamsscalar('pack');`

isNull(x)	Returns value 1 if x is a null variable, 0 otherwise. See also null().
	[New in Gekko 3.1.13]
		`#m = ((1, 2, 3), (4, null(), 6));`
		`if(isNull(#m[2, 2]));`
		`  //do something if cell [2, 2] is null`
		`end;`

isUtf8File(x)	Check if the text file x (string path) seems to be in UTF-8 format or not (if not, the format may be ANSI). If the file is stated without path, Gekko will also look in loaded libraries for it. See also existFile(), readFile() and writeFile().
	Returns: value 0 or 1.
	[New in Gekko 3.1.16]
		`%b = isUtf8File('rawdata.txt');`
		
		`//The checking is done by 'tasting' the file.`
		`//See also 'option system read encoding = ...' and`
		`//'option system write encoding = ...'`

lhsRhs(file)
lhsRhs(file, version)
	Experimental function that analyzes a Gekko `gcm` file syntactically and returns lists of timeseries names that appear on left-hand or right-hand sides of assignments ("series statements"). Input is filename (string) and version (value) -- value must be 2 (Gekko 2) or 3 (Gekko 3), default is 3. [New in Gekko 3.1.19]	`lhsRhs('data.gcm');`
		`lhsRhs('data.gcm', 2);`

map()	Returns an empty map.	`#m = map();`

null()	Returns a null variable. At the moment, null variables are mostly used to indicate empty "cells" in lists. You cannot perform calculations on null variables, but you can use the `type()` function to see the type of a given variable/cell (see example). See also isNull(). [New in 3.0.6]. Returns: null variable	`#m = ((1, 2, 3), (4, null(), 6));`
		`prt #m;`
		`prt #m[2, 2].type(); //'null'`

readFile(x)	Reads the file x (string) into a string. If the file is stated without path, Gekko will also look in loaded libraries for it. See also existFile(), writeFile() and readScreen(). Returns: string	`%s = readFile('rawdata.txt');`
		
		`//See also 'option system encoding ...'`

readScreen()	Reads the Gekko output screen (the upper part) into a string. See also readFile(). You may use the split() and nl() functions to split the string into lines. Returns: string	`#m = readscreen().split(nl(), 0);`
		`//This list contains the lines`
		`//shown on the screen. The 0 argument`
		`//makes sure empty lines are retained.`

tic()	Starts a timer, see `toc()`. Used to time Gekko programs.	`tic();`

toc()	Returns the number of seconds elapsed since a timer was started with `tic()`. Used to time Gekko programs.	`tic();`
		`PRT toc();`
		`//Prints seconds elapsed since tic().`

type(x)	Returns the type of a given variable `x`. The type is `'val'`, `'date'`, `'string'`, `'series'`, `'list'`, `'map'`, `'matrix'` or `'null'`. See examples. At the moment, null variables are mostly used to indicate empty "cells" in lists. [New in 3.0.6].
	Returns: string
		`#m = (1, 2021, 2021a, 'cat', null());`
		`p #m;`
		`p #m[1].type(); //'val'`
		`p #m[2].type(); //'val'`
		`p #m[3].type(); //'date'`
		`p #m[4].type(); //'string'`
		`p #m[5].type(); //'null'`

writeFile(x, s)	Writes the string s to the file x (string). A newline can be indicated with `'\n'`. See also existFile() and readFile().	`writeFile('rawdata.txt', '170 121 387');`
		
		`//See also 'option system encoding ...'`

yesno(x)	Converts value 1 or 0 into string `'yes'` or `'no'`. Quite a lot of options accept `yes`/`no`, but not 1/0. [New in Gekko 3.1.9]	`tell existfile('c:\data\scenario5.gbk').yesno();`

Part V

5 Gekcel (Excel add-in)

Gekcel is a Gekko add-in for Microsoft Excel. Gekko proper already has Excel interfaces (the SHEET, IMPORT and EXPORT statements), but these are called from within Gekko, using the Gekko graphical user interface (GUI). Instead of using these Excel interfaces from within Gekko, the Gekcel add-in does the inverse: that is, the add-in makes it possible to use Gekko from within Excel. Gekcel allows all Gekko components to be used from within Excel, including read/write of Gekko databanks (.gbk) etc.

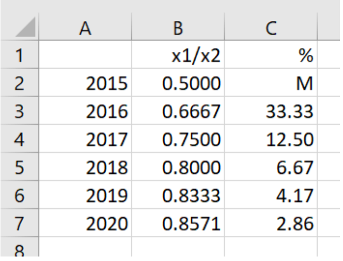
Note: The Gekcel interface is quite basic and simple in its current version. See the section on "Further development" regarding how the interface could be augmented and refined in the longer run. To use Gekcel, you need Microsoft Excel installed on your pc (Gekcel will not work for LibreOffice, Google Sheets or similar). About installation, see the guided tour.

When using the Gekcel add-in, only Excel is opened up, and the Gekko GUI is not shown at all. In Excel, Gekko results are shown in the current worksheet tab (as Excel cells), and/or in the so-called "Immediate Window" (more on this later on). For instance, when using the Gekcel add-in, the user can issue Gekko statements from within Excel, like for instance statements that read a Gekko databank or print timeseries.

The following Excel VBA (Visual Basic) script illustrates how to read a databank and print out the ratio of two timeseries `x1` and `x2`:

```
Public Sub Demo()
  Gekko "read demo.gbk;"
  Gekko "prt <2015 2020> x1/x2;"
  Gekko_Get
End Sub
```

This syntax etc. will be explained in more details in the following sections, so it is just to illustrate that interacting with Gekko via VBA scripts is quite easy when using Gekcel. As an alternative to using VBA, the so-called Excel "Immediate Window" can also be used to the same effect. The result of the above VBA code will look like the following in Excel:

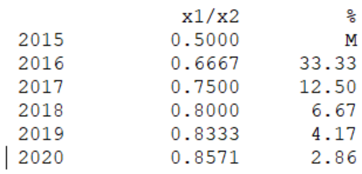


	A	B	C
1		x1/x2	%
2	2015	0.5000	M
3	2016	0.6667	33.33
4	2017	0.7500	12.50
5	2018	0.8000	6.67
6	2019	0.8333	4.17
7	2020	0.8571	2.86
8			

If, instead, the user starts up Gekko proper and the corresponding statements are typed into the Gekko GUI input window:

```
read demo.gbk;
prt <2015 2020> x1/x2;
```

the following would be shown in the Gekko output window:



```
             x1/x2          %
2015        0.5000          M
2016        0.6667      33.33
2017        0.7500      12.50
2018        0.8000       6.67
2019        0.8333       4.17
2020        0.8571       2.86
```

So in that sense, the Gekcel add-in for Excel is capable of showing the same kinds of results that the normal Gekko program does. The familiarity of using an Excel spreadsheet is one of the reasons that the Gekcel add-in was developed. Calling an add-in from within Excel may seem a less daunting task than opening up the full Gekko program and issuing statements from there.

The Gekko add-in is not limited in any way, so anything Gekko can do can also be done via the add-in. The add-in is also complete in the sense that nothing else than the add-in needs to be installed in order to use Gekcel.

Read more about Gekcel in the sub-sections.

5.1 Gekcel basics

The next section (Gekcel guided tour) provides a step by step guided tour of Gekcel and how to use it, whereas the current section tries to explain some of the basic ideas behind the add-in. (The impatient may skip the current section and jump directly to the guided tour).

Gekcel comes in form of a macro-enabled `Gekcel.xlsm` file, together with `Gekcel.xll` , `Gekcel.dll`, and some other .exe and .dll files. The `Gekcel.xlsm` file is primary in the sense that it contains Excel VBA functions to interact with Gekko.

These interfacing Excel functions are written in VBA (Visual Basic), the programming language of Excel. This does not imply that you are expected to code complicated stuff in VBA, since basic use of Gekcel only entails the use of the simplest kinds of VBA calls like `Gekko "read demo.gbk;"` and similar. So don't worry: Gekcel is meant to make it more simple for Excel users to use Gekko, not to force anyone to learn how to code VBA.

As mentioned, the next section (guided tour) takes you through an example step by step, and after that guided tour you will probably realize that the VBA part of Gekcel mostly works like a kind of Gekko program file (.gcm), storing the Gekko statements in sequences. And if you really dislike VBA, you may alternatively use the so-called

Excel Immediate Window, completely avoiding the use of the Excel VBA editor (there is a section on that, too).

To return to the functionality of Gekcel, in its essence, the Gekcel add-in is capable of starting up a silent Gekko session in the background (without opening up the Gekko graphical user interface), and issue Gekko commands from within Excel (typically via VBA). In addition, data can be transferred from Gekko to Excel worksheet cells and vice versa. There are the following fundamental Gekcel VBA functions available:

- **Gekko()**. Runs a Gekko statement from within Excel.
- **Gekko_Get()**. Transfers data from Gekko into an Excel worksheet tab.
- **Gekko_Put()**. Transfers data from an Excel worksheet tab to Gekko.

The `Gekko_Get()` and `Gekko_Put()` subroutines are explained in more details here. In `Gekcel.xlsm`, the three VBA subroutines can be seen in the VBA Editor in Excel. More on how to open and use the Excel VBA Editor in the next sections, but the functionality of the three subroutines is as follows:

- The `Gekko()` function is straightforward. For instance, you may use `Gekko("read demo.gbk;")` in VBA to issue the Gekko statement `read demo.gbk;`, opening up that particular Gekko databank.
- The `Gekko_Get()` function uses the following logic. When a Gekko statement is called from within Excel/Gekcel, instead of producing an .xlsx file or printing out results in the Gekko main window, some of the Gekko statements instead produce an internal two-dimensional table containing the results. This table can subsequently be loaded into the active Excel worksheet by means of the VBA `Gekko_Get()` function. At the moment, two Gekko statements can be used in this way: SHEET and PRT. So the idea is to call for instance SHEET with `Gekko("sheet x1, x2;")`, which produces the before-mentioned two-dimensional internal table. After this, the internal table can be transferred to the active worksheet in Excel with `Gekko_Get()`. More details here.
- The `Gekko_Put()` function does the reverse, that is, it takes all cells from the currently active worksheet tab and transfers them to the internal two-dimensional table. After this, the Gekko IMPORT statement can be used to transfer data from the internal table to Gekko (to its first-position databank). So the procedure is to first call the `Gekko_Put()` function, and then use `Gekko("import <xlsx> gekcel;")` to transfer the data to Gekko. The use of `gekcel` as filename is arbitrary just to indicate where the data comes from: when Gekcel is used, all `import<xlsx>` statements will take data from the internal table, instead of looking for an xlsx file. (It is planned to make it possible to use just `Gekko("import <xlsx>;")` in Gekcel, skipping the superfluous filename). More details here.

So to reiterate: when "remote controlling" Gekko from Excel/Gekcel, three of the Gekko statements are rewired, namely SHEET, PRT and IMPORT<xlsx>. The first two statements produce an internal two-dimensional table that can be fetched into Excel via the `Gekko_Get()` VBA function. To do the reverse, `Gekko_Put()` transfers the current worksheet cells to the same kind of internal table, which IMPORT<xlsx> can subsequently read into the Gekko first-position databank.

When getting data from Gekko, note that both SHEET and PRT have a lot of functionality besides "just" printing out variables. For instance, you may use mathematical expressions, and there are a lot of operators available, like `<p>` for percentage growth, `<m>` for databank comparisons, etc. Therefore, you can often avoid the effort of transforming the data via point and click (and dragging formulas) in Excel, and instead make Gekko do the heavy lifting. Besides mathematical expressions and operators, SHEET and PRT can also use wildcards and lists to search for or "compose" variable names.

It should be noted that when used from Excel/Gekcel, the Gekko SHEET and PRT statements produce the same contents, except for a difference regarding presentation:

- SHEET produces a data table with periods in each row, and variables in each column
- PRT produces a data table with variables in each row, and periods in each column. Besides, percentage growth is also shown per default.

There are a lot of options available to control this formatting if needed (cf. `option sheet ... ;` and `option print ... ;`).

5.2 Gekcel guided tour

In the previous section on Gekcel basics, the basic ideas behind the Gekcel add-in for Excel were explained.

In the current section we will go through a simple practical example step by step. In the example, we will create some timeseries in Gekko (remote-controlled from Excel/Gekcel), transfer these to Excel, change them in Excel, and transfer them back to Gekko.

Download and installation

First you need to "install" Gekcel. To download and install, do the following:

1. Download the `Gekcel.zip` file. If in doubt, take `Gekcel.zip` from the latest development version (here). If you are using Excel 32-bit: download Gekcel 32-bit, and if you are using Excel 64-bit: download Gekcel 64-bit. How to determine bitness of your Excel version?: +toggle.

On a 64-bit Windows pc, Microsoft Excel can be installed as either a 32-bit or 64-bit version. Until 2020, 32-bit was the default when installing Excel (or Office), so there are a lot of 32-bit Excel versions installed out there, also on 64-bit Window systems. Gekcel needs the same "bitness" as Excel to run, and to check the bitness of you current Excel versions, perform the following (for Excel 2013, 2016 or 2019): Select the 'File' tab from the Excel ribbon, select 'Account' and then 'About Excel'.

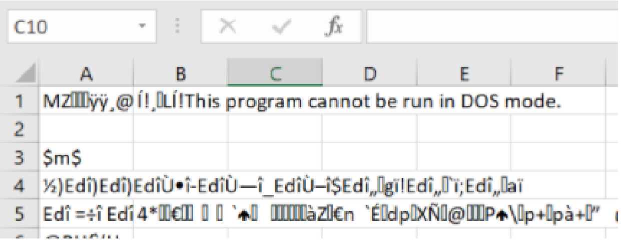
The version and bit-level of Excel will be displayed in the top line of the window (for Excel 2010, you should select 'Help' instead of 'Account'). Excel 2007 and prior is always 32-bit.

2. Create a new separate folder for the Gekcel "installation", and unpack the files inside `Gekko.zip` into this folder.
3. From the "installation" folder, open the `Gekcel.xlsm` file (double click it). You may get a security warning about macros being disabled (activate these macros).
4. If you already have an older version of Gekcel installed, you *may* have to remove this older version manually first (probably this step can be skipped in most cases). This can be done from the File tab, click Options. Under Manage choose COM Add-ins in the drop-down and click Go. From there, you can choose Remove. Do this for all components with "Gekcel" in the name. After this, it is perhaps best to go to File tab, click Options. Under Manage choose Excel Add-ins in the drop-down (this is default), and click Go. Here, you can deactivate Gekcel in a checkbox.
5. From the "installation" folder, drag (in File Explorer/Windows Explorer) the file `Gekcel.xll` into the recently opened `Gekcel.xlsm` Excel spreadsheet (you may get a security warning: accept to activate the add-in). Alternatively if this does not work (perhaps because of security settings) you can do this: from the recently opened `Gekcel.xlsm` spreadsheet, click the File tab, click Options, under Manage choose Excel Add-Ins, click Go, and in the newly opened window, you can now browse for and select the `Gekcel.xll` file (click ok after that). If during this step there is an Excel window asking something like "Copy 'Gekcel.xll' to the Addins folder for [your name]?", answer no to that question (Gekcel depends upon other files thatn Gekcel.xll, so answering yes will cause problems). When Gekcel.xll is active on a spreadsheet, there is a "Gekko" tab at the Ribbon.
6. Close `Gekcel.xlsm`.
7. Optional. If you prefer, you may now take a copy of `Gekcel.xlsm`, and transfer it to some "working" folder of your choice. For new users, it is perhaps best to skip this step, to avoid potential folder issues. If you do make a copy, the copied `Gekcel.xlsm` should still remember the location of the other Gekko files (in the "installation" folder).

Note 1: Gekcel is a self-contained product bundle, containing all necessary Gekko files inside (including `gekko.exe` etc.). Therefore, Gekcel runs completely independently of any normal Gekko installation (and vice versa), and the version number and bitness of Gekcel vs. Gekko installations may differ.

Note 2: When Gekko is called via Gekcel, the default working folder is always the same folder as the Excel worksheet resides in (in this example, `Gekcel.xlsm`). If this folder is read-only or if the folder cannot be identified, Gekko will use the desktop as working folder. Alternatively, you can set the working folder manually with the VBA statement `Gekko "option folder working = g:\data\testing;"` (for example).

Note 3: When trying to open/load Gekcel.xll, you may encounter a warning about the file format and the file being potentially corrupted. If opening anyhow, you may see something like the following:



```
C10
   A  B  C  D  E  F
1  MZ￿￿ÿÿ¸@Í!¸￿LÍ!This program cannot be run in DOS mode.
2
3  $m$
4  ½)Edî)Edî)EdîÙ•î-EdîÙ—î_EdîÙ–î$Edî„￿gï!Edî„￿`ï;Edî„￿aï
5  Edî =÷î Edí 4*￿￿€￿￿ ￿ ￿ `￿ ￿￿￿￿àZ￿€n `É￿dp￿XÑ￿@￿￿￿P￿\￿p+￿pà+￿”
```

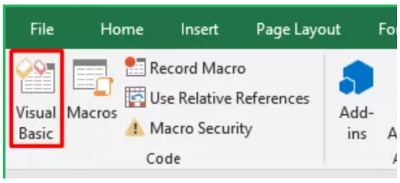
This is often an indication that the bitness is wrong: that you are either trying to use 32-bit Gekcel on 64-bit Excel, or vice versa. In that case, Excel will not provide any helpful explanation, but will just fail like shown above.

Running the demo (VBA subroutine)

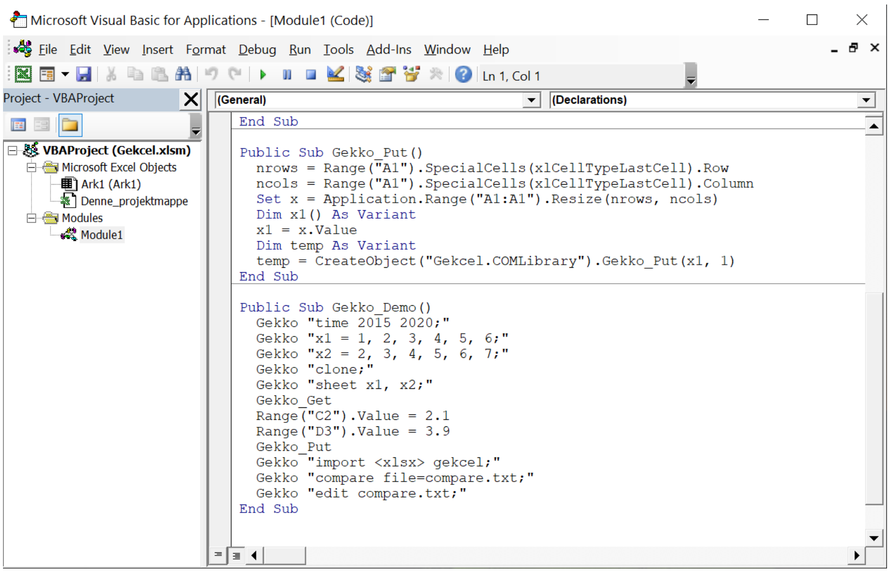
Open up the same `Gekcel.xlsm` again. It should hopefully remember Gekcel, and show a "Gekko" tab at the Ribbon. If not, you may have to drag `Gekcel.xll` onto `Gekcel.xlsm` again.

First, we will try to run the in-built `Gekko_Demo()` VBA subroutine inside `Gekcel.xlsm`. (A VBA subroutine is just a VBA function that does not return anything). To do this, you will first have to open up the Excel VBA editor.

The VBA editor is found in the Developer tab, click the Visual Basic button. (If you do not see the Developer tab, go to File --> Options --> Customize Ribbon and make sure that "Developer" is checked in the right pane. Alternatively, you may use the Macros section of the View tab, but enabling the Developer tab is better.).



This should open up the VBA editor:



```
End Sub

Public Sub Gekko_Put()
  nrows = Range("A1").SpecialCells(xlCellTypeLastCell).Row
  ncols = Range("A1").SpecialCells(xlCellTypeLastCell).Column
  Set x = Application.Range("A1:A1").Resize(nrows, ncols)
  Dim x1() As Variant
  x1 = x.Value
  Dim temp As Variant
  temp = CreateObject("Gekcel.COMLibrary").Gekko_Put(x1, 1)
End Sub

Public Sub Gekko_Demo()
  Gekko "time 2015 2020;"
  Gekko "x1 = 1, 2, 3, 4, 5, 6;"
  Gekko "x2 = 2, 3, 4, 5, 6, 7;"
  Gekko "clone;"
  Gekko "sheet x1, x2;"
  Gekko_Get
  Range("C2").Value = 2.1
  Range("D3").Value = 3.9
  Gekko_Put
  Gekko "import <xlsx> gekcel;"
  Gekko "compare file=compare.txt;"
  Gekko "edit compare.txt;"
End Sub
```

Near the bottom of the VBA editor, you should find the Gekko_Demo() subroutine:

```
Public Sub Gekko_Demo()
  Gekko "tell gekkoinfo('short4');"          'Print version and
working folder
  Gekko "time 2015 2020;"                    'Set period 2015-20
  Gekko "x1 = 1, 2, 3, 4, 5, 6;"             'Set series x1
  Gekko "x2 = 2, 3, 4, 5, 6, 7;"             'Set series x2
  Gekko "clone;"                             'Copy Work databank to
Ref databank
  Gekko "sheet x1, x2;"                      'Send Work data to
internal table
  Gekko_Get                                  'Get internal table into
active Excel worksheet
  Range("C2").Value = 2.1                    'Change something in
Excel (x1 in 2016)
  Range("D3").Value = 3.9                    'Change something in
Excel (x2 in 2017)
  Gekko_Put                                  'Put active Excel
worksheet into internal table
  Gekko "import <xlsx> gekcel;"              'Import data from
internal table
  Gekko "compare file=compare.txt;"          'Compare data in Work
and Ref databanks
  Gekko "edit compare.txt;"                  'Show the differences
(compare.txt)
End Sub
```

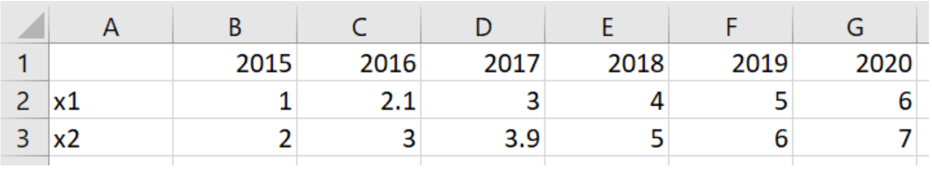
The demo creates two annual timeseries `x1` and `x2` in Gekko, transfers them to the current Excel worksheet tab, changes two cells in that worksheet (corresponding to `x1[2016]` and `x2[2017]`), transfers the two timeseries back to Gekko, and finally compares the two timeseries to their original versions (in Gekko).

To run `Gekko_Demo()`, place the cursor somewhere inside the subroutine and press F5 or hit the "Run sub" green button. (This may fail with a "The macros in this project are disabled". To enable them, click the File tab, Options, Trust Center, Trust Center Settings, Macro Settings, activate them and click ok).

This will run the statements, and in the end, the file `compare.txt` is shown in Notepad (the file identifies the changes corresponding to the two altered cells `C2` and `D3`). The `compare.txt` file looks something like this:

```
Out of the 2 common series, there are differences regarding 2 of them:

x1              WORK        REFERENCE             ABS DIFF       % DIFF
-------------------------------------------------------------------
2015          1.0000          1.0000               0.0000          0.00
2016          2.1000          2.0000               0.1000          5.00
2017          3.0000          3.0000               0.0000          0.00
2018          4.0000          4.0000               0.0000          0.00
2019          5.0000          5.0000               0.0000          0.00
2020          6.0000          6.0000               0.0000          0.00

x2              WORK        REFERENCE             ABS DIFF       % DIFF
-------------------------------------------------------------------
2015          2.0000          2.0000               0.0000          0.00
2016          3.0000          3.0000               0.0000          0.00
2017          3.9000          4.0000              -0.1000         -2.50
2018          5.0000          5.0000               0.0000          0.00
2019          6.0000          6.0000               0.0000          0.00
2020          7.0000          7.0000               0.0000          0.00
```

In the Excel worksheet, you will see the following:



	A	B	C	D	E	F	G
1		2015	2016	2017	2018	2019	2020
2	x1	1	2.1	3	4	5	6
3	x2	2	3	3.9	5	6	7

These are the values of `x1` and `x2` (with the two changes in cells `C2` and `D3` seen).

So, in VBA, you can just use the prefix `Gekko ` followed by the Gekko statement inside double quotes (`"`), because you may omit the more strict function call parentheses (`Gekko("time 2015 2020;")`). In the same manner, you may use `Gekko_Put` instead of `Gekko_Put()`, and `Gekko_Get` instead of `Gekko_Get()`. Remember that Gekko statements must end with semicolon `;`, otherwise you will get a syntax error!

Consider using OPEN/CLOSE

The VBA Gekcel interface is quite simple to use, as the `Gekko_Demo()` example above illustrates. Beware however that if the databank used is shared between users on a common network drive, it may be better to use OPEN<edit> and CLOSE, rather than IMPORT/EXPORT or READ/WRITE. For instance:

```
Public Sub Gekko_Update()
  Gekko "open <edit> data;"                 'Load data.gbk as the
first-position databank
  Gekko "sheet x1, x2;"                     'Send first-position
data to internal table
  Gekko_Get                                 'Get internal table into
active Excel worksheet
  Gekko "pause 'Click ok when finished';"   'Pause so that the user
has time to update values
  // ... change Excel data values ...
  Gekko_Put                                 'Put active Excel
worksheet into internal table
  Gekko "import <xlsx> gekcel;"             'Import data from
internal table into the first-position databank
  Gekko "close data;"                       'Close data.gbk (write
any changes)
End Sub
```

Using the statement pair `open<edit>data;` ... `close data;` is better than using for instance using `read data;` ... `write data;` because when `open<edit>` is used, Gekko will check the status of `data.gbk` when the databank is closed. If its last written date has changed since it was opened, Gekko will refuse to write the data back to the databank. This guards agains synchronization issues, for instance if another user has changed `data.gbk` after the `open<edit>` statement.

The subroutine could alternatively be split in two parts like the following:

```
Public Sub Gekko_Update1()
  Gekko "open <edit> data;"                 'Load data.gbk as the
first-position databank
  Gekko "sheet x1, x2;"                     'Send first-position
data to internal table
  Gekko_Get                                 'Get internal table into
active Excel worksheet
End Sub
```

```
Public Sub Gekko_Update2()
  Gekko_Put                                 'Put active Excel
worksheet into internal table
  Gekko "import <xlsx> gekcel;"             'Import data from
internal table into the first-position databank
  Gekko "close data;"                       'Close data.gbk (write
any changes)
End Sub
```

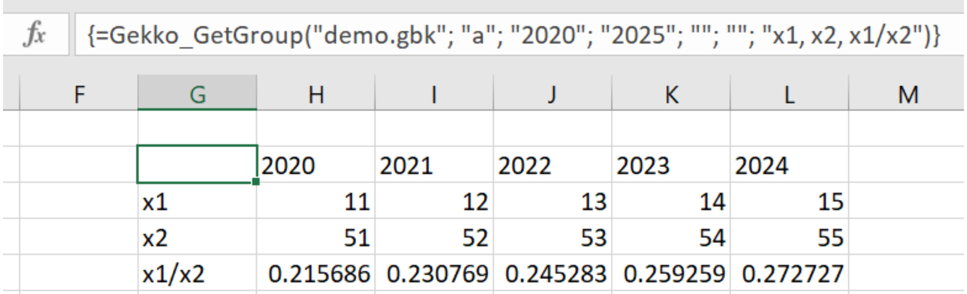
Then `Gekko_Update1()` would load the data (here: series `x1` and `x2`) from `data.gbk` into the spreadsheet, and when finished `Gekko_Update2()` would transfer the data back to `data.gbk` (while checking that `data.gbk` has not been tampered with).

Getting data with an array function

The `Gekko_Get()` function used above obtains data fetched in a previous SHEET statement (`Gekko "sheet ...;"`). You may instead fetch the data in one go, using the more tailor-made `Gekko_GetGroup()` function.

The function is an array function that outputs an array of cells, and you must therefore use Ctrl+Shift+Enter to enter it (this is also called a "CSE formula"). To use it, first mark a two-dimensional range of cells where you want to show the data (for instance 4 rows and 6 columns somewhere in your sheet). Then try to enter the following formula, finishing with Ctrl+Shift+Enter. (It is assumed that the `demo.gbk` databank from the `Gekcel.zip` file is present in the same folder as the spreadsheet).

```
=Gekko_GetGroup("demo.gbk"; "a"; "2020"; "2025"; ""; ""; "x1, x2,
x1/x2")
```



{=Gekko_GetGroup("demo.gbk"; "a"; "2020"; "2025"; ""; ""; "x1, x2, x1/x2")}

F	G	H	I	J	K	L	M
		2020	2021	2022	2023	2024	
	x1	11	12	13	14	15	
	x2	51	52	53	54	55	
	x1/x2	0.215686	0.230769	0.245283	0.259259	0.272727	

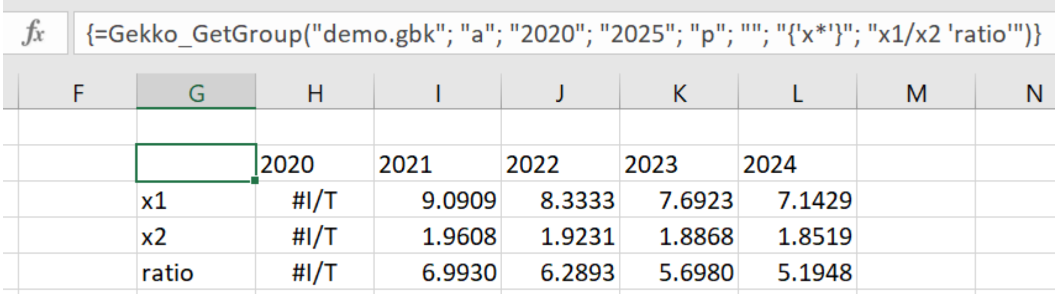
The above results should be shown. Excel automatically inserts the `{}`-curlies after the Ctrl+Shift+Enter to indicate that this is an array function. Note that the year 2025 is not shown above, because only 6 columns were selected in the two-dimensional range. When using the `Gekko_GetGroup()` array function, if the selected area of Excel cells is too small to show the full results, the results are truncated. And if the area is too large, missing values (here: `#I/T`, in English Excel versions: `#N/A`) will fill out the area.

In the function call, the first argument is the Gekko databank, followed by the frequency of the data, and start and end periods. The fifth argument is an optional operator (try for instance `"p"` for percentage growth), whereas the sixth argument is for other options (not used at the moment). The last argument is the timeseries that are to be shown (separated by comma). Instead of one string at the end, you may alternatively use three strings separated by `;` as shown here:

```
=Gekko_GetGroup("demo.gbk"; "a"; "2020"; "2025"; ""; ""; "x1";
"x2"; "x1/x2")
```

You may use any legal Gekko-expressions, and wildcards should be stated like for instance `{'a*b?c'}` for a `a*b?c` wildcard. Labels can be set with single quotes. For instance:

```
=Gekko_GetGroup("demo.gbk"; "a"; "2020"; "2025"; "p"; ""; "{'x*'}";
 "x1/x2 'ratio'")
```



fx {=Gekko_GetGroup("demo.gbk"; "a"; "2020"; "2025"; "p"; ""; "{'x*'}"; "x1/x2 'ratio'")}

F	G	H	I	J	K	L	M	N
		2020	2021	2022	2023	2024		
	x1	#I/T	9.0909	8.3333	7.6923	7.1429		
	x2	#I/T	1.9608	1.9231	1.8868	1.8519		
	ratio	#I/T	6.9930	6.2893	5.6980	5.1948		

Here, `x1` and `x2` match the wildcard `x*`, and `x1/x2` is shown with the label `'ratio'`. The values are shown as percentage growth (because of operator `"p"`). If you use for instance `"n p"` instead of just `"p"`, you will get both levels and percentage growth rates shown.

In essence, `=Gekko_GetGroup("demo.gbk"; "a"; "2020"; "2025"; "n p"; ""; "{'x*'}"; "x1/x2 'ratio'")` is translated into the corresponding Gekko statements: `read demo.gbk; option freq a; time 2020 2025; sheet <n p> {'x*'}, x1/x2 'ratio';`. You can see how this "translation" is done in the VBA function `Gekko_GetGropup()` inside the spreadsheet.

Use of immediate Window

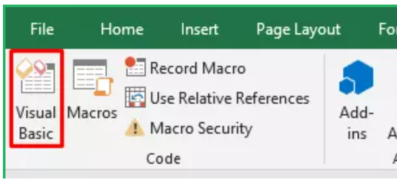
When used as above, Gekko will run silently, without showing any text messages while running.

To see Gekko text messages (including error messages) and issue Gekko statements in a more interactive fashion without VBA, you may alternatively activate and use the so-called "Immediate Window" in Excel. Read more about this in the next section.

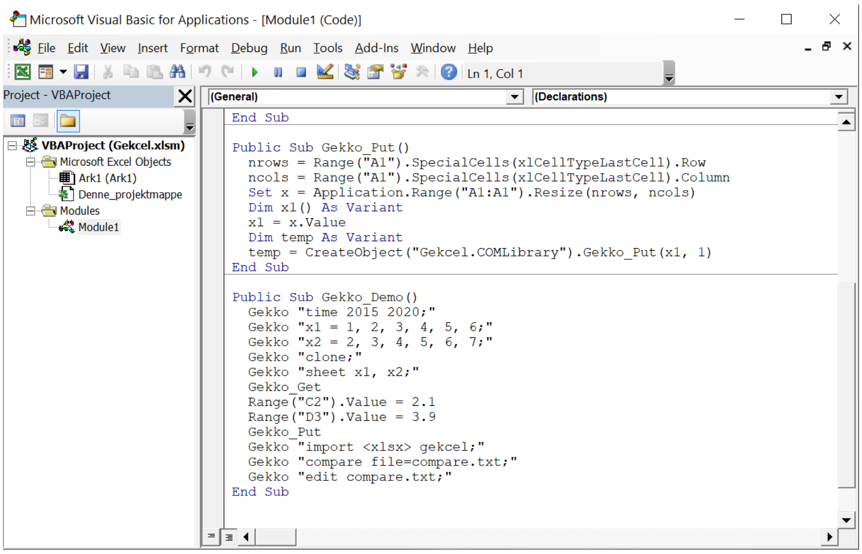
5.3 Immediate Window

When using the Gekcel add-in for Excel, there is a more direct way of issuing VBA statements, rather than defining and running a VBA subroutine containing the statement(s). This direct way is by means of the so-called "Immediate Window" in Excel, and this window can also show Gekko text messages or other text output (like Gekko errors).

To open the Immediate window from the `Gekcel.xlsm` file, do the following. From the Developer tab, click the Visual Basic button to open up the Excel VBA editor. (If you don't see the Developer tab, go to File --> Options --> Customize Ribbon and make sure that "Developer" is checked in the right pane).

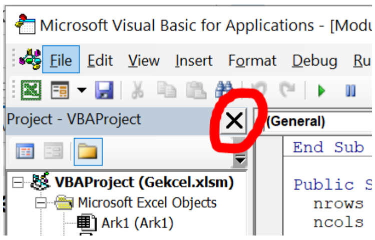


This should show a window similar to the the following. In this window, the left part shows the "Project" where the VBA code is located, here under Modules, Module1. The right part of the windows shows the VBA code itself.



```
End Sub

Public Sub Gekko_Put()
  nrows = Range("A1").SpecialCells(xlCellTypeLastCell).Row
  ncols = Range("A1").SpecialCells(xlCellTypeLastCell).Column
  Set x = Application.Range("A1:A1").Resize(nrows, ncols)
  Dim x1() As Variant
  x1 = x.Value
  Dim temp As Variant
  temp = CreateObject("Gekcel.COMLibrary").Gekko_Put(x1, 1)
End Sub

Public Sub Gekko_Demo()
  Gekko "time 2015 2020;"
  Gekko "x1 = 1, 2, 3, 4, 5, 6;"
  Gekko "x2 = 2, 3, 4, 5, 6, 7;"
  Gekko "clone;"
  Gekko "sheet x1, x2;"
  Gekko_Get
  Range("C2").Value = 2.1
  Range("D3").Value = 3.9
  Gekko_Put
  Gekko "import <xlsx> gekcel;"
  Gekko "compare file=compare.txt;"
  Gekko "edit compare.txt;"
End Sub
```

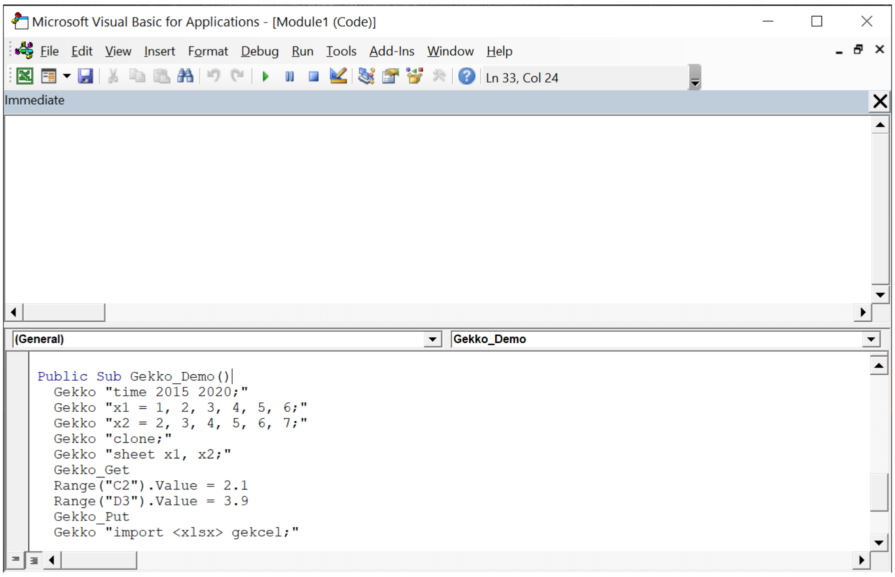
To make this look a little bit more like the Gekko user interface, we may remove the unnecessary left part of the window and add the so-called Immediate Window. Remove the left part of the window ("Project"), by simply closing it with the close button:



You may also have to remove a similar "Properties" window on the left part of the VBA editor: this is done in the same way.

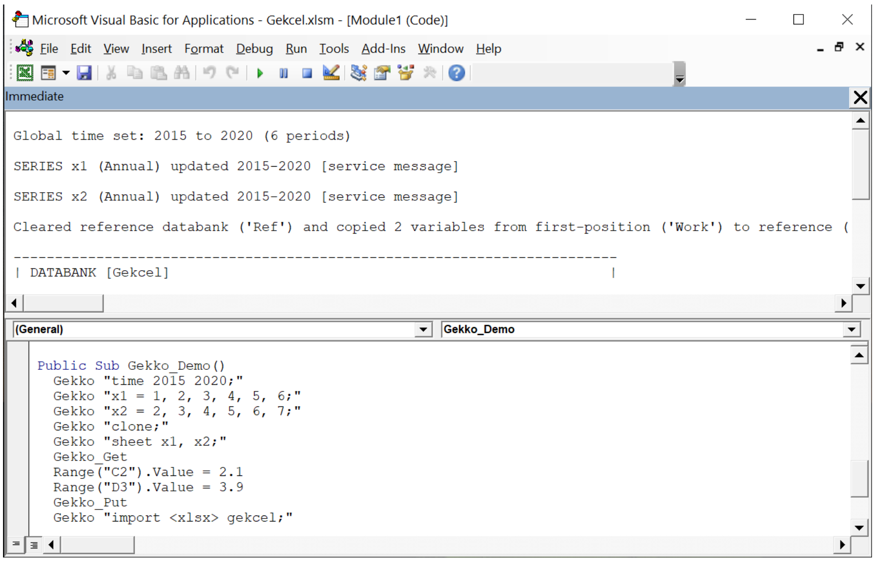
Next, you should open up the Immediate Window via the Excel menu: View --> Immediate Window (or use the short-cut Ctrl+G). The Immediate Window will probably show up in the lower part of the VBA editor. Try dragging it (drag the blue

"Immediate" band) to the top of the VBA editor, so that the Excel VBA editor ends up looking like this:


Microsoft Visual Basic for Applications - [Module1 (Code)]
File Edit View Insert Format Debug Run Tools Add-Ins Window Help
Ln 33, Col 24
Immediate
(General)
Gekko_Demo

```
Public Sub Gekko_Demo()
  Gekko "time 2015 2020;"
  Gekko "x1 = 1, 2, 3, 4, 5, 6;"
  Gekko "x2 = 2, 3, 4, 5, 6, 7;"
  Gekko "clone;"
  Gekko "sheet x1, x2;"
  Gekko_Get
  Range("C2").Value = 2.1
  Range("D3").Value = 3.9
  Gekko_Put
  Gekko "import <xlsx> gekcel;"
```

This now looks a bit more like the standard Gekko graphical user interface. Now, you can again run the subroutine `Gekko_Demo()`. Click somewhere inside the subroutine and click the green "Run Sub" button (or hit F5). This time, the Gekko output will be shown in the Immediate Window, like for instance:


Microsoft Visual Basic for Applications - Gekcel.xlsm - [Module1 (Code)]
File Edit View Insert Format Debug Run Tools Add-Ins Window Help
Immediate
(General)
Gekko_Demo

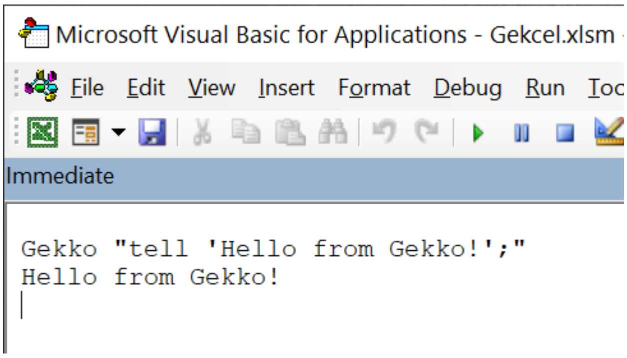
```
Global time set: 2015 to 2020 (6 periods)

SERIES x1 (Annual) updated 2015-2020 [service message]

SERIES x2 (Annual) updated 2015-2020 [service message]

Cleared reference databank ('Ref') and copied 2 variables from first-position ('Work') to reference (

------------------------------------------------------------------------
| DATABANK [Gekcel]                                                    |
```

```
Public Sub Gekko_Demo()
  Gekko "time 2015 2020;"
  Gekko "x1 = 1, 2, 3, 4, 5, 6;"
  Gekko "x2 = 2, 3, 4, 5, 6, 7;"
  Gekko "clone;"
  Gekko "sheet x1, x2;"
  Gekko_Get
  Range("C2").Value = 2.1
  Range("D3").Value = 3.9
  Gekko_Put
  Gekko "import <xlsx> gekcel;"
```

In this way, normal Gekko text messages and Gekko errors will be shown, making it easier to see what Gekko is actually doing when running a VBA subroutine containing Gekko statements.

The Immediate Window can also be used to input smaller snippets/lines of Gekko code. For instance, in the Immediate Window you may try to issue a Gekko TELL statement by writing the line `Gekko "tell 'Hello from Gekko!';"` in the

Immediate Window and hitting [Enter]. After doing this, the answer from Gekko will be shown just below like here:

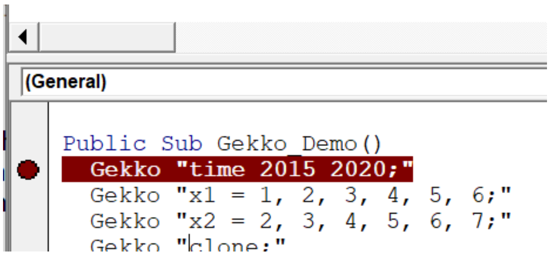

Microsoft Visual Basic for Applications - Gekcel.xlsm
File Edit View Insert Format Debug Run Too
Immediate

```
Gekko "tell 'Hello from Gekko!';"
Hello from Gekko!
```

Used like this, you may issue Gekko statements directly from the Immediate Window, without defining and running a VBA subroutine. When hitting [Enter] in the Immediate Window, Gekko will try to interpret and run the current line. If you need to insert a newline without executing the current line, you can use Ctrl+Enter (like in the Gekko GUI window proper).

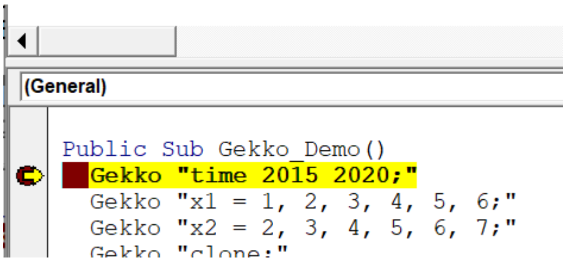
Unfortunately, it does not seem possible to operate with two such Immediate Windows, issuing statements in one of them and receiving Gekko output in the other. So when using the Immediate Window to issue Gekko statements, Gekko statements and Gekko output get mixed up in the Immediate Window. To avoid this mix up, you have to run the the Gekko statement(s) as an Excel subroutine (VBA).

When using VBA, you can insert a breakpoint to make Excel stop execution at a specific line (click on the grey bar to the left of the line):



```
Public Sub Gekko_Demo()
  Gekko "time 2015 2020;"
  Gekko "x1 = 1, 2, 3, 4, 5, 6;"
  Gekko "x2 = 2, 3, 4, 5, 6, 7;"
  Gekko "clone;"
```

After this, when you run the `Gekko_Demo()` subroutine (with the green "Run Sub" button or F5), Excel will stop and wait before the execution of the red-marked line.



```
Public Sub Gekko_Demo()
  Gekko "time 2015 2020;"
  Gekko "x1 = 1, 2, 3, 4, 5, 6;"
  Gekko "x2 = 2, 3, 4, 5, 6, 7;"
  Gekko "clone;"
```

After this, you may execute the lines one by one with Shift+F8. This way, you may run Gekko statements one by one in an orderly fashion, while inspecting the Gekko output. This provides an experience similar to running Gekko statements from the Gekko GUI window proper.

Loops etc.

One thing to note about this way of executing Gekko statements is that a statement like for instance `Gekko "time 2015 2020;"` is run as a whole, corresponding to writing `time 2015 2020;` in the normal Gekko GUI window and hitting [Enter].

If you for instance have a loop like `for val %i = 1 to 3; tell %i.string(); end;` (printing the numbers 1, 2, and 3), you cannot issue this statement in VBA in three parts, like for instance:

```
Public Sub Gekko_Loop()
  Gekko "for val %i = 1 to 3;"
  Gekko "tell %i.string();"
  Gekko "end;"
End Sub
```

This will fail with a Gekko parsing error (same error as issuing the statement `for val %i = 1 to 3;` in the normal Gekko input window). Instead, in Gekcel, you have to issue the loop as one statement, like for instance:

```
Public Sub Gekko_Loop()
  Gekko "for val %i = 1 to 3; tell %i.string(); end;"
End Sub
```

This will print the numbers 1, 2, and 3 in the Immediate Window (if this is activated/shown).

Therefore, when using Gekcel, program structures containing BLOCK, FOR, IF, FUNCTION or PROCEDURE (and corresponding END statements) must be issued as *one* VBA statement.

If you need to use such functionality, you can just put this code in an external Gekko `.gcm` file which you subsequently call with a RUN statement from Excel VBA.

5.4 Further development

At the moment, the Gekcel interface is rather basic, just enabling the Gekko statements SHEET and PRT to transfer data from Gekko to Excel, and the statement Gekko `IMPORT<xlsx>gekcel;` to transfer data from Excel to Gekko.

Some ideas and suggestions:

- There are some questions about how the Gekko working folder is set, when an Excel workbook with an Gekcel add-in starts. If this poses problems, the user can always set the Gekko working folder manually with the Gekko statement `option folder working = ... ;`, but the issue should be investigated.
- It is not at all clear how `Gekcel.xlsm` "knows" that it is coupled to a `Gekcel.xll` add-in. This information is not stored in `Gekcel.xlsm`, so somehow Windows keeps track of this.
- The Quarterly National Accounts of Statistics Denmark have written the VBA functions `Get_Gekko_Group()` and `Put_Gekko_Group()` where the user can state a list of timeseries names, and a date range. This way, a limited part of the current worksheet can be operated on, limiting the scope for errors (in essence, keeping parts of the worksheet read-only). At the moment, Gekko only implements the first of these two, implemented as `Gekko_GetGroup()` function. There is also the question of using CSE array-functions, because they are a bit cumbersome. With some tricks, it is possible to state a VBA function in one cell and have the function "spill" into multiple cells. This is normally not legal in Excel (until the new dynamic arrays are common in Excel). The Central Bank of Denmark are exploring such Excel tricks for Gekcel, using a Windows timer to do it.
- In the longer run, a Gekko Ribbon tab could be added to Excel, making it possible to use some of the functionality of Gekko by means of input dialog windows (point and click). For instance opening a particular bank, selection time period, selecting timeseries, and having these timeseries show up in an Excel workbook tab. And vice versa transfer data from such a workbook tab back to Gekko.
- As an alternative or supplement, Excel functions could be added, so that the user can type `=SomeGekkoFunction()` in a cell, and get results displayed without using the VBA editor window at all.
- The `Gekko_Get()` and `Gekko_Put()` calls could probably be omitted: these calls were mostly added to comply more tightly with a corresponding EViews interface for Excel. Perhaps a SHEET or PRT statement issued from Gekcel could itself append a `Gekko_Get()` call, and an `import<xlsx>gekcel;` issued from Gekcel could itself prepend a `Gekko_Put()` call?
- Perhaps CLS called from Gekcel should wipe out (clear) the Immediate Window?
- From Gekcel, it should be possible to write just `import<xlsx>;` instead of `import<xlsx>gekcel;`. Actually, at the moment, you can write anything as "filename" instead of `gekcel`.
- Other statements could decorate Excel cells, something like `prt <cell = 'B2'> 'Hello';` to print something to a particular Excel cell.
- An option to avoid overwrites. At the moment, when using SHEET or PRT together with `Gekko_Get()`, cells may be overwritten. Also an option to wipe out the contents of the current worksheet tab, before writing to it with `Gekko_Get()`.
- If possible, some functionality that sets up the Excel Immediate Window in a Gekko-like fashion, without the user having to open and drag the window, etc.

- Decorate Excel (Gekcel) with Gekko version number if somehow possible. To see the Gekko version number, you can issue a `Gekko "tell gekkoversion();"` from the Immediate Window (or from VBA). To see more info, you can alternatively issue a `Gekko "tell gekkoInfo('short5');"` (shows version, bitness, frequency, period, and working folder).
- Is it possible to put all of the files inside one Excel .xll file? And how does the logic actually work regarding the `Gekcel.xlsm`, `Gekcel.xll` files? When moving or copying the .xlsm file, it seems to somehow remember the location of the Gekcel.xll and other files, without storing this information in the `Gekcel.xlsm` file itself.
- Formatting: there are some issues with missing values turning up as 65535, but this is perhaps now fixed via the `Handle_Cells()` VBA function. Also, it would be nice if the Gekko formatting of values could carry over (4 decimals for levels, 2 decimals for percentages).

5.5 More: Gekko_Get/Put

The two subroutines `Gekko_Get()` and `Gekko_Put()` are not complicated at all, and if you have special requirements regarding how to fetch or store data, you may rewrite them to suit your particular needs.

The basic idea of both subroutines is that an internal two-dimensional VBA table (think of it as a matrix or array) is used as an interface from Gekko to Excel (`Gekko_Get()`) and from Excel to Gekko (`Gekko_Put()`). Below are these two subroutines, as they look like in Gekcel 3.1.14.

Gekko_Get (transfer from Gekko to Excel)

```
Public Sub Gekko_Get()
  Dim cells As Variant
  cells = CreateObject("Gekcel.COMLibrary").Gekko_Get()
  cells = Handle_Cells(cells)
  nrows = UBound(cells, 1) - LBound(cells, 1) + 1
  ncols = UBound(cells, 2) - LBound(cells, 2) + 1
  Set rValues = Application.Range("A1:A1").Resize(nrows, ncols)
  rValues.ClearContents
  rValues.Value = cells
End Sub
```

All of this except the line containing `"Gekcel.COMLibrary"` is normal VBA code. It first defines `cells` as a type that can receive a 2-d table of data from Gekko, which takes place in line 2. The right-hand side of line 2 technically returns an `object[,]` from Gekko, that is, an array of two dimensions, where the elements can be of any type. When Gekko is run from inside Excel, this array is produced by either the PRT or SHEET statement. Line 3 handles any missings (NaN) in the array, and the next two lines figures out how many rows and cols the array has. Next, a rectangular range of this size is selected (starting at the A1 cell in the current worksheet), this range is cleared, and finally the cells are "pasted" into the range.

Gekko_Put (transfer from Excel to Gekko)

```
Public Sub Gekko_Put()
  nrows = Range("A1").SpecialCells(xlCellTypeLastCell).Row
  ncols = Range("A1").SpecialCells(xlCellTypeLastCell).Column
  Set x = Application.Range("A1:A1").Resize(nrows, ncols)
  Dim x1() As Variant
  x1 = x.Value
  Dim temp As Variant
  temp = CreateObject("Gekcel.COMLibrary").Gekko_Put(x1, 1)
End Sub
```

All of this except the line containing `"Gekcel.COMLibrary"` is normal VBA code. The first two lines try to figure out the data dimensions of the current worksheet (using `xlCellTypeLastCell` to do this). In line 3, a rectangular range of this size is selected (this more or less corresponds to putting the cursor at A1 and hitting [Ctrl+A] to select the current data area). The VBA object `x1` contains the data and is a `object[,]` type, that is, an array of two dimensions, where the elements can be of any type. In the last line of the subroutine, this array is is transferred to Gekko. When Gekko is run from inside Excel, the array is consumed by calling the IMPORT<xlsx> method. It should be noted that in this transferal, integers, floating values and strings are accepted, but Excel dates may pose problems (if so, use strings or integers as dates, for instance `2023q3` or `2024`).

All in all, these subroutines are centered around a 2-dimensional table-like data representation (`object[,]`) of the data that is being fetched or stored, either like this:

	date1	date2	date3
var1	...	...	...
var2	...	...	...

or like this:

	var1	var2
date1	...	...
date2	...	...
date3	...	...

If you use this data representation as interface when talking to Gekko, you may use VBA to handle how and where the data areas are found, retrieved, filled, etc. VBA does that part of the job, not Gekko or Gekcel.

Part VI

6 Appendix

The following pages contain different appendices.

6.1 AREMOS translator details

The AREMOS translator (cf. the TRANSLATE statement) tries to translate AREMOS command files into corresponding Gekko program files.

This translator translates from AREMOS into Gekko 3.0 syntax.

First of all, Gekko recognizes AREMOS abbreviations, and will for instance recognize DE, DEL, DELE, DELET, and DELETE as all being the same DELETE command. Gekko has few abbreviations, because it may render program files hard to decipher.

The following commands are translated:

- ACCEPT has a note that the syntax is a bit different.
- ASSIGN x value #y ==> %x = %y.
- ASSIGN x integer #y ==> %x = %y. If the integer is date-like, a Gekko DATE is used instead, for instance "ASSIGN per1 integer 1966;" ==> "DATE per1 = 1966;".
- ASSIGN x string #y ==> %x = %y.
- ASSIGN x literal #y ==> %x = %y. If the literal is date-like, a Gekko DATE is used instead, for instance "ASSIGN per1 literal '1966a1;" ==> "DATE per1 = 1966A1;".
- CLOSEALL ==> RESTART. CLOSEALL is a procedure. A note is given that in some cases, CLOSE *; CLEAR; is a better replacement.
- CONVERT ==> COLLAPSE. CONVERT seems to be a procedure that does the same as COLLAPSE.
- COPY ... AS ... ==> COPY ... TO ... . Regarding COPY, AREMOS allows for instance "COPY d:var1 as e:;" with an 'e:' to indicate the destination databank. Gekko demands a '*' after the colon, so such a statement is translated into "COPY d:var1 to e:*;".
- EXCELEXPORT ==> SHEET. A note is given on syntax differences.
- EXCELIMPORT ==> SHEET<import>. A note is given on syntax differences.
- FOR: Gekko tries to guess the type of the loop variable, if it is FOR y = x1 TO x2 BY x3. Here, if x1 and x2 look like dates, Gekko will use FOR data y = ..., else if they look like normal values, Gekko will use FOR val y = ... Assign-variables starting with 'per' (for instance #per0) are always deemed to have date-flavor!
- GRAPH ==> PLOT.
- IF: AREMOS apparantly accepts IF-statements without parentheses, for instance "IF #v > 100; SERIES x = 200; END;". Gekko puts parentheses like this: "IF (#v > 100); SERIES x = 200; END;", to conform with Gekko syntax.
- INDEX: Syntax changed a bit, and options showbank=no and showfreq=no are set, to correspond with AREMOS.
- OBEY ==> RUN.
- OPEN<protect> ==> just OPEN (Gekko databanks are protected as default).
- OPEN<primary> is changed to OPEN<edit>
- PRINT ==> PRT.
- RESTORE ==> gets a not about using RUN, where .opt is added to filename if this extension is not there already.
- SET ==> OPTION.
- SET PERIOD ==> TIME.
- SET REPORT DECIMALS ... ==> Note that the syntax is: OPTION print fields ndec ... .

- SET REPORT COLUMNS ... ==> Note that the syntax is: OPTION print fields nwidth = ...
- SET SAVEFILE ... ==> gets a note on using PIPE or PIPE con.
- SERIES with inline if-then-else are decorated with a comment about using the iif() function. Note in general that "SERIES y = 100 rep *;" is not necessary in Gekko ("SERIES y = 100;" will do). Gekko will ignore such stray 'rep *', but they may clutter the program files. SERIES labels like "SERIES 'label' y = x;" are transformed into "SERIES <label = 'label'> y = x;".
- SOLVE ==> SIM.
- SPOOL ==> PIPE.
- STOP ==> EXIT.
- UNSPOOL ==> PIPE con.
- VIS ==> PLOT.

The translator keeps track of assign-vars, lists and matrices. Assign-vars get '%'-identifier, lists keep their '#'-identifier, and matrices get '#'-identifier (apart from this, please note that the Gekko matrix syntax is somewhat different from AREMOS matrix syntax -- the translator does not handle these differences).

In addition:

- '!' is translated into '//'
- Double quotes (") are translated into single quotes (')
- List operators + and * are translated into || and &&.
- Double '#' like '##x' have a note about their proper translation ('%{%x}' or '#{%x}')
- 'repeat' is changed to 'rep'
- Tries to eliminate spurious '|', for instance "PRINT a#i| + b#i|;" ==> "PRINT a%i + b%i;". AREMOS accepts '|' almost anywhere, so AREMOS programs get easily infected with too many of these. The elimination is heuristic.
- Numbers: Gekko 2.0 does not support '123.' to be the same as '123' or '123.0'. The reason is that Gekko uses '..' for ranges (for instance #m[2..4]), where a number like '123.' would interfere badly. So Gekko translates for instance '123.' into '123.0'.
- Functions: Gekko does not support for instance "VAL v = log (10);", note the blank between the log function and the parenthesis. So the translator removes such blanks for the functions log(), exp(), pow(), abs(), pch(), dlog().
- Logical operators: In AREMOS they may be separated by blanks, for instance '< =' instead of '<=' or '< >' instead of '<>'. Gekko removes such blanks.
- For lists, #a + #b ==> #a || #b, and #a * #b ==> #a && #b.
- Any '.cmd' is changed into '.gcm', for instance "RUN f.cmd;" ==> "RUN f.gcm;".
- Strings like 'the value is #x' are replaced with 'the value is {%x}'.
- Frequencies .a, .q and .m use '!' instead, for instance x.q ==> x!q.
- Variables on the left-handside get '%' or '#' if they are not series, for instance LIST m = a, b; --> LIST #m = a, b;
- sum(0, a, b) is changed into sum(a, b). Gekko does not have this problem.
- An option field like SERIES <2020 2030> y = 100; is moved to the right of y, so it becomes y <2020 2030> = 100;.
- The translator tries to set {}-curlies where they are needed (for instance around assign literals).

- Dynamic series like series `series x = x[-1] + 2;` are identified and decorated with `<dyn>` tag.

6.2 Gekko 2.0 translator details

The Gekko 2.0 to 3.0 translator (cf. the TRANSLATE statement) tries to translate Gekko 2.0/2.2/2.4/2.5.x program files into corresponding Gekko 3.0/3.1.x program files.

The following statements are translated:

- `COMPARE` has a note about changed syntax
- `COLLAPSE` has `.` changed into `!`
- `DOWNLOAD` has a note about quotes
- `FOR` has type and type symbol added.
- `INDEX a*b m` --> `INDEX a*b TO #m`. Added `<showbank=no showfreq=no>` option.
- `IMPORT/EXPORT` have `<all>` added to the option field, if the option field does not indicate two dates. Also, `IMPORT<ser>` --> `IMPORT<flat>`.
- `(series pp, series qq) = laspchain(...)` --> `pp = laspchain(...).p; qq = laspchain(...).q;`
- `LIST listfile m = ... ;` --> `#(listfile m) = ... ;`
- `NAME` changed to `STRING`.
- `hpfilter()` and `unpack()` have `CREATE` removed.
- `LIST`: `&+` --> `||`, `&*` --> `&&`, `&-` --> `-`
- `LIST<direct>` is removed.
- `LIST`: `prefix` --> `.prefix()`, `suffix` --> `.suffix()`, `trim` --> `.unique()`, `sort` --> `sort()`, `strip` --> `replaceinside()`. Use of `strip` in list has a new syntax explanation
- `MATRIX`: `||` --> `;`
- `VAL`, `STRING`, `DATE` have `%` added on left-hand side and statement removed.
- `LIST` and `MATRIX` have `#` added on left-hand side and statement removed.
- `SERIES` has statement removed.
- `SERIES`: Operator `^` --> `^=`, `%` --> `%=`, `+` --> `+=`, `*` --> `*=`, `#` --> `#=`.
- `SERIES #m = ... ;` --> `{#m} = ... ;`
- `SERIES y = 100 rep *;` --> `y = 100;`. Superfluous `rep *` are often inherited from AREMOS.
- `SERIES x = x[-1] + 1;` --> `x <dyn> = x[-1] + 1;`
- `SHOW` renamed to `PRT`.
- Wildcards like `[a*b?c]` are changed into `['a*b?c']` or `{'a*b?c'}` depending on context. Ranges like `[a1..b5]` are changed into `['a1'..'b5']` or `{'a1'..'b5'}`.
- Variables like `%(...)` or `#(...)` have a note that it is better to use {}-curlies. For instance, `%(a%b)` can be written as `%a{%b}`.

In addition:

- In general, the translator tries to add `{}`-curlies around `%x` or `#x` variables, when these curlies are not already present but required in Gekko 3.0. To do this, the translator may look at the definitions of `%x` (whether string, name, date or val) or `#x` (whether list or matrix). If the variable is explicitly of date, val or matrix type, no `{}`-curlies are added. In other cases, there may be an element of guessing, depending upon the particular statement or function used, so in some cases, `{}`-curlies may be added in places where they should not. To determine the type of `%x` or `#x`, only the current program file is looked at (this may pose a problem if the variable is defined somewhere else). All `{i}` are replaced with `{%i}`. The translator never removes existing `{}`-curlies.

- Index `[0]` --> `length()`, for instance `#m[0]` --> `#m.length()`.
- Paths like `data\%x\%y` are transformed into `data\{%x}\{%y}`, for instance for READ etc.
- A string like `'ab %s cd'` is transformed into `'ab {%s} cd'` (string interpolation). Also translates `~` and `|` inside strings.
- `avg()` and `sum()` will convert for instance `sum(#m)` into `sum({#m})`. The summing `sum(#i, x[#i])` will not be translated correctly, however. It will become `sum({#i}, x[#i])`, where it should not be altered.
- `avgt()` --> note about `<>` syntax for local time
- `difference()` --> `except()`
- `endswith()` --> note about case-insensitive in 3.0
- `fromseries()` --> note about the fact that the first argument must be series, not string.
- `hpfilter()` --> note about `<>` syntax for local time
- `pack()` --> note about `<>` syntax for local time
- `piece()` --> `substring()`
- `replace()` --> note about case-insensitive in 3.0
- `search()` --> `index()`
- `startswith()` --> note about case-insensitive in 3.0
- `strip()` --> `replace()`
- `sumt()` --> note about `<>` syntax for local time
- `trim()` --> `strip()`
- `unpack()` --> note about `<>` syntax for local time
- An option field like `series <2020 2030> y = 100;` becomes `y <2020 2030> = 100;`.
- Differences in globals options (`OPTION` statement) are handled.
- You may optionally keep types when translating assignments, so that `VAL x = 100;` becomes `VAL %x = 100;` instead of just `%x = 100;`. This is controlled with the `<keeptypes>` option when calling the translator. Applies to SERIES, VAL, DATE, STRING, LIST and MATRIX.
- All VAL, DATE, STRING, LIST and MATRIX definitions have `global:` added. For instance, `VAL x = 100;` becomes `global:%x = 100;` or `VAL global:%x = 100;`. And `LIST m = a, b;` becomes `global:#m = a, b;` or `LIST global:#m = a, b;`.

Since the databank logic and simulation parts are not changed between Gekko 2.x.x and Gekko 3.x.x, translation is generally not too difficult regarding the more basic kinds of statements. If using more complicated stuff from Gekko 2.x.x, the translation may be wrong. In that case, please consult the Gekko 3.x.x documentation to see how the statement looks in the new syntax.

For non-series, the translator will put a `global:` on the variable. These can often be omitted, but in Gekko 2.x, non-series did not live in databanks, hence the use of `global:` in the translation.

6.3 Assignments

Assignments are of the form `... = ... ;` with a `=` symbol, for instance `y = 2 * x;`, `%y = 2 * %x;`, `#m = a, b;`, and so on. So the right-hand side (RHS) expression is assigned to the left-hand side (LHS) variable. There may also be a type indicator, like `SERIES y = 2 * x;`, stating that `y` should become a series type. There are the following type indicators:

- series: `SERIES y = ...;`
- val: `VAL %y = ...;`
- date: `DATE %y = ...;`
- string: `STRING %y = ...;`
- list: `LIST #y = ...;`
- map: `MAP #y = ...;`
- matrix: `MATRIX #y = ...;`
- var: `y = ...;` or `%y = ...;` or `#y = ...;` or `VAR y = ...;` or `VAR %y = ...;` or `VAR #y = ...;`

- Note that var type in the descriptions below include the common case where the type is not explicitly stated.
- In the following, LHS means left-hand side
- In the following, RHS means right-hand side

There are the following rules regarding assignments.

LHS name starts with `%` symbol (scalar).

- Fails if there is LHS type indicator and this is not val, date, string or var type.
- `%y = ...timeless...;`. The RHS is a timeless series. If the LHS type is val or var, the LHS becomes a val with the value of the timeless series.
- `%y = ...val...;`. The RHS is a val. If the LHS type is val or var, the LHS becomes the RHS val. If the LHS type is a date, Gekko tries to convert the val into a date (for instance, `2020` into `2020a`).
- `%y = ...string...;`. The RHS is a string. If the LHS type is string or var, the LHS becomes the RHS string.
- `%y = ...date...;`. The RHS is a date. If the LHS type is date or var, the LHS becomes the RHS date. If the LHS type is a date, Gekko tries to convert the val into a date (for instance, `2020` into `2020a`).
- `%y = ...1x1 matrix...;`. The RHS is a 1x1 matrix. If the LHS type is val or var, the LHS becomes the RHS matrix element.

Left-hand side name starts with `#` symbol (collection).

- Fails if there is LHS type indicator and this is not list, map or matrix.
- `matrix #y = ...series...;`. The RHS is a normal series. If the LHS type is matrix, the LHS becomes a matrix corresponding to the values inside the series. If the RHS is a timeless series, and the LHS type is matrix, the LHS becomes a matrix corresponding to the value inside the series (duplicated for each period in the current time period). If the type is not stated, for instance `#y = a;`, Gekko will not

auto-convert the series `a` into a list of values `#y` (this could create confusion in relation to naked list definitions).

- `#y = ...list...;`. The RHS is a list. If the LHS type is list or var, the LHS becomes a list corresponding to the RHS list.
- `#y = ...map...;`. The RHS is a map. If the LHS type is map or var, the LHS becomes a map corresponding to the RHS map.
- `#y = ...matrix...;`. The RHS is a matrix. If the LHS type is matrix or var, the LHS becomes a matrix corresponding to the RHS matrix.

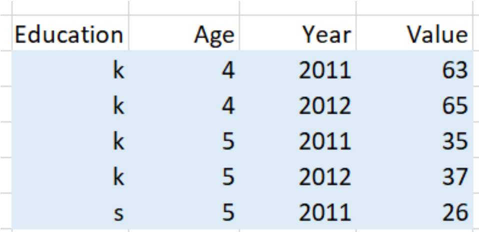
Left-hand side does not start with `%` or `#` symbol (series).

- Fails if there is LHS type indicator and this is not series
- `y = ...series...;`. The RHS is a series. If the LHS type is series or var, the LHS becomes a series corresponding to the RHS series (frequencies must conform).
- `y = ...val...;`. The RHS is a val. The val is duplicated into all of the series observations.
- `y = ...list...;`. The RHS is a list. The list elements are put into the series observations (the list elements must be of val type, and if the last list element is stated with `rep *`, the list is expanded to fit with the time period).
- `y = ...1x1 matrix...;`. The RHS is a 1x1 matrix. The matrix element is duplicated into all of the series observations.
- `y = ...nx1 matrix...;`. The RHS is a nx1 matrix. If the size of the matrix fits the time period, the matrix elements are put into the series observations.

Note in general that when assigning variables, the left-hand side is always assigned to a *copy* of the right-hand side (pass by value, not pass by reference).

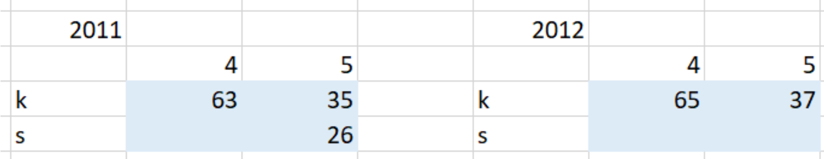
6.4 Missings

Gekko timeseries, including array-timeseries, can be represented via dimensions, where one of the dimensions is time. In order to understand the concepts, a practical example is used throughout this section. The data contains information on 4- and 5-year old children, counted at the beginning of August each year, cf. the following table:



Education	Age	Year	Value
k	4	2011	63
k	4	2012	65
k	5	2011	35
k	5	2012	37
s	5	2011	26

This table can be thought of as representing "rows" of data, or rows of multidimensional "observations" (in SQL, such a structure is called a "table"; in data science software it is called a "data frame"). The columns apart from the value (count of children) can be thought of as dimensions, in this case education (kindergarden or school), age, and year. Another way of visualizing at the table could be this (for each of the two years):



2011				2012		
	4	5			4	5
k	63	35		k	65	37
s		26		s		

As it can be seen, there are no data corresponding to 4-year old schoolchildren (they do not go to school), and regarding 5-year old schoolchildren in 2012, the value seems to be missing. This may be because the data was accidentally not updated, but if not, we would have preferred an explicit value 0 for this combination (in GAMS: the special value `eps`, more about this later on).

In Gekko, the data could be represented as the following timeseries:

```
reset; time 2011 2012;
#e = k, s;
#a = ('4', '5');
x = series(2);  //2 dimensions
x[k, 4] = 63, 65;
x[k, 5] = 35, 37;
x[s, 5] = 26, m(); //m() = missing
prt <n> x;
```

```
               x[k, 4]        x[k, 5]        x[s, 5]
  2011         63.0000        35.0000        26.0000
  2012         65.0000        37.0000              M
```

Note that `x` is two-dimensional, since there are two dimensions apart from the time dimension. Note also that using `prt <n> x[#e, #a];` would fail with an error (missing sub-series/element `x[s, 4]`). We will return to that question later on. Each column in the above print represents an array-subseries, where `x` is the array-series, and for instance `x[k, 4]` is an array-subseries belonging to `x`.

In GAMS, the same would look like this:

```
set e /k, s/;
set a /4, 5/;
set t /2011, 2012/;
parameter x(e, a, t);
x('k', '4', '2011') = 63;
x('k', '4', '2012') = 65;
x('k', '5', '2011') = 35;
x('k', '5', '2012') = 37;
x('s', '5', '2011') = 26;
display x;
```

```
           2011        2012
k.4      63.000      65.000
k.5      35.000      37.000
s.5      26.000
```

Here, the `x` variable (defined as a parameter) is three-dimensional, since in GAMS, the time dimension is just a dimension among the other dimensions. The print looks similar, but it should be noted that whereas in Gekko the data is represented by three building blocks (the three array-series `x[k, 4]`, `x[k, 5]`, and `x[s, 5]`), in GAMS the data corresponds to the table rows, that is, seven building blocks, one for each value shown in the GAMS print (the combination `s.5` in 2012 does not exist).

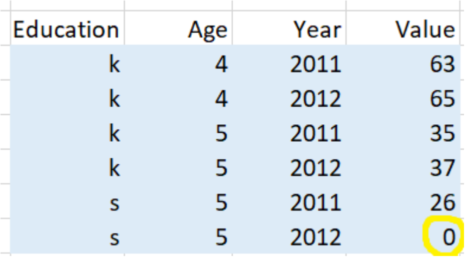
Missing series or missing data value?

From the Gekko and GAMS prints, it is seen that the data value `x[s, 5][2012]` is missing, whereas it is a bit less evident to see that the sub-series `x[s, 4]` does not exist at all.

As indicated above, GAMS does not make the distinction between missing series or missing data values. GAMS treats the elements of a variable `x` as simply rows of multidimensional data, where the time dimension is not special. In Gekko, the fact that there are no rows corresponding to the combination `s, 4` entails that the array-series `x` does not contain any array-sub-series `x[s, 4]`, so that particular sub-series is a missing series (does not exist inside `x`).

In contrast, regarding the data value `x[s, 5][2012]`, the combination `x[s, 5][2011]` is present among the data rows, and therefore the sub-series `x[s, 5]` does exist (with missing value for the year 2012).

As it is seen, Gekko distinguishes between missing series and missing data values, and in many cases, this distinction can be practical. In the concrete case, if the data is supposed to have been updated for the period 2011-12, we would not expect an array-subseries to contain a missing data value like `x[s, 5][2012]`. It *could* be the case that there are no 5-year olds in school in 2012, for instance if the school is closing for new pupils. But in that case, it would have been sensible to add an explicit row with the value 0 to indicate that, instead of just skipping the row:



Education	Age	Year	Value
k	4	2011	63
k	4	2012	65
k	5	2011	35
k	5	2012	37
s	5	2011	26
s	5	2012	0

Because the default data value for a Gekko series is missing value, with an explicit 0, the user would know that someone actually set that value to 0 -- that is, updated it (in GAMS, the special value `eps` could be used to indicate a "real" updated 0). Setting such missing data values to 0 in Gekko does not put any burden on processor or memory. In contrast, creating an array-subseries `x[s, 4]` in Gekko and filling it with missing values (or 0's) would be a waste of processor and memory ressources. This particular combination (4-year olds in schools) simply does not exist, and should therefore not take up any memory space. Such cases where some sub-series are non-existing are very common, because data is often sparse in the non-time dimensions. For instance, "education" may span nursery, kindergarden, school, secondary education, etc., but for each of these, there is typically only a limited age range. For instance, storing array-subseries full of missing data values representing 10-year olds in kindergarden does not make much sense, and is certainly a waste of space.

There are many cases where a missing data value is always to be interpreted as missing in the sense of "needs to be updated" or "forgotten". An example could be a timeseries representing GDP, which simply does not make sense with the value 0. Another example is for instance the value of deliveries from one aggregate sector to another. If an array-subseries containing such data has data in 2011 and 2013, but a missing data value in 2012, this is probably an error. The nominal value of input-output deliveries between sectors do not normally change from non-zero to zero from year to year, so a missing value probably means that someone should update that value. Still, regarding deliveries between small sectors, such a value *may* become exactly 0 in some years, but representing this with a missing value should be considered bad practice. In that case, the user should by all means use a real zero to indicate that the data has been updated and is not just forgotten. All in all, the sparsity of data is often less prevalent in the time dimension than in other dimensions.

Missings, and three Gekko options

As noted above, Gekko distinguishes between missing series and missing data values, because this distinction is sometimes useful, for instance when summing up or printing array-series. When for instance summing up the nominal value of input-output deliveries between aggregated sectors, one of the deliveries may contain a missing value for, say, the year 2012. This may be unexpected, and as a consequence, the sum will contain a missing value for that year, too. In this way, a probable error can be identified and corrected, instead of just treating the value as if it had been 0. In contrast, when summing up data like this, a missing array-subseries is to be expected, because not all sectors deliver goods to all other sectors. So such a missing array-subseries can just be skipped (treated as 0), when summing up or printing, and therefore the distinction between missing series and missing data values can be useful.

To return to the kindergarden/school example, there is a missing array-subseries `x[s, 4]`, corresponding to 4-year old schoolchildren. Such a series is absent, simply because there are no 4-year olds in the school classes. So in that sense, it is correct to treat `x[s, 4]` is implicitly containing 0's. In contrast, the missing data value `x[s, 5][2012]` is suspect. It could mean that the school has stopped admitting new 5-year old pupils in 2012, but is more probably an error (missing update).

In Gekko, there are the following options to control how missing series and missing data values are treated (default values are shown, too):

- `option series array calc missing = error;`
- `option series array print missing = error;`
- `option series data missing = m;`

1. The first option (`calc`) manages what happens if a whole array-subseries is missing, for instance in the expression `x[s, 4]` or `sum((#e, x[#e, 4])`, where `#e`

contains the element `s`. The `calc` option deals with "controlled" indexes, that is, either a direct index like the `s` in `x[s, 4]`, or a list index like `x[#e, 4]`, where `#e` is controlled by for instance a sum-function.

2. The second option (`print`) manages "uncontrolled" indexing of array-subseries, for instance `prt x[#e];`, where `#e` is not "controlled" via for instance a sum function. In that case, the elements of `x[#e]` are unfolded into columns (instead of being summed), and the option controls what happens if some of the "uncontrolled" sub-series do not exist.
3. The third option (`data`) manages what happens if a series contains missing data values.

Regarding the first and second options: note that a statement like `y = x[#e];` is not legal. Using list names as indexes in general assignments is only possible if the list is controlled in a sum-function, but in the PRT/PLOT statement, a statement like`prt x[#e];` with a free-floating (non-controlled) `#e` list is implicitly converted into `prt unfold(#e, x[#e]);` where the unfold()-function puts the elements into a list for subsequent printing/plotting as columns.

The following sections explains this in more details, with examples.

1. First option: calc missing

We will first take a look at `option series array calc missing`. This option deals with controlled sets/lists, for instance a sum like `sum((#e, #a), x[#e, #a])`, looping over the combinations of the sets/lists `#e` and `#a`. Consider the following, building upon the Gekko-code shown at the start of the section:

```
prt <n> sum((#e, #a), x[#e, #a]);
```

This will fail with an error, because the array-subseries `x[s, 4]` does not exist. With `option series array calc missing = m;`, Gekko can be asked to treat such an array-subseries as if it contained missing values instead, but in that case the sum will just be missing for each year. Instead, treating the missing sub-series as 0's does the trick:

```
option series array calc missing = zero;
prt <n> sum((#e, #a), x[#e, #a]);

             sum((#e, #a
          ), x[#e, #a])
  2011         124.0000
  2012                M
```

Now there is at least a value in 2011, whereas the value in 2012 is missing, since `x[s, 5][2012]` has a missing data value. But the missing array-subseries `x[s, 4]` is ignored as expected.

2. Second option: print missing

Next, we will take a look at `option series array print missing`. As mentioned above, this option affects uncontrolled sets/lists,

```
prt <n> x[#e, #a];
```

This will abort with an error, because the array-subseries `x[s, 4]` does not exist. Now, you may try the following:

```
option series array print missing = m;
prt <n> x[#e, #a];
```

```
                x[k, 4]        x[s, 4]        x[k, 5]        x[s, 5]
  2011          63.0000              N        35.0000        26.0000
  2012          65.0000              N        37.0000              M
```

Here, the missing x[s, 4] is shown as missing data values (actually 'N' instead of 'M' to indicate that the series is non-existing). Else, 0's can be printed instead, for instance:

```
option series array print missing = zero;
prt <n> x[#e, #a];
```

```
                x[k, 4]        x[s, 4]        x[k, 5]        x[s, 5]
  2011          63.0000         0.0000        35.0000        26.0000
  2012          65.0000         0.0000        37.0000              M
```

Here, it is hard to tell from the print whether the sub-series `x[s, 4]` exists or not. Finally, the column may be skipped altogether with `skip`, for instance:

```
option series array print missing = skip;
prt <n> x[#e, #a];
```

```
                x[k, 4]        x[k, 5]        x[s, 5]
  2011          63.0000        35.0000        26.0000
  2012          65.0000        37.0000              M
```

This corresponds to what GAMS does when printing, and is practical for sparse data.

3. Third option: data missing

Finally, there is `option series data missing`. This option is normally set to `m` (missing), but if set to `zero`, any expression containing a databank timeseries (either a normal series or array-subseries) with a missing data value will be interpreted as if the data value was 0. We may try the following, again building upon the example at the start of this page:

```
option series array calc missing = zero;
option series data missing = zero;
prt <n> sum((#e, #a), x[#e, #a]);
```

```
          sum((#e, #a
        ), x[#e, #a])
```

```
  2011       124.0000
  2012       102.0000
```

The first option handles the missing sub-series `x[s, 4]`, whereas the second option handles the missing data value `x[s, 5][2012]`. Another example could be a print without summing. In that case, the `#e` and `#a` lists are non-controlled, and `option series array calc missing` will not have any effect. Instead, `option series array print` could be used:

```
option series array print missing = zero;
option series data missing = zero;
prt <n> x[#e, #a];
```

```
              x[k, 4]        x[s, 4]        x[k, 5]        x[s, 5]
  2011        63.0000         0.0000        35.0000        26.0000
  2012        65.0000         0.0000        37.0000         0.0000
```

Instead, the following option is perhaps more suitable:

```
option series array print missing = skip;
option series data missing = m; //default
prt <n> x[#e, #a];
```

```
              x[k, 4]        x[k, 5]        x[s, 5]
  2011        63.0000        35.0000        26.0000
  2012        65.0000        37.0000              M
```

Here, the missing sub-series `x[s, 4]` is skipped, and the missing data value `x[2, 5][2012]` is shown as `M`.

Combining the options, and the <missing = ignore> option

The above-mentioned options can be combined:

```
option series array print missing = skip;
option series array calc missing = zero;
option series data missing = zero;
prt <n> sum((#e, #a), x[#e, #a]), x[#e, #a];
```

```
          sum((#e, #a
        ), x[#e, #a])        x[k, 4]        x[k, 5]        x[s, 5]
  2011       124.0000        63.0000        35.0000        26.0000
  2012       102.0000        65.0000        37.0000         0.0000
```

The first column is the sum of the next three colums, and the missing sub-series `x[k, 5]` and missing data value `x[2, 5][2012]` are handled as if they were 0. Note that in the first column, `#e` and `#a` are controlled by the sum function (and thus managed by the `array calc` option). The column corresponding to `x[s, 4]` is skipped, because it does not exist, and this is managed by the `array print` option for uncontrolled indexes.

Since a print like the above is often practical if missing series and missing data values are to be ignored, Gekko contains the local option `<missing = ignore>` for both PRT/PLOT, but also for assignments. The local option sets the following options temporarily, and reverts them after the statement:

```
option series array print missing = skip;
option series array calc missing = zero;
option series data missing = zero;
```

Example:

```
prt <n missing = ignore> sum((#e, #a), x[#e, #a]), x[#e, #a];
y <missing = ignore> = sum((#e, #a), x[#e, #a]);
prt <n> y;
```

```
         sum((#e, #a
        ), x[#e, #a])          x[k, 4]          x[k, 5]          x[s, 5]
  2011       124.0000          63.0000          35.0000          26.0000
  2012       102.0000          65.0000          37.0000           0.0000

                    y
  2011       124.0000
  2012       102.0000
```

The first of the above options is set to `skip`, not `zero`, to avoid printing of superfluous uncontrolled array-subseries that do not exist.

Illegal mathematical operations

`option series data missing` only affects data values directly accessed from a timeseries (possibly array-subseries) residing in a databank, and not for instance the results of expressions, functions, etc. Example:

```
reset; time 2011 2012;
option series data missing = zero;
y1 = -2, 2;
ly2 = log(y1);
prt <n> log(y1), ly2;
```

```
              log(y1)              ly2
  2011              M           0.0000
  2012         0.6931           0.6931
```

In this print, with `option series data missing = zero;`, `log(y1)` has a missing in 2011, whereas `ly2` is 0 in 2011. The reason is that in `log(y1)`, the missing value does not originate from a missing data value in a databank series (rather, the missing values stems from the mathematical operation `log(-2)`), whereas in the `y2` case, the missing value does originate from a data value in a databank series (`ly2`, stored in the first-position databank, has a missing value in 2011). With `option series data missing = m;`, we would get

```
              log(y1)            ly2
  2011              M              M
  2012         0.6931         0.6931
```

In GAMS, we get the following:

```
set t /2011, 2012/;
parameter y1(t);
parameter ly2(t);
y1('2011') = -2;
y1('2012') = 2;
ly2(t) = log(y1(t));
display ly2;  //display log(y1) or log(y1(t)) is not legal
```

```
2011  UNDF,     2012 0.693
```

So GAMS calculates and stores `ly2` in 2011 as 'undefined' (UNDF), not 0.

Note

See also this Gekko blog post on array-series: http://t-t.dk/gekko/array-series.

6.5 Indexing

Indexing means the use of `[]`-brackets to *index* a variable, like for instance `x[a]` (the `'a'` element of the array-series `x`). In some cases *dot notation* is possible too: for instance, `#m.a` can be used as synonym for the map element `#m['a']`, and `x.1` can be used as synonym for the lag `x[-1]`. Gekko has 7 variable types, and in the following it will be shown how indexing can be used for each of these types.

Auto-quoting
Regarding indexing in Gekko, beware that using a simple non-quoted variable name inside `[]`-brackets is always interpreted as if it was quoted. For instance, `x[a]` is interpreted as `x['a']`, and hence some typing of `'`-quotes can be omitted. This convenience only applies if the contents of the `[]`-bracket is 1 simple variable name (for instance `abc`, `_xy`, `px45` and the like). The name cannot contain other symbols than alphanumeric or underscore if it is to be auto-quoted (and it cannot start with a digit).

Value

A value like `%v = 123;` cannot be indexed.

Date

A date like `%d = 2023q3;` cannot be indexed.

String

A string like `%s = 'abc';` can be indexed, but only with integers and ranges `..`.

```
%s = 'abc';
prt %s[2]; //'b'
prt %s[2..3]; //'bc'
prt %s[2..]; //'bc'
```

Such indexing picks out an element or a range of elements, see more here. If you try to use a non-integer index, Gekko will issue an error. For instance, `%s[b]` or `%s['b']` will fail. To find substrings inside a string, you can use the index() function.

Series

For a normal series `x`, you can use indexing to pick out a particular observation value, like for instance `x[2023q3]`. In addition, you can use lags or leads, like `x[-1]` or `x[+1]`. Note that for lags or leads, a `-` or `+` must always be the first character of the index. To lag a normal series, you may alternatively use dot-notation `x.1` instead of `x[-1]`.

For an array-series `x`, you can use notation like `x[a]` or `x['a']` (if `x` is 1-dimensional) to pick out the subseries `'a'` inside `x`. You may also use lists of strings as indexes, for instance:

```
x = series(1); //1-dimensional
x[a] = 2; x[b] = 3; x[c] = 4;
#m = a, c;
prt x[#m];  //prints out x[a] and x[c]
x[#m] = 5;  //changes values of x[a] and x[c]
```

For an array-series `x`, you cannot (at the moment at least) use for instance dot notation `x.a` as synonym for `x['a']`. This is to avoid confusion.

For an array-series `x`, you cannot use for instance `x['a*']` to search for subseries.

List

Lists can be indexed in different ways. You may use integers and ranges `..` to select sub-lists. For instance `#m[2]` to get element #2, `#m[2..3]` or `#m[2..]` to get a range of elements. For multidimensional nested lists (lists inside lists), there are some special Python-inspired rules regarding such 'slices', cf. the descriptions here.

On a list of strings, if you use a string as index, you essentially ask if the list contains the string (case-insensitive), and Gekko returns 0 (false) or 1 (true). Or you can use a wildcard or range, in which case Gekko returns a list of hits. Examples:

```
#m = ax, ay, bz, aw2; //four strings
prt #m['az'], #m['bz']; //0 and 1
prt #m['a*']; //returns the list ax, ay, aw2
prt #m['a?']; //returns the list ax, ay
prt #m['aw'..'ax']; //returns the list ax, aw2
```

Matrix

Matrices can only contain values, and you may use integers and ranges `..` to select sub-matrices.

```
#m = [1, 2, 3; 4, 5, 6; 7, 8, 9]; //3x3 matrix
prt #m[2, 3]; //6
prt #m[2..3, 1..2]; //[4, 5; 7, 8]
prt #m[2.., ..]; //[4, 5, 6; 7, 8, 9]
```

Map

Maps can be thought of as mini-databanks. To pick out a series `a` from a map `#m`, you can use `#m['a']`, or the shorter dot-notation `#m.a`. Examples:

```
#m = (a = 100, %d = 2023q3); //a map with a series and a date
inside
prt #m['a'], #m.a; //prints out the series a inside #m (note: #m[a]
can be used for series)
prt #m['%d'], #m.%d; //prints out the date %d inside #m (note: #m[%
d] will fail)
```

At the moment, no other form of indexing is possible for maps. If a databank `m` contains the series `a`, it can be referenced by means of the colon syntax `m:a`, and the dot syntax `#m.a` for maps can be thought of as being analogous to the colon.

It is planned to offer wildcard and range searching for maps, for instance `#m['a*']`, but this has not been implemented yet. For such wildcards, the same logic as databank search will be used (special handling of `%`, `#` and `!` characters).

6.6 Data tracing

Performance and memory/file considerations

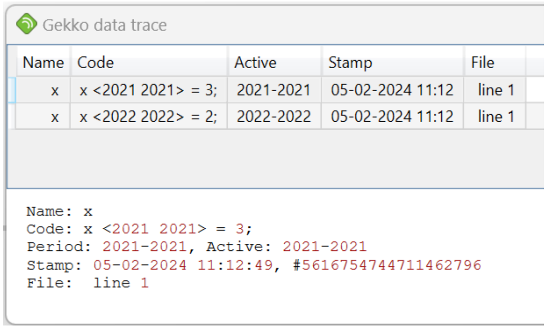
Data tracing is activated per default in Gekko >= 3.1.16. This tracing has some performance costs, and execution time used on data tracing is shown as a percentage after each Gekko job (often, this percentage is quite low). The .gbk databanks become larger, depending upon the number of data traces. You may look into a .gbk databank file if you open it as a .zip file: the `trace.data` file inside stores all traces. If you prefer, you can just remove this file manually from the `gbk` .zip file to get rid of all traces. Alternatively, use READ/WRITE/OPEN with option `<trace = no>`.

NOTE: When running the same series statements again and again iteratively (for instance, iterating over rows and columns in an input-output table until convergence), you may consider switching off data tracing with `option databank trace = no;`. For such solver-like scenarios, data tracing may slow down the calculations needlessly.

To switch off data tracing, set this option: `option databank trace = no;`

Data tracing uses time-shadowing, that is, a trace may be deleted if the time periods it covers are completely "shadowed" (replaced) by one or more newer traces. For instance with `x <2021 2022> = 1; x <2022 2022> = 2;`, the series `x` contains two traces: trace *t1* is defined for `2021`, and trace *t2* is defined for `2022`. In that sense, trace *t2* "shadows" parts of trace *t1*. If now a third statement `x <2021 2021> = 3;` is added, trace *t1* is removed completely since only trace *t2* and *t3* are relevant. Cf this:

```
reset;
x <2021 2022> = 1; //trace t1
x <2022 2022> = 2; //trace t2
x <2021 2021> = 3; //trace t3 (trace t1 is now irrelevant)
trace2 x;
```



Gekko data trace

Name	Code	Active	Stamp	File
x	x <2021 2021> = 3;	2021-2021	05-02-2024 11:12	line 1
x	x <2022 2022> = 2;	2022-2022	05-02-2024 11:12	line 1

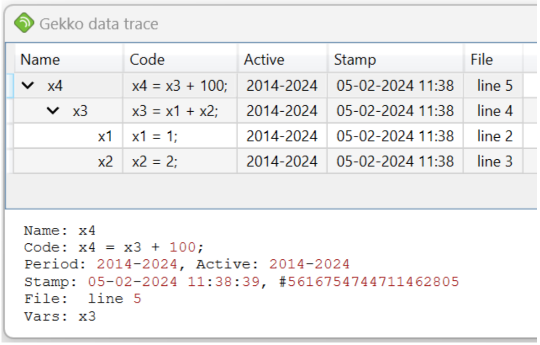
```
Name: x
Code: x <2021 2021> = 3;
Period: 2021-2021, Active: 2021-2021
Stamp: 05-02-2024 11:12:49, #5616754744711462796
File:  line 1
```

The trace `x <2021 2022> = 1;` has been "time-shadowed" and thus removed. For each trace, the "Active" field describes which periods out of the original period have not been "shadowed" by subsequent traces.

Traces are a graph

Nested data traces may seem like a tree structure, but they are really a graph. Consider this example:

```
reset;
x1 = 1;
x2 = 2;
x3 = x1 + x2;
x4 = x3 + 100;
tracestats2();
trace2 x4;
```

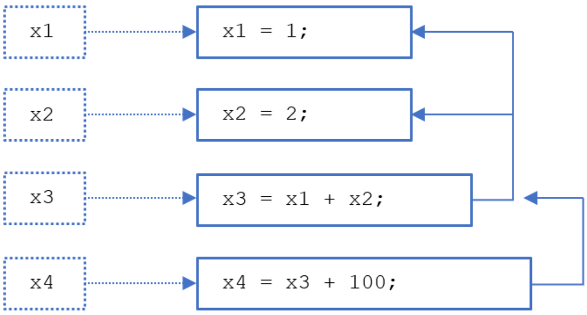


Gekko data trace

Name	Code	Active	Stamp	File
x4	x4 = x3 + 100;	2014-2024	05-02-2024 11:38	line 5
x3	x3 = x1 + x2;	2014-2024	05-02-2024 11:38	line 4
x1	x1 = 1;	2014-2024	05-02-2024 11:38	line 2
x2	x2 = 2;	2014-2024	05-02-2024 11:38	line 3

```
Name: x4
Code: x4 = x3 + 100;
Period: 2014-2024, Active: 2014-2024
Stamp: 05-02-2024 11:38:39, #5616754744711462805
File:  line 5
Vars: x3
```

When unfolding the traces of `x4`, it is seen that `x4` depends upon `x3`, which in turn depends upon `x1` and `x2`. If we depict the timeseries objects as dotted boxes and the traces a solid boxes, we can depict the traces as follows:



Starting from the series `x1`, this will only show the trace `x1 = 1;`. Similarly regarding series `x2`. Then series `x3` has the trace `x3 = x1 + x2;`, but also the sub-traces `x1 = 1;` and `x2 = 2;`. The series `x4` has the trace `x4 = x3 + 100;`, the sub-trace `x3 = x1`

`+ x2;` and the sub-sub-traces `x1 = 1;` and `x2 = 2;`. Thus, for instance, the trace `x1 = 1;` can be reached in three ways: from series `x1` itself, from series `x3` (sub-trace) and from series `x4` (sub-sub-trace).

All the four traces have depth 0, because they are all directly connected to a timeseries, cf. the result of `tracestats2();`:

```
Databank 'Work' has 4 series with 4 reachable traces in total.
--> depth: 0, traces: 4
Traces of depth 0 connect directly to a series/array-series in the databank.
```

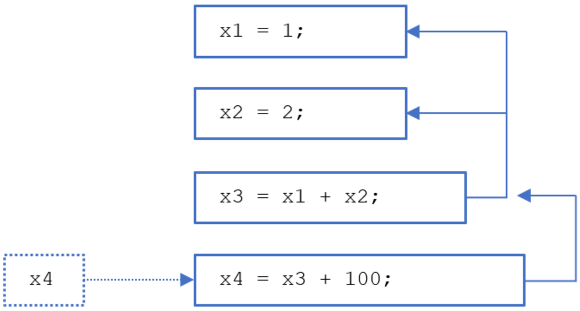
Now we can try to delete the first three series before showing traces:

```
reset;
x1 = 1;
x2 = 2;
x3 = x1 + x2;
x4 = x3 + 100;
delete x1, x2, x3;
tracestats2();
trace2 x4;
```

The trace viewer shows the same contents, but `tracestats2()` shows something different:

```
Databank 'Work' has 1 series with 4 reachable traces in total.
--> depth: 0, traces: 1
--> depth: 1, traces: 1
--> depth: 2, traces: 2
Traces of depth 0 connect directly to a series/array-series in the databank.
A trace of depth 1 connects to a trace of depth 0, and so on.
```

There are still 4 reachable traces, but only 1 at depth 0. This is because the traces now looks like this:



The series objects `x1`, `x2` and `x3` are removed, but their traces live on because of their "historical" impacts on series `x4`. In general, it pays off to think of traces as an interconnected graph structure, where many "paths" may lead to the same trace object.

Data traces typically accumulate. If the data trace parts of .gbk databanks become too large when running Gekko jobs, pruning/discarding some of these or alternatively sending them off to long-term storage may become an option. Such a feature is not built into the current Gekko version (3.1.16), but the building blocks are ready. Each Gekko data trace is universally unique, since it has an ID consisting of a time stamp (to millisecond precision) and a large random number (this is what is shown in the "Stamp" section of the data trace viewer). Because of this uniqueness, data traces can be removed from one databank and sent off to some other storage/backup databank, with the option of always being able to restore the traces again if needed.

When pruning or storing data traces, criteria for doing this could be depth (as defined above, since it depicts the level of indirection), time stamp (newer traces are often most important) or active periods (if the remaining active period for a trace is old, it may not be too important to keep).

Archive databank

Traces may make a `gbk` databank large and/or slow to read/write.

TODO: Write about how traces can be pruned off `gbk` databanks and stored in a "archive" trace databank.

6.7 GAMS details

Gekko implements syntax that resembles GAMS, in order to facilitate interoperability between the two languages. This section will be augmented, for now only the basics are described.

Translate from GAMS to Gekko

When using MODEL<gms>, Gekko will try to translate the raw (unfolded) GAMS equations into equivalent Gekko statements. The translated equations are shown when using e.g. DISP or DECOMP.

But you may also use this translator more interactively. Simply mark some GAMS code (statement or equation) in your editor, copy it (Ctrl+C), and then in the Gekko main window right-click and select `Paste special` followed by `Paste GAMS --> Gekko`. For instance, if the GAMS code is `y(i, t) = sum(i, x(i, j, t))`, Gekko will paste the corresponding Gekko code `y[#i] = sum(#i, x[#i, #j])`.

Using a GAMS scalar model

The above-mentioned translation from GAMS to Gekko was straightforward enough, but GAMS statement or equation syntax may become complicated, and for performance reasons, and to ensure that the "true" GAMS equations are used, Gekko offers to use a so-called GAMS scalar model as an interface between GAMS and Gekko. See more about GAMS scalar models under MODEL.

Getting GAMS to produce a scalar model is in principle easy. If you have, for instance, the GAMS statement `solve mymodel using mcp;`, (solving the model 'mymodel' with the solver MCP), you may simply add this line just before the solve statement: `option mcp = convert;`. Then, instead of solving the model, GAMS will output two files `gams.gms` and `dict.txt`. These files contain the scalar model that Gekko can use for SIM<res>, DECOMP and other things (the files need to be zipped into for instance `mymodel.zip`, followed by a `MODEL<gms> mymodel.zip;` in Gekko). Inside `mymodel.zip`, you may also add a file `raw.gms`, containing the 'raw' GAMS equations).

The problem with .holdfixed = 1 and exogenous variables

If you are using the GAMS option `.holdfixed = 1`, exogenous variables will not show up in the scalar equations in the form of variables, but in the form of hard-coded numbers. Therefore, set `.holdfixed = 0` before producing the scalar model. This *may* render the model non-square (# equations and # variables does not match), and if so, GAMS will refuse to produce the scalar model. The following shows a way of mitigating that problem, using GAMS + embedded Python. The example uses the CNS solver and its particular "must be square" error message.

In GAMS first run squarefix1.gms, then run squarefix2.gms. This should produce the files `gams.gms` and `dict.txt`. Activate the other equations in squarefix_eqs.gms, and that should work, too. (Using maxExecError in this setup may not be strictly necessary).

squarefix_eqs.gms

```
* === Too many variables, too few equations ===
*  equation e1, e2, e3;
*  variable x1, x2, x3, x4;
*  e1 .. x1 +  x2 + x3 =e= 6;
*  e2 .. x1 + 2*x2 + x3 =e= 8;
*  e3 .. x3 =e= 2 + x4;
*  model m / e1, e2, e3 /;
* === Too many equations, too few variables ===
  equation e1, e2, e3;
  variable x1, x2, x3;
  e1 .. x1 +  x2 + x3 =e= 6;
  e2 .. x1 + 2*x2 + x3 =e= 8;
  e3 .. x3 =e= 2;
  model m / e1, e2, e3 /;
  x3.fx = 2;
```

squarefix1.gms

```
* The model is expected to have name m
$include squarefix_eqs.gms
* This is expected to fail
maxExecError = 100;
```

```
option cns = convert;
model temp / m /;
temp.holdfixed = 0;
solve temp using cns;
execerror = 0;
```

squarefix2.gms

```
* The model is expected to have name 'm',
* the .lst file the name 'squarefix1.lst',
$include squarefix_eqs.gms
* --- Gets rows minus columns = the number of redundant equations
scalar n_extra;
embeddedCode Python:
  import os
  filename = "squarefix1.lst"
  extra = 0
  b = False
  with open(filename, "r") as s:
    for line in s:
      if (b):
        m = line.split(" ")
        extra = int(m[3]) - int(m[7])
        break
      if (line.strip() == "**** ERROR: CNS models must be square"):
        b = True
  gams.set('n_extra', [extra])
  if os.path.exists("gams.gms"):
    os.remove("gams.gms")
  if os.path.exists("dict.txt"):
    os.remove("dict.txt")
endEmbeddedCode n_extra
* --- If n_extra is negative, there are too few equations.
* --- If n_extra is positive, there are too few variables.
* --- This is handled by either adding more (irrelevant) equations, or
* --- adding an equation with a sum of (irrelevant) variables.
set super_temp / 1*1000000 /;
set dyn_temp1(super_temp);
set dyn_temp2(super_temp);
dyn_temp1(super_temp) = ord(super_temp) <= max(0, 1 - round(n_extra));
dyn_temp2(super_temp) = ord(super_temp) <= max(0, round(n_extra) + 1);
variable x_temp1;
variable x_temp2[super_temp];
equation e_temp1[super_temp];
equation e_temp2;
e_temp1[super_temp] $ dyn_temp1(super_temp) .. 1 =e= x_temp1;
e_temp2 .. sum(super_temp $ dyn_temp2(super_temp), x_temp2[super_temp])
=e= 0;
model temp / m, e_temp1, e_temp2 /;
temp.holdfixed = 0;
option cns = convert;
solve temp using cns;
```

6.8 Gekko 2.4 to 3.0 cheat sheet

The following "cheat sheet" shows syntax differences between Gekko 2.4 and 3.0, focusing on some of the most oft-used parts of Gekko 2.4. Regarding 3.0 syntax, the short elevator pitch is a useful read, and besides there is the guide. Among other thinds, the syntax changes in 3.0 have to do with (a) how the symbols % and # are interpreted, (b) databanks now contain all kinds of variables not just series, and (c) lists now contain all kinds of variables not just strings (more on this here). Beware that "2.4" covers all versions 2.x.x, and "3.0" covers all versions 3.x.x. A two-page printing-ready pdf version of the cheat sheet is available here.

Gekko 2.4 to 3.0 cheat sheet

Series	Gekko 2.4	Gekko 3.0
definition	series x = 1, 2, m, 3;	x = 1, 2, m(), 3;
local period	series <2020 2021> y = x;	y <2020 2021> = x;
blanks separation	series x = 1 2 m 3;	x = data('1 2 m 3');
dynamic series	series x = x[-1] + 1;	x <dyn> = x[-1] + 1;
element access	series y[2021] = x[2020];	y[2021] = x[2020];
name-composition	series y{i}a = x{i}b;	y{%i}a = x{%i}b;
name-composition	series y%i|a = x%i|b;	y%i|a = x%i|b;
list of names	series #m = 100;	{#m} = 100;
operators	series y ^ 5; series y % 5;	y ^= 5; y %= 5;
operators	series y + 5; series y +$ 5;	y += 5; y <keep=p> += 5;
operators	series y * 5; series y # 5;	y *= 5; y #= 5;
frequency symbol	collapse x.a = x.q;	collapse x!a = x!q;

Lists	Gekko 2.4	Gekko 3.0
list of strings	list m1 = a, b, c;	#m1 = a, b, c;
singleton list	list m1 = a;	#m1 = a,;
'funny' strings	list <direct> m2 = 007, 7z;	#m2 = 007, 7z;
mix strings and lists	list m3 = x, #m1, y, #m2;	#m3 = x, {#m1}, y, {#m2};
listfile f.lst out	list listfile f = a, b, c;	#(listfile f) = a, b, c;
listfile f.lst in	list m = #(listfile f);	#m = #(listfile f);
union	list m3 = #m1 &+ #m2;	#m3 = #m1 || #m2;
intersection	list m3 = #m1 &* #m2;	#m3 = #m1 && #m2;
difference		#m3 = #m1 - #m2;
concatenate		#m3 = #m1 + #m2;
prefix (and suffix)		

remove element text
sort/trim
 	
element access
print list elements
print series
	list m3 = #m1 &- #m2;
	list m3 = #m1, #m2;
	
	list m = #m2 prefix = 'p';
	list m = #m2 strip = 'z';
	list m = #m2 sort trim;
	
	string s = #m3[2];
	list ? #m;
	prt #m;
		
		#m = #m2.prefix('p');
		#m = #m2 .replaceinside('z', '');
		#m = #m2.sort().unique();
		
		%s = #m3[2];
		prt #m;
		prt {#m};

Scalars	Gekko 2.4	Gekko 3.0

value definition
date definition
string definition
string concatenate
print string
print series
	val v = 1.23;
	date d = 2021q1;
	string s = 'xy5';
	string s2 = 'a' + %s + 'b';
	string ? %s;
	prt {%s};
		%v = 1.23;
		%d = 2021q1;
		%s = 'xy5';
		%s2 = 'a{%s}b'; //or like 2.4
		prt %s;
		prt {%s};

Wildcards	Gekko 2.4	Gekko 3.0

index with wildcard
index with range
search inside databank
search inside list
print matching series
	index x* m;
	index a..d m;
	list m2 = [x*], [a..d];
	list m2 = #m1[x*], #m1[a..d];
	prt [x*], [a..d];
		index x* to #m;
		index a..d to #m;
		#m2 = ['x*'] + ['a'..'d'];
		#m2 = #m1['x*'] + #m1['a'..'d'];
		prt {'x*'}, {'a'..'d'};

Loops	Gekko 2.4	Gekko 3.0

date loop
value loop
string loop
	for date d = %d1 to %d2; ...
	for val i = %i1 to %i2; ...
	for s = #m; ...
		for date %d = %d1 to %d2; ...
		for val %i = %i1 to %i2; ...
		for string %s = #m; ...

Matrix	Gekko 2.4	Gekko 3.0

matrix definition
element access
print matrix
print matrix
	matrix m = [1, 2 || 3, 4];
	matrix m[1, 2] = #m[2, 1];
	matrix ? #m;
	show #m * #m;
		#m = [1, 2; 3, 4];
		#m[1, 2] = #m[2, 1];
		prt #m;
		prt #m * #m;

Miscellaneous	Gekko 2.4	Gekko 3.0

import for all periods
export for all periods
use {} for path parts
use {} for sum
value to survive read
list to survive read
	import <csv> data;
	export <csv> data;
	read c:\gekko\%path\data;
	series y = sum(#m);
	val v = 1.23;
	list m1 = a, b, c;
		import <csv all> data;
		export <csv all> data;
		read c:\gekko\{%path}\data;
		y = sum({#m});
		global:%v = 1.23;
		global:#m1 = a, b, c;

Model/sim example	Gekko 2.4	Gekko 3.0

The model/sim syntax is very similar. In 3.0, lists and scalars that must survive read statements must be placed in the Global databank. Series operators include a "=" symbol. And you must use more {}-curlies when referring to scalars and lists.	reset;
	mode sim;
	list fx = fxnx, fxqx;
	string path = 'c:\models';
	model %path\model2;
	read bank2;
	time 2021 2025;
	sim;
	clone;
	series tg + 0.01;
	sim;
	mulprt #fx;
	read bank2;
	endo tg;
	exo fy;
	sim;
	mulprt tg, fy, #fx;
		reset;
		mode sim;
		global:#fx = fxnx, fxqx;
		global:%path = 'c:\models';
		model {%path}\model2;
		read bank2;
		time 2021 2025;
		sim;
		clone;
		tg += 0.01;
		sim;
		mulprt {#fx};
		read bank2;
		endo tg;
		exo fy;
		sim;
		mulprt tg, fy, {#fx};

About {}-curlies in Gekko 3.0

In Gekko 3.0, it is more often required to enclose a string `%s` or a list of strings `#m` in curlies, that is, `{%s}` or `{#m}`. This applies to the following statements: `assignments`, `analyze`, `checkoff`, `collapse`, `compare`, `copy`, `disp`, `doc`, `endo`, `exo`, `export`, `findmissingdata`, `import`, `interpolate`, `itershow`, `ols`, `prt/mulprt/plot/sheet`, `read`, `rebase`, `rename`, `smooth`, `splice`, `truncate`, `write`, `x12a`, where `assignments` refer to series/scalar/list etc. statements. Consider this difference:

```
delete {%s}, {#m};
delete %s, #m;
```

If the string `%s` contains `'x1'` and the list `#m` contains the strings `'x2'`, `'x3'`, `'x4'`, the first statement deletes the series `x1`, `x2`, `x3`, `x4` (4 in all). In the next statement, it is the string and list objects `%s` and `#m` themselves that are deleted (2 in all). Understanding this conceptual difference is important!

6.9 Gekko 2.4 to 3.0 reasons

Migrating from Gekko 2.4 to 3.0 can be frustrating syntax-wise, because both versions have their syntax peculiarities. Existing Gekko 2.4 users therefore have to unlearn some peculiarities, and learn other peculiarities, in order to migrate to Gekko 3.0.

There are the following points to note about Gekko 2.4 vs. 3.0:

1. In Gekko 3.0, the symbols `%` and `#` in variable names (for scalars and collections) are interpreted is if they were just normal characters alongside the normal `a` ... `z` alphabet, and the "characters" `%` and `#` are interpreted as an integrated part of the variable name. In Gekko 2.4, the `%` and `#` parts of variable names worked more like a kind of operators.
2. Databanks in Gekko 3.0 can contain any kinds of variables, not just timeseries.
3. Lists in Gekko 3.0 can contain any kinds of variables, not just strings.

The flexibilities of (2) and (3) imply a lot of advantages. Databanks can store all kinds of stuff, and lists can represent a lot of different things. For instance, you can import spreadsheet cells into a nested Gekko list (list of lists), where each cell is represented as a Gekko value, date, or string. Lists in Gekko 2.4 can only contain simple strings.

There are some drawbacks to the flexibility, too. For instance, more `{}`-curlies are needed in Gekko 3.0 than in 2.4, but this has to do with point (2) and (3). To understand the use of `{}`-curlies more, let us envision a completely different Python-like language:

```
//"Python"
a = 1
b = ('x', 'y')
c = 'y';
x = 2
y = 3
print a, b, c
```

In this language, variable names cannot contain the special symbols `%` or `#`, and the code above defines three numerical values (variables `a`, `x`, `y`), a list of strings (variable `b`), and a string (`c`). When printing `a, b, c`, you would expect `1`, `('x', 'y')`, and `'y'` to show up on the screen (a value, a list of strings, and a string).

But what if you intended `print a, b, c` to mean "print `a`", then "print the variables referred to by `b`", and finally "print the variable referred to by `c`"? In that case, "the variables in `b`" should be understood as the variables `x` and `y`, and "the variable in `c`" should be understood as the variable `y`. And if *that* was the intention, Gekko should print out the four values `1`, `2`, `3` and `3` (corresponding to `a`, `x`, `y` and `y`). But how would Gekko know that you intended this, and not the printing of raw strings?

To get `1`, `2`, `3`, `3` printed, Gekko operates with `{}`-curlies, which in this case would look like the following:

```
//"Python"
print a, {b}, {c}
```

This would print `1`, `2`, `3`, `3`, because the `{}`-curlies tell the program that the inside of the curlies is to be understood as a *reference* to other variables. Now, let us return to Gekko syntax. Consider this statement in Gekko:

```
//Gekko
print a, #b, %c;
```

Here, `a` is a timeseries, `#b` is a list of strings containing `('x', 'y')`, and `%c` is a string `'y'`. Here, Gekko 3.0 will print out the data of the series `a`, followed by the list `('x', 'y')`, followed by the string `'y'`. If you intend to print out the *variables* that the list `#b` and the string `%c` refer to, you must use `{}`-curlies:

```
//Gekko
print a, {#b}, {%c};
```

In Gekko 3.0, this prints out the data of the timeseries `a`, `x`, `y`, `y`. A second example is copying. In Gekko 3.0, we can do the following:

```
//Gekko 3.0
copy #b to #b2;
```

This means make a copy of the list `#b`, and call this copy `#b2` (if `#b` contains the two strings `'x'` and `'y'`, so will the `#b2` list). The following does something very different:

```
//Gekko 3.0
copy {#b} to {#b2};
```

Here, Gekko looks for the two existing lists `#b` and `#b2` and uses the string names inside them as variable references. If `#b` contains the strings `('x', 'y')`, and `#b2` contains the strings `('z', 'w')`, the statement amounts to copying the series `x` into `z`, and the series `y` into `w`. Again, the user might think that when using `copy #b to #b2;` it is obvious that it is the two *series* `x` and `y` that are to be copied into something else, but when databanks may contain lists, how can Gekko know that this is the intention, without the user indicating it somehow?

This is different in Gekko 2.4, because in 2.4, databanks cannot contain anything besides timeseries, and therefore, in Gekko 2.4, the statement `copy #b to #b2;` actually copies the two timeseries `x` and `y`, into `z` and `w`, because Gekko "knows" it does not make sense to talk about copying a list in a databank. Is the 2.4 syntax easier to write in this case? Absolutely, but the databanks are also much less flexible.

A third example regarding differences is list construction. Consider this example in Gekko 3.0:

```
//Gekko 3.0
#b = x, y;
#b2 = a, {#b}, c;
```

Here, `#b = x, y;` is a so-called naked list definition, avoiding the more tedious `#b = ('x', 'y');`. After the second statement is executed, the list `#b2` will contain the four strings `('a', 'x', 'y', 'c')`, because the `{}`-curlies look inside `#b` and fetches its contents. But then what about this?:

```
//Gekko 3.0
#b = x, y;
#b2 = a, #b, c; //fails
```

In Gekko 2.4, `#b2 = a, {#b}, c;` creates the list `#b2` as the four strings `('a', 'x', 'y', 'c')`, whereas `#b2 = a, #b, c;` fails. This is intentional, to avoid too much confusion (in principle it could produce the list of three strings `('a', '#b', 'c')`, but the confusion would outweigh the advantages).

As the fourth and final example, consider this syntax in Gekko 3.0:

```
//Gekko 3.0
#b = x, y;
{#b} = 100;
```

This sets the two series `x` and `y` to the constant value 100. In Gekko 2.4, you can omit the curlies:

```
//Gekko 2.4
list b = x, y;
series #b = 100;
```

Again, this syntax may seem more simple, because the `{}`-curlies can be omitted. We may try to do something similar in Gekko 3.0:

```
//Gekko 3.0
#b = x, y;
#b = 100;
series #b = 100;
```

In the second statement, `#b` on the left-hand side is just interpreted as a variable name consisting of two characters from the special "Gekko alphabet". So Gekko first calculates the right-hand side (= scalar value 100) and next tries to assign the scalar to the variable name `#b`. This fails with a type error, since variables starting with `#` in Gekko can only be a list, a matrix, or a map. The last statement fails, too, but from a different reason. Here, it is stated explicitly that the variable `#b` must be of series type, but Gekko refuses to construct a series name that starts with `#` (which is not legal).

Now, why does Gekko 3.0 not just "know" that the intention of `#b = 100;` or `series #b = 100;` is to define the two timeseries variables corresponding to the two strings inside `#b`? In principle, this could be done, but unfortunate side-effects would creep in. When a variable name like `#b` can also be for instance a matrix, couldn't the intention of `#b = 100;` rather be a simplified way to define a 1x1 matrix? Also, when using the type indicator, `series #b = 100;` could make more sense, but then we have the problem of `series #b = 100;` working, but `#b = 100;` not working, which seems inconsistent.

Omitting the `{}`-curlies around lists names on the left-hand side opens up other ambiguities, too. In Gekko 3.0, you can for instance do the following:

```
//Gekko 3.0
time 2001 2003;
#m1 = a, b, c;
#m2 = 1, 2, 3;
{#m1} = #m2;
print <n> a, b, c;
//                   a              b              c
// 2001         1.0000         1.0000         1.0000
// 2002         2.0000         2.0000         2.0000
// 2003         3.0000         3.0000         3.0000
```

Here, `#m1` is a list of three strings, and `#m2` is a list of three values. The convenient Gekko 3.0 statement `{#m1} = #m2;` assigns the three values to each of the three series `a`, `b`, and `c`. This is practical in many cases, but what if we were instead using the assignment `#m1 = #m2;`. Shouldn't Gekko just *know* that `#m1 = #m2;` intends to update three timeseries? But how should it know? The statement `#m1 = #m2;` looks much more like assigning the elements of the `#m2` list into the list `#m1`, so that `#m1` = `(1, 2, 3)` afterwards.

But then what about a statement like `series #m1 = #m2;` in Gekko 3.0? This *could* be rewired to produce three timeseries, instead of failing. But again, there would be side-effects. What if someone removes the `series` type indicator, because `series` type indicators are in general superfluous in Gekko 3.0? In that case, the statement still runs, but does something altogether different (namely defining a list `#m1`). This is bound to create confusion, and perhaps errors.

Note that simplified names like for instance `x%i%j` still work in Gekko 3.0, where `%i` and `%j` are two strings that are in-substituted. In that case, you do not have to use the more formal `x{%i}{%j}`. This former way of succinct writing is no longer officially endorsed in Gekko 3.0, but it is still legal. One of the reasons why it is no longer endorsed is that the string `'x{%i}{%j}'` will perform the same in-substitution of `%i` and `%j` (so-called string interpolation), whereas `'x%i%j'` will not.

Wildcards

Another place where Gekko 3.0 differs from Gekko 2.4 syntax-wise is wildcards. Gekko 2.4 allows a statement like `index a*b m;`, which will match timeseries in the databank starting with `'a'` and ending with `'b'`, and create a list `#m` to store the results. Incidentally, this is a good case of a `#` character ambiguity in Gekko 2.4: should the `index` statement end with `m` or `#m`? It ends with `m` because this corresponds better with the statement `list m = [a*b];` where the name is without `#`-symbol, because it is on the left-hand side. That the legal statement is not `index a*b #m;` in Gekko 2.4 is basically a quite arbitrary choice.

In places where the `*` can mean multiplication, Gekko 2.4 demands the use of `[]`-brackets, for instance `print [a*b];`. If omitted, Gekko 2.4 will just print the product of the two timeseries `a` and `b`.

Gekko 3.0 takes a different approach to wildcards, demanding that these are in principle quoted inside `[]`-brackets, for instance `print ['a*b'];`. Adding the quotes is a bit tedious, but omitting them like in Gekko 2.4. creates all sorts of syntax ambiguities for the parser, and also makes it next to impossible to *compose* a wildcard from other strings (one problem about a wildcard like `[a*b]` is that it also parser-wise looks like a 1x1 matrix definition).

The expression `['a*b']` returns a list of strings corresponding to the matching timeseries, so `print ['a*b']` prints strings, not timeseries. To print the data of the matching timeseries, you should use `print {'a*b'};`, using curlies instead. So the Gekko 3.0 statement `print {'a*b'};` corresponds to the Gekko 2.4 statement `print [a*b];`. This costs a couple of single quotes, but many statements in Gekko 3.0 allow simplified wildcards (which is also the case for Gekko 2.4). So you can write for instance the simple `index a*b to #m;` or `copy a*b to x*;` in Gekko 3.0, which is very similar syntax-wise to Gekko 2.4.

There are some other differences between wildcard search in Gekko 2.4 and Gekko 3.0. But these are more semantic and are caused by the fact that Gekko 3.0 databanks can contain variables that contain `%` and `#` symbols (and `!` for frequency for that matter).

Miscellaneous annoyances

- In Gekko 2.4, you can write for instance `exo #b;` to exogenize the two variables (series) `x` and `y`, presupposing that `#b = ('x', 'y')`. In Gekko 3.0, you have to write `exo {#b};` to indicate that you are referring to the *elements* of `#b`. Since you cannot use a raw list of strings directly for exogenization, something like `exo #b;` *could* be made legal in Gekko 3.0, too. But even though this saves some typing of `{}`-curlies, it also affects the consistency of the language. The users may ask themselves why they should write `print {#b};` to print the variables that the strings refer to, whereas `exo #b;` is enough?

- Gekko 3.0 demands a `=` symbol in any assignment statement. Therefore, it is no longer possible to write something like the Gekko 2.4 syntax `series x % 2;` to make the series `x` grow with 2% per period. In Gekko 3.0, this instead reads `x %= 2;`, which is also more in line with modern programming syntax. Omitting the `=` symbol would be hard syntax-wise in Gekko, because it helps identifying assignment statements, also for the human reader (remember that in Gekko 3.0, it is no longer mandatory to decorate assignment statements with type indicators like `series`, `val`, `date`, etc).
- In Gekko 3.0 series definitions, numerical values must in general be separated by commas (blanks not allowed). However, you may use the `data()` function to handle blank-separated data, for instance `x = data('1 2 3');` instead of `x = 1, 2, 3;`.
- In Gekko 3.0, dynamic statements like `x = x[-1] + 1;` have to be decorated with `<dyn>` tag, for instance `x <dyn> = x[-1] + 1;`. The reason for this is long-winded, but newer Gekko 3.1.x versions will abort with an error if the tag is unintentionally forgotten. And some users may even appreciate that dynamic statements are clearly visible via the tag.
- As mentioned above, in Gekko 3.0, you cannot use for instance `#b = 1;` and expect Gekko to understand that this means that you want to set the *elements* of `#b` (the series `x` and `y`) to the value 1 (instead you must use `{#b} = 1;`).
- In Gekko 3.0, you have to use for instance `x{%i}{%j}` for a composed name using the strings `%i` and `%j`, whereas you can no longer write it as `x{i}{j}`. The reason for this (and for other syntax changes) is explained at the bottom of this page. On the positive side, variable names like `x{#i}{#j}` can be used in Gekko 3.0, where `#i` and `#j` are lists of strings (Gekko will automatically combine and unfold the two lists into composed variable names).
- The READ statement and the Global databank. In Gekko 3.0 (as was the case for Gekko 2.4), the READ statement first clears the first-position databank (typically Work), before reading in a databank. READ is often used when modeling, but since scalars and collections now live in databanks, this means that a READ statement also wipes out all "settings"-variables that the user may have defined at the top of his/her Gekko program file. This could be file paths, start/end dates, scale factors, lists containing names of sectors, etc. It seems arbitrary to require that READ only clears timeseries variables in the first-position databank, so instead, regarding "settings"-variables, these can profitably be put into the so-called Global databank. This databank is always open and always searched last, and survives READ, CLEAR, etc. So at the top of your program file, you just define for instance `global:%path = 'c:\forecasts\bank1';`, or `global:#sectors = a, nz, qz, o;`, and you can subsequently refer to `%path` and `#sectors` without worrying about them being wiped out by READ or CLEAR.

Preprocessor for simplifyed Gekko 3.0 syntax?

There is no denying that Gekko 3.0 in some statements introduces some `{}`-symbols and other "extras" like single quotes, `%`-symbols and `=`-symbols (or the occasional `global:`) compared to Gekko 2.4.

As explained above, this has much to do with a desire to treat the %- and #- symbols in a more consistent and less arbitrary way than in Gekko 2.4, combined with the fact that Gekko databanks and Gekko lists now allow all kinds of variables inside, including %-variables (scalars) and #-variables (collections). On the contrary, Gekko 3.0 also allows quite a bit of "noise" to be removed, since type indicators like `series`, `val`, `date`, etc. are no longer mandatory in assignment statements.

Could Gekko 3.0 support a simplified syntax like for instance the following, where it is implicitly presupposed that all use of #-lists is to be interpreted as operating on the *elements* of the lists (on the timeseries that these elements refer to)?

```
//Simplified 3.0 syntax
#b = x, y;             //('x', 'y')
#b2 = a, #b, c;        //('a', 'x', 'y', 'c')
copy #b to bank3:#b;   //copy two series
#b % 2;                //growth of two series set to 2%
exo #b2;               //exogenize
print #b;              //print two series
print [a*b];           //wildcard to print series
for i = x1, x2;        //print x1
  print %i;            //and x2 via
end;                   //a loop
```

This is how it would look in Gekko 3.0:

```
//Gekko 3.0
#b = x, y;
#b2 = a, {#b}, c;
copy {#b} to bank3:{#b};
{#b} %= 2;
exo {#b2};
print {#b};
print {'a*b'};
for string %i = x1, x2;
  print {%i};
end;
```

Making the simplified syntax work directly in Gekko 3.0 would take some work, and it would complicate both the parser and the code that transforms the parser output into suitable C# code that is executed later on.

Since the above is to a large degree a question of {}-curlies, perhaps a kind of preprocessor could be developed to handle such uses (for instance when doing modeling), where lists always are expected to contain strings (of variables names), and where scalar strings mostly represent variable names, too. Perhaps an option like the following could be possible:

```
option syntax = simplified;
```

When Gekko encounteres this option, it performs a pure syntactical transformation of the subsequent statements, where all the necessary `{}`-curlies (and possibly a few `=` etc.) are simply added to the statements before they are sent to Gekko for parsing, compilation and execution.

This translation would not be particularly difficult to do, using a so-called tokenizer already built into Gekko and used for similar tasks like translating from AREMOS, GAMS, etc. The advantage of this preprocessor/translator approach is that the Gekko parser and compiler only "sees" strict Gekko 3.0 code, and the transformations that the preprocessor performs would be well-defined and well-described.

Another way of doing this would be to optionally transform the syntax when hitting [Enter] on a statement in the Gekko main window. Gekko already inserts an ending `;` is this is missing, so when in "simplified" mode and the user hits [Enter] on a statement like `print #b;`, it will be transformed into `print {#b};`. This transformation will actually happen in the lower part of the Gekko main window, just like any missing `;` are actually inserted "physically" into the text editor part of Gekko.

Perhaps the users can live with the `{}`-curlies (and are aware of when to use them), but just find them annoying to write. If so, a function key like for instance F4 could be wired to put `{}`-curlies around the variable that the cursor is currently resting at in the text editor part of the main Gekko window (lower part). Again, this would happen "physically" in the text editor, so hitting F4 when the cursor is at the variable `#b` would transform it into `{#b}`. Hitting F4 again would remove the curlies. This is a bit similar to what Excel does with the F4 function key when editing an Excel formula (if the cursor is on for instance `A1` in a formula, the F4 button circles through the variants `$A$1`, `A$1`, `$A1`, and `A1`).

6.10 Gekko solvers

This is a short section on some of the solvers in Gekko. At the moment, only the following is detailed here:

- Newton-Fair-Taylor

6.10.1 Newton-Fair-Taylor

The Fair-Taylor (FT) algorithm can be perceived as performing Gauss-Seidel over time, instead of over equations. If the model is simulated over the period t1 to t2, this simulation is repeated again and again (where the values of any leaded variable are simply taken from the databank), until the leaded variable(s) converge (that is, do not move from iteration to iteration). The particular way each of the simulations from t1 to t2 are performed is irrelevant here, as long as the simulations solve the model.

Consider this one-equation model:

```
  y = 0.1*y[-1] + 0.2*y + 0.3*y[+1] + 100
```

If the model is simulated from 2001 to 2004, the first simulation (2001) will take y[+1] as the 2002 value, the second simulation (2002) will take y[+1] as the 2003 value, the third simulation (2003) will take y[+1] as the 2004 value, and the last simulation (2004) will take y[+1] as the 2005 value. Regarding the 2005 value, there is the question of terminal conditions, but we might imagine that option 'exo' is used regarding terminals, so that the real (databank) value regarding y[2005] is used. We might consider this the first FT-iteration, that produced new values regarding y[2001], y[2002], y[2003] and y[2004].

In the next FT-iteration, consider the first sub-simulation, that is, the simulation of the year 2001. In this simulation, y[+1] is taken as the y[2002] value that was computed in the first FT-iteration. And likewise, regarding the simulation of the year 2002, y[+1] is taken as the y[2003] value that was computed in the first FT-iteration, and so on. The last simulation (2004) will still use the fixed value of y[2005], because we use the 'exo' option regarding terminals.

This process is repeated over and over, and in many cases it converges nicely. It can also be damped, and in spirit, the method is very similar to the Gauss-Seidel algorithm that solves the individual periods (hence, Fair-Taylor is also called Gauss-Seidel over time). Still, the process is by no means guaranteed to converge, and in cases with heavy intertemporal influences (for instance, if the parameter regarding y[+1] were 0.9 instead of 0.3), the 'signals' from the leaded variables from FT-iteration to FT-iteration may be slow to propagate from the last periods to first periods. If the intertemporal influence is too heavy, for instance with 0.9*y[+1] instead of 0.3*y[+1], the Fair-Taylor algorithm cannot solve the problem (no matter how much damping is used etc.).

In such cases, a more robust Newton solver is often used. The most common way to progress is typically to 'stack' the equations, eliminating all lags and leads. For instance, the system regarding 2001-2004 could be written in the following way:

```
y1 = 0.1*y0 + 0.2*y1 + 0.3*y2 + 100
y2 = 0.1*y1 + 0.2*y2 + 0.3*y3 + 100
y3 = 0.1*y2 + 0.2*y3 + 0.3*y4 + 100
y4 = 0.1*y3 + 0.2*y4 + 0.3*y5 + 100
```

This system is time-less (or static) in the sense that there are no lags or leads, but only variables with names that indicate the period. Provided that some values regarding y0 (= y[2000]) and y5 (= y[2005]) are provided, the system can be solved with a 'normal' solver (that is, a solver that solves one period at a time).

Now, such a system can become quite large, because the equations are unfolded (stacked) like in the example above. For a 100 year simulation period, the model becomes 100 times larger than a normal model, and the Jacobi matrix obtains 100*100 = 10.000 times more elements. There are a lot of tricks and methods to alleviate this explosion of variables, but implementing such tricks (for instance sparse matrices) is a time-consuming process.

Gekko implements such a stacked time solver, but it is mostly suited for smaller models, or for limited time periods. For larger models solved over long time periods, an alternative solver can be used. Provided that there are not too many variables with leads, this solver is quite powerful. The solver is called Newton-Fair-Taylor, and the basic principles of this solver is presented in the following section.

Newton-Fair-Taylor

This solver is activated by means of the following option:

```
option solve forward method = nfair;   //default is 'fair'
```

The underlying idea is to view Fair-Taylor as an iterated process, where the goal is to find a fixed point in this process. If we limit the problem to one variable (y) containing leads, the process can be stated as follows:

$$y_{new} = F(y_{old})$$

So y_{old} is a nx1 vector of initial values for y regarding the simulation period (with n observations), and this is fed to the normal solution algorithm, where leaded variables are just used at face value. The normal solution algorithm simulates the n periods, which produces n new values for y. The process is repeated over and over, and as soon as $y_{new} = y_{old}$, the problem is solved.

This system can be linearized around some particular vector y_{old}, by performing a small perturbation ε on one of the elements of y_{old}, simulating the model over n periods, and observing the effects on y_{new}, compared to a simulation without ε. These perturbations are performed for each period, and produce n effects (for each of the periods contained in y_{new}). All in all, we obtain a n x n matrix of effects. For example, if the model y = 0.1*y[-1] + 0.2*y + 0.3*y[+1] is simulated over 4 periods, we obtain the following matrix:

```
                     1              2              3              4

     1          0.0000         0.0000         0.0000         0.0000
     2          0.3750         0.0469         0.0059         0.0007
     3          0.0000         0.3750         0.0469         0.0059
     4          0.0000         0.0000         0.3750         0.0469
```

The first row shows what happens to y[2001], y[2002], y[2003] and y[2004], if y[2001] is changed by one unit. In 2001, here is no effect of changing y[2001], since this only affects the starting values. As there are no effects in 2001, there are no effects in 2002-2004 either, so the first row is full of zeroes. The second row shows what happens, if y[2002] is changed by one unit. In 2001 this affects y, via the leaded variable. The effect is 0.3/(1-0.2) = 0.375. This in turn affects y in 2002 via the lag 0.1*y[-1]. This effect is 0.375*0.1/(1-0.2) = 0.0469, and so on.

This matrix is used to transform the error (y_{new}-y_{old}) into a new guess regarding y. In order to do this, the 4x4 identity matrix is subtracted, and the resulting matrix is inverted. This matrix can then be combined with the error, to produce a new guess.

With option 'const' (option solve forward terminal = const) regarding terminal values, the idea is that regarding the terminal value y[2005], this value is set equal to y[2004]. Hence, the terminal value is assumed constant ('const') in relation to the y[2004] value, and in a stacked system, this would amount to the following:

y1 = 0.1*y0 + 0.2*y1 + 0.3*y2 + 100
y2 = 0.1*y1 + 0.2*y2 + 0.3*y3 + 100
y3 = 0.1*y2 + 0.2*y3 + 0.3*y4 + 100
y4 = 0.1*y3 + 0.2*y4 + 0.3*y4 + 100

Note the last equation, where y4 is used instead of y5. Using the 'const' option changes the incidence matrix, which would now be the following:

```
                    1              2              3              4
     1          0.0000         0.0000         0.0000         0.0000
     2          0.3750         0.0469         0.0059         0.0012
     3          0.0000         0.3750         0.0469         0.0094
     4          0.0000         0.0000         0.3750         0.0750
```

This changes the last column, for instance the element (4, 4) is changed from 0.0469 to 0.0750, that is a factor (1-0.2)/(1-0.2-0.3) = 1.6. With "OPTION solve forward terminal feed = internal", the incidence matrix will look like the matrix above, whereas with this option set to 'feed = external', the incidence matrix will look like the first one shown (with element (4, 4) = 0.0469).

With 'feed = internal', a linear model with leads will solve in one Fair-Taylor iteration, regardless of the coefficients (provided that the model is solvable). Using 'feed = external', more Fair-Taylor iterations are needed.

It would be tempting to provide a 'growth' option regarding terminal values, corresponding to this stacked system:

$$\begin{aligned}
y1 &= 0.1*y0 + 0.2*y1 + 0.3*y2 + 100 \\
y2 &= 0.1*y1 + 0.2*y2 + 0.3*y3 + 100 \\
y3 &= 0.1*y2 + 0.2*y3 + 0.3*y4 + 100 \\
y4 &= 0.1*y3 + 0.2*y4 + 0.3*y4*y4/y3 + 100
\end{aligned}$$

The last equation corresponds to y5/y4 = y4/y3, and the problem is that this equation may be hard to solve when solved on its own (because the right-hand side contains a term with y4 squared). Because of this, a 'growth' option regarding terminal values is not provided at the moment.

All in all, the Newton-Fair-Taylor solver ('nfair') is quite powerful regarding forward-looking models. If the number of variables with leads is limited, the incidence matrix does not become too large, and filling it up with coefficients does not become too time consuming. With k lead variables, the incidence matrix becomes (k*n) x (k*n) instead of n x n, and the matrix will contain cross-effects from one lead-variable to another. Such a matrix may be time-consuming to compute, since in principle k*n simulations are needed to fill it (not all simulations need to solve the full period, but there is still a lot of work to be done).

Looking at the original incidence matrix, an obvious idea springs to mind:

```
                 1             2             3             4

  1         0.0000        0.0000        0.0000        0.0000
  2         0.3750        0.0469        0.0059        0.0007
  3         0.0000        0.3750        0.0469        0.0059
  4         0.0000        0.0000        0.3750        0.0469
```

It is clear that the matrix is repetitive, with the same number (cf. the colors) running diagonally downwards. Using this idea, and assuming that the coefficients do not change too much over time in non-linear models, only k simulations are needed to fill the matrix (k being the number of leaded variables).

Some care must be taken regarding the coefficients corresponding to terminal values (if 'const' terminal condition is used), but a quite good approximation of the real matrix can probably be produced with little effort using this idea.

Newton-Fair-Taylor example

Provided the following model file:

```
------------- y.frm -----------------------------------

frml _i y = 0.1*y[-1] + 0.2*y + 0.3*y[+1] + 100;

-------------------------------------------------------
```

You may try the following statements:

```
RESET;
OPTION solve forward method = nfair;
OPTION solve forward dump = yes;
TIME 2001 2004;
MODEL y;
CREATE y;
SERIES <2000 2000> y = 200;
SIM;
PRT y;
PRT #ft_1;
```

Option 'nfair' is set regarding forward solving (default regarding terminals is always 'const'). Option 'dump' is set, so that the gradient matrix can be shown afterwards (#ft_1). The result is the following:

```
 +++ NOTE: There are 1 variable(s) with leads: Fair-Taylor algorithm is used
 #1: Gauss simulation 2001-2004 took 0.00 sec -- 10/14/8.8 iterations (min/max/avg)
     Gradient 1 of 4 (var 1 per 2001)
     Gradient 2 of 4 (var 1 per 2002)
     Gradient 3 of 4 (var 1 per 2003)
     Gradient 4 of 4 (var 1 per 2004)
 #2: Gauss simulation 2001-2004 took 0.01 sec -- 10/14/43.4 iterations
(min/max/avg)
 Newton-Fair-Taylor (leads) algorithm converged in 3 NFT-iterations (1.34 sec)

                    y          %
  2001       243.4254      21.71
  2002       249.1343       2.35
  2003       249.8830       0.30
  2004       249.9766       0.04

 #ft_1
                      1                2                3                4
     1           0.0000           0.0000           0.0000           0.0000
     2           0.3750           0.0469           0.0059           0.0012
     3           0.0000           0.3750           0.0469           0.0094
     4           0.0000           0.0000           0.3750           0.0750
```

The gradient matrix uses four simulations (one for each period), and the algorithm converges in 3 Newton-Fair-Taylor iterations, which is always the case for a linear model. Note that in this simulation, the actual databank value of y in 2005 is not used at any point (it is missing value anyway). From the results, it is seen that y is developing towards it's equilibrium value 250. This value can be found by removing all lags/leads in the equation y = 0.1*y[-1] + 0.2*y + 0.3*y[+1] + 100, so that it becomes y = 0.1*y + 0.2*y + 0.3*y + 100 (which has y = 250 as solution).

Setting "OPTION solve forward terminal = exo", and setting y[2005] = 200 would change the solution completely:

```
  2001       242.2783      21.14
  2002       246.0754       1.57
  2003       242.1082      -1.61
  2004       230.2635      -4.89
```

In this case, the y[2004] is 'attracted' to y[2005] = 200, and hence cannot attain a value close to 250 anymore. So the choice of terminal condition should not be taken too lightly, and the 'const' option makes much more sense than the 'exo' option in most cases.

6.11 Comparison with similar software

In this section, Gekko is compared to other similar software packages, so that potential user can more easily judge whether Gekko suits their needs. The comparison is done regarding the older Gekko 2.0 version (Gekko 3.0 has a lot more capabilities).

- AREMOS (version 5.4.1)
- EViews (version 8)

6.11.1 Compare with AREMOS

This section compares AREMOS (version 5.4.1) modules and functionality with Gekko 2.0 (as of December 2015). The comparison is focused on overall functionality, and not so much on particular details. So this is not a command-by-command comparison.

The following two comparisons are provided:

- Checking which AREMOS components are available/missing in Gekko
- Describing Gekko components not available in AREMOS

Conclusion: Gekko implements almost all relevant features regarding datarevision projects. Regarding modeling and solving models, Gekko is superior. And regarding econometrics, Gekko lacks a lot of these features, but these features are not the main point of the software (for the time being at least).

Note in general that econometrics is skipped in the comparison. AREMOS contains quite a lot of econometrics functionality, but it is not the intention to emulate this part of AREMOS in Gekko (but instead provide good interfaces to econometrics packages like R etc.).

AREMOS component	Status in Gekko 2.0	Comment
Program structure	ok	Like AREMOS, Gekko is timeseries and databank oriented, allowing any number of open databanks containing timeseries with any frequency. Since OPEN<prim>, local time settings like <2010 2020>, etc. are implemented similarly, the overall structure of a Gekko program will typically be very similar to the structure of a corresponding AREMOS program.
Syntax	ok	Related to program structure, the basic Gekko syntax (abstracting from concrete command names) is also basically similar to AREMOS syntax, albeit with some differences. • Lists of items are separated by commas (','), for instance "PRINT x1, x2 x3;" • All lines end with semicolon (';') • Indexing uses square brackets ('[' and ']'), including lags. For instance gdp[-1], gdp[2020], #m[2, 3]. Ranges are '..' instead of '-' in Gekko: #m[2..3, 1..5]. • Lists use '#' prefix, for instance #m. Contrary to AREMOS, Gekko matrices also use '#' prefix to distinguish them from timeseries.

		• Scalars STRING, NAME, DATE and VAL use '%' as prefix, to distinguish them from lists. These scalars correspond to the equivalent ASSIGN variables in AREMOS. • Timeseries are without prefix as in AREMOS. Name/string scalars can be used for name composition, including the use of the '|' concatenator. • Banks are referred to by colon (':'), as in AREMOS. • Local options are set inside a '<' and '>' braces, for instance "PRT <2010 2020> gdp;". This complies with AREMOS. • Loops (FOR) and conditionals (IF) work more or less in the same way as AREMOS. REPEAT and WHILE will be added. • Comments use '//' instead of '!' • Strings use single quotes (') only, AREMOS accepts double quotes too (") • Dot (.) can be used for frequencies, to distinguish gdp.q from gdp.a. Dots will probably be possible in variable names, but are not allowed at the moment. Dots can also be used for lags, for instance x.2 = x[-2]. • Mathematical and logical operators work like in AREMOS. Many of the same in-built functions are provided.
Databanks	ok	Gekko provides the same functionality, with a list of open databanks, searchable for timeseries. The concept of the first-position databank is extended with a 'reference' databank concept. This makes many kinds of comparisons much easier. Databanks are closed with CLOSE and written to automatically if changed, and protected banks are supported. Banks are referred to with single colon, and all AREMOS functionality regarding timeseries copying, clearing, indexing, renaming, counting etc. is available. Gekko does not have a support bank, but it is intended to put procedures/functions into a general 'support' folder instead. Gekko includes READ/WRITE for dealing with databanks. This is often practical. IMPORT/EXPORT is also supported. At the moment, Gekko databanks contain series only (annual, quarterly, monthly or undated). It is considered to allow scalars, matrices and lists into databanks, whereas procedures/functions and equations/models are kept outside.

		Advanced structures and documents are not supported, but there is a DOC statement to handle label and source for timeseries. Contrary to AREMOS, searching for a name in a Gekko databank always takes the same amount of time, regardless of the databank size (Gekko uses hash tables internally).
Speed	ok	Gekko runs models and program files much quicker than AREMOS. For program files, AREMOS uses interpretation on a line-by-line basis, whereas Gekko compiles the statements into machine code.
Timeseries and frequencies	partial	Gekko supports annual, quarterly, monthly or undated frequencies, with the same codes (A, Q, M, U). Higher frequencies could be added if necessary. Gekko does not use colon for dates, for instance 2015:3. Instead the more informative 2015q3 notation is used (which AREMOS also uses). Like AREMOS, the time period and frequency can be set globally, and is used implicitly in many commands. Gekko timeseries can have any number of observations, and Gekko uses missings to avoid the dreaded "** Observation outside current period" message in AREMOS. COLLAPSE, SMOOTH, SPLICE, TRUNCATE and ANALYZE are supported. Seasonal adjustment (X12A) is provided. Transformation series are not supported.
Lists	ok	Gekko supports almost precisely the same components for list manipulation, including listfiles.
Wildcards	ok	Gekko also uses '*' and '?' for wildcards, but uses '..' for ranges instead of '-'.
Matrices	ok	Basically all of AREMOS' matrix functions are provided, including indexing of sub-matrices etc. The syntax is a little bit different, since Gekko refers to matrices with '#' prefix, and matrix construction uses the [1, 2 || 3, 4] pattern rather than [1, 2]||[3, 4]. In addition, ranges are '..', not '-'. Pack/unpack regarding timeseries is provided. A matrix editor is not provided, but copy/pasting a matrix to Excel is easy (just use the copy-button in the Gekko interface).

Scalars	ok	STRING, NAME, VAL and DATE provide most of the functionality of the ASSIGN command. In Gekko, STRING corresponds to ASSIGN ... STRING ..., whereas NAME corresponds to ASSIGN ... LITERAL ... . Note that Gekko scalars are referred to by '%' prefix, not '#' like AREMOS. There are quite a lot of string functions available.
Models	ok	Models and equations are done a bit differently from AREMOS. In Gekko, only one model can be loaded at the time, but on the other hand models solve much faster. Gekko's solvers and modeling facilities are more powerful than AREMOS', and Gekko also solves with leads. The Gekko concept of a 'reference' databank alongside the first-position databank eases comparisons of scenarios.
Econometrics	limited	Only simple OLS (and ANALYZE) is implemented in Gekko. It is not the intention to provide advanced econometrics in Gekko, but rater focus on interfaces to R, TSP, etc. (cf. R_RUN).
Graphs	partial	Gekko implements graphs via the gnuplot engine. Compared to AREMOS graphs with large template files (.gra), Gekko graphs are lacking. But still, normal graphs work fine in Gekko, and exporting data to Excel for further graphing possibilities is easy. Gekko 2.2 has improved PLOT.
Reports	almost	Printing of timeseries, lists, matrices, scalars, etc. is implemented, including TELL. TABULATE is not implemented.
Tables	differen t	Gekko has rich features regarding the production of timeseries tables and menus from config files (including html tables). But Gekko does not have an Excel-like TABLE editor like AREMOS, and some of the advanced features like mixing frequencies in tables are missing in Gekko (Gekko tables are mostly used for annual data).
File transfer	ok	Besides Gekko's own open gbk format, Gekko supports formats like tsd, csv, prn, xlsx, etc. for data interchange.
Options	ok	Gekko operates with globals option settings in the same way as AREMOS. The syntax is OPTION rather than SET, however.
Functions/pro cedures	almost	User defined functions are supported in Gekko. Library logic regarding functions/procedures will soon be added.
Store/restore settings	ok	Settings can be stored in gekko.ini similar to aremos.opt, and will be loaded when using RESTART rather than RESET. The content of the Work bank, sample, frequency, etc., is not automatically stored from session to session

		(this can be done manually if needed). Local options inside <...> angle brackets are implemented like in AREMOS.
Client/server	no	Gekko is *one* component, and there is *one* main window. Gekko can be called silently in batch mode if needed.
Editor	no	Gekko does not contain an inbuilt editor for program files or procedures, but the EDIT statement calls Notepad. External editors like VS Code, Sublime or Notepad work fine with the RUN statement. There is a XEDIT statement for Gekko 2.2.
Structure/document/write	no	This is not implemented in general (only label/source, see DOC), but Gekko stores the last SERIES statement inside the source field of individual timeseries. Some AREMOS users are happy to use custom designed documentation fields in AREMOS, and something similar (and perhaps a bit more modern) will be added to Gekko.
INTERPOLATE	no	This command is missing in Gekko (creating a higher-frequency timeseries from a lower-frequency one). Implemented in Gekko 2.2
RANK	no	This command is missing in Gekko (sorting timeseries).
REBASE	no	Rebasing a timeseries = 1 in a particular period, missing as a statement in Gekko. Implemented in Gekko 2.2
REPEAT/WHILE	no	You can use FOR loops with IF instead, but REPEAT/WHILE will be added.
UNASSIGN	no	There is actually no way to delete scalars in Gekko. This will be done.

The following section highlights some of the functionality that Gekko provides, but AREMOS does not.

Gekko component	**Comment**
Open source	Gekko is open source, and hence free to download (including all source code files), use and alter (under the GNU GPL license). Everything in Gekko is C#.NET + external modules like gnuplot, 7zip and x12a. Easy installation of Gekko with all-inclusive installer. The only thing to worry about is wheather one should install R, too - this can be done separately, before or after the Gekko installation.

Databanks	Gekko 2.0 introduces the concept of a 'reference' databank, opening up all the advantages of statements/operators like PRT<m>, PRT<q>, MULPRT<pch> etc. The default databank format (gbk) is a fast and open/nonproprietary format (zipped protobuf files). All data and internal representations are in double-precision (64 bit, around 15 significant digits).
Reading	READ/WRITE for easier data loading into the first-position or reference databank. This can be cumbersome in AREMOS via OPEN, COPY, etc.
Models	Gekko has better and faster solvers, including solvers for goals-means, and solvers for leaded endogenous variables. Models can be signed, and you can use dlog() etc. in the equations (also on the left hand side). Automatic handling of add-factors, exogenization etc. Model signatures implemented.
Translators	Gekko can translate from the older Gekko 1.8, and from AREMOS (see the TRANSLATE statement). The translators are pretty advanced, using a parser internally, so they do much more than simple search & replace.
Statistikbanken	AREMOS has a QuickData component for linking up to Global Insight's databases. Gekko has a similar component for linking up to the Danish 'Statistikbanken' database, via its new API. Meta-data regarding the timeseries is imported as labels.
R interface	Easy interface to R, so that you can manipulate your timeseries in R and get them back again.
Tables/menus	The Gekko menu and table generator is actually quite powerful, using xml and html. You can use html menus to organize the tables.
Suggestions	Suggestions pop up when the user types OPTION values. There will also be syntax suggestions from the translators, if the statement entered looks like Gekko 1.8 or AREMOS syntax.
String/name	Gekko provides a cleaner string/name distinction, and conversion possibilities (name to string: '%n' or $n, and string to name: {%s} or {s}).
HP-Filter	In-built HP-Filter function
Run status window	(Double-click the green/yellow/red light in the lower bottom of the main window.) This window tracks the progression of Gekko jobs = .gcm files running. It is practical for larger time-consuming jobs, and to see which jobs have finished.

Databank window	(Click F2). In this window, you can change the position of Gekko databanks in the search list via drag and drop.
Decomposition	The DECOMP statement has no equivalent in AREMOS. Used to track effects from equation to equation in models, and decompose these effects into contributions from precedents. User-defined informative labels for the timeseries are supported.
Equation browser	The equation browser has no equivalent in AREMOS. It is used to click through and show the equations and data in the model (via precedents and dependents). User-defined informative labels for the timeseries are supported.
MULPRT and operators	Using the concept of first-position and reference databanks, a lot of comparisons are easy to perform. "MULPRT y;" will print the difference between y in the two databanks, but more complicated transformations are possible via so-called print codes. You may for instance use "PRINT <p> x, y, z <m q>;", where the first <p> tells Gekko to print out x, y in growth percentages (<p>), whereas z is treated individually and is printed out as absolute and relative multiplier (<m q>). This way, you avoid tedious expressions like "PRINT 100*(x/x[-1]-1), 100*(y/y[-1]-1), z - ref:z, 100*(z/ref:z-1);". The operator transformations also work on expressions and lists. There are even 'long' and 'short' print codes, to suit different purposes. The SERIES statement also supports the 'short' operators, so you may state, for instance, "SERIES <p> x = 15;" to set the growth rate of x to 15%. The operators work for tables, too.
Time filters	Gekko supports time filters (see TIMEFILTER). Using these, observations may be skipped or aggregated when reported. This is useful regarding long samples.
Modes	Via the MODE statement, Gekko can run in sim-, data-, or mixed mode. For instance, databank searching is switched off in simulation mode, but active in databank mode. Mixed mode synthesizes sim- and data-mode, that is, mixes model simulation and data handling.
Unlimited	There are no limits on any datastructures in Gekko, other than what available RAM permits.

6.11.2 Compare with EViews

This section compares EViews (version 8) modules and functionality with Gekko 2.0 (as of December 2015).

The comparison is limited to data revision purposes, where the following questions can be raised.

EViews is indeed a polished and professional software product. It is not the intention here to judge the EViews capabilities regarding model solving or econometrics, but instead we will take a look at EViews for data revision purposes (on timeseries data).

The following points may be biased, or even wrong (comments are appreciated), but they reflect an honest attempt to perform some timeseries related tasks in EViews, as compared to running them in Gekko.

- When you open up EViews with a new 'workfile', it seems a bit cumbersome that the user always as the first thing has to pick out a sample (period) for the work file. Workfiles can have 'pages' (a bit like tabs in Excel), and for each page there seems to be a global sample period (defined once and for all) and fixed frequency. So if there are several frequencies, these seem to have to live on separate pages in the workfile. The databank/workfile format seems proprietary and undocumented, in contrast to the open-format Gekko databanks (protobuffers).
- It does not seem possible to refer directly to objects on other pages or workfiles. These have to be copied/cloned first, before use. For instance "copy wf1::page1\ser1", but it does not seem possible to use "wf1::page1\ser1" in expressions (in contrast to Gekko's use of colon). This will result in quite a lot of copying, and then the user must remember to delete these objects afterwards.
- It does not seem possible to obtain a line-by-line execution of statements (like in Gekko or AREMOS or other packages). The EViews user can run a whole file 'batch', or a block of statements can be marked and executed by right-clicking and choosing in the menu. The only way to run a single statement line seems to be to mark it, right-click it, and choose 'Run selected' (for which there is no short-cut). But even if the user does this, there is for instance a problem with executing the lines "%s = "hello"" and "show %s" one by one. After the first line is executed, %s no longer exists in memory when it is to be shown in the second line (this variable only seems to live while the first line is executing). So these two lines must be executed as one block, and cannot be tried out separately.
- Apparently, EViews does not provide the line number, when errors occur, neither for marked blocks of statements (run from the command window), or from batch/program files run with EXEC. This is not very user-friendly when running larger files/systems. The user can see the file involved, and how the offending line looks. But EViews inserts concrete values from control variables, so the user may

receive an error like "!m is not defined or is an illegal command in '!m = 0 + 0'", in stead of a raw depiction of the line ("!m = !m + !k"). Sometimes it can be convenient to see the concrete value of control variables in a line, but this does not make it easier to locate the line in a program file...

- In the main window, there does not seem to be a dedicated output window (like the upper part of the main window in Gekko, or the main window in AREMOS). Instead, there are floating sub-windows. If, as an example, the user wants to print out a scalar or timeseries, it only seems possible by means of opening up a sub-window with that content (in a kind of mini-spreadsheet). Maybe there are other ways to do it, but if not, quite a lot of sub-windows may clutter the working area (and these, by the way, do not seem to be closeable with the Escape key). As an consequence, back-browsing/looking in the output window like in Gekko or AREMOS does not seem possible. Related to this, it seems a bit odd that printing out a text string via "show %s" opens up a whole new sub-window and shows the result in a mini-spreadsheet with the value of the string shown for each observation in the sample. Isn't it possible to use a TELL statement like in Gekko or AREMOS? Using "print %s" seems to send the results to the printer...

- User-defined *functions* do not seem possible in EViews. It is possible to design userdefined *procedures*, and these can return stuff via their ingoing arguments. But it is not possible to have a function that returns something that can be used directly as an argument in other places/functions. So it does not seem possible to design a user-defined function product(x, y), where product(2, 3) = 6, and product(product(2,3), 4) = 24. This seems like a serious limitation, and such functions do not seem to be underway according the the EViews user forums.

- Dates seem to be internally represented as strings in EViews. In order to 'calculate' on such dates, it seems necessary to use the functions @dtoo() og @otod(), that is, date-to-observationindex and observationindex-to-date, respectively. These functions obtain the number of observations from the start period of the workfile itself, relative to some date written as a text string. But if you need an 'offset' of dates (for instance, adding five periods to some date %t), it seems necessary to use something like this: %tt = @otod(@dtoo(%t)+5); !x = @elem(y1, %tt). Of if you need to find the timeseries observations that in Gekko would be VAL x = y1[%t+5], in EViews it seems necessary to write something like this: !x = @elem(y1, @otod(@dtoo(@str(!t))+5)). Since EViews does not have user-defined functions, this is proabably not something that could be hidden/wrappted inside a function. Correspondingly, refererence like for instance y1[2010] seem a bit cumbersome to write in EViews, since it seems to involve writing @elem(fy, "2010:1"). Normal lags in timeseries, however, are non-problematic and are written like y1(-1) etc.

- In a similar vein, it seems involved to set some observation of a timeseries to some particular value. In order to do what in Gekko would be "SERIES <2010m1 2010m1> fy = fx[2000m1];" seems to involve the following in EViews: "smpl

2010m1 2010m1; fy = @elem(fx, "2000m1"); smpl @all;". It seems necessary to alter the globals sample period to do this in EViews.

- The EViews syntax originally origins from TSP, since EViews origins from MicroTSP, which again stems from TSP. It is probably a matter of taste, but somehow the syntax of EViews seems a little bit old-fashioned, for instance with its use of blanks instead of commads to separate items.

- It seems a bit strange that the EViews user can operate with both string-replacement variables (%s) on the one hand, and simultaneously on the other hand operate with string-*objects* (that can live in a workfile). These have almost the same functionality, so why two types? The same goes for scalars, where you can either use the control variable !x, or a corresponding scalar object. The reasons are probably historical, but it is a bit confusing.

- EViews does not seem to compile program files into executable code, but seems to use a so-called interpreter instead. This hampers the execution speed of EViews program files, even though EViews itself is written in (speedy) C++. The speed of model simulations has not been tested, since this is not the focus of the present section.

Index