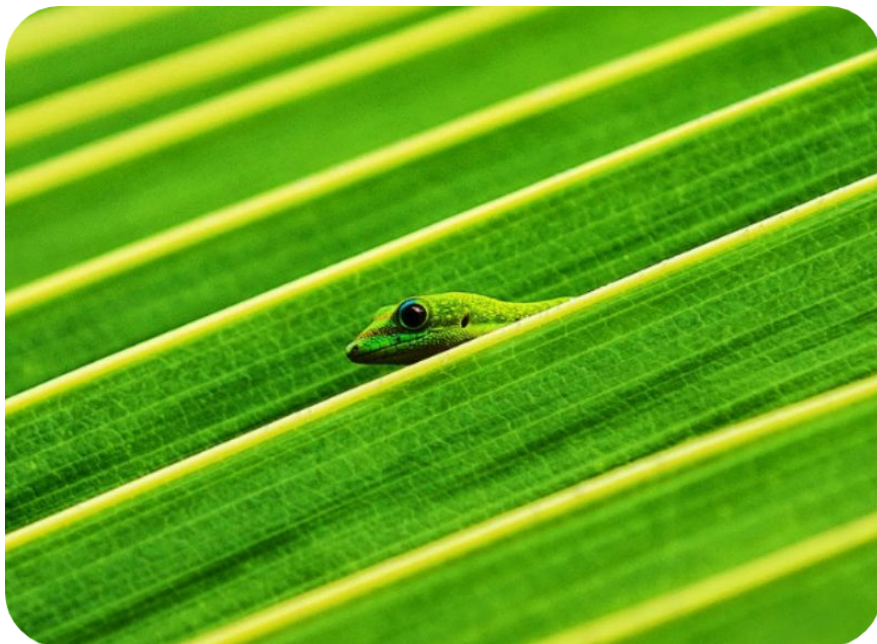




Gekko 3.0 user manual



Gekko Timeseries & Modeling

T-T Analyse

Table of Contents

Part I Gekko 3.0 user manual	6
1 New features	9
2 Note about Gekko 3.0	14
3 Setup	17
4 Basic concepts	19
5 Time periods	26
6 Databank search	26
7 Wildcards	30
8 Naked list	36
9 Filenames	39
10 Function keys, etc.	40
11 Help system	41
12 Under the hood	42
13 Guided tour	44
Part II Gekko syntax basics	46
1 Basic syntax rules	48
2 More about syntax	53
3 Indexing: list, matrix, map	58
4 Syntax diagrams	61
Part III Gekko commands	66
1 Reading guide	68
2 Command overview	69
Sim-mode command overview	74
Data-mode command overview	77
3 ACCEPT	81
4 ANALYZE	83
5 BLOCK	85
6 CHECKOFF	87
7 CLEAR	89
8 CLIP	91
9 CLONE	92
10 CLOSE	93
11 CLS	95
12 COLLAPSE	96

13 COMPARE	98
14 COPY	102
15 COUNT	107
16 CUT	109
17 CREATE	110
18 DATE	112
19 DECOMP	117
20 DELETE	121
21 DISP	123
22 DOC	126
23 DOWNLOAD	128
24 EDIT	132
25 ELSE	134
26 END	135
27 ENDO	136
28 EXIT	140
29 EXO	141
30 EXPORT	142
31 FINDMISSINGDATA	147
32 FOR	150
33 FUNCTION	156
34 GLOBAL	162
35 GOTO	165
36 HDG	167
37 HELP	168
38 IF	170
39 IMPORT	174
40 INDEX	181
41 INI	186
42 INTERPOLATE	187
43 ITERSHOW	189
44 LIST	191
45 LOCAL	209
46 LOCK	211
47 MAP	212
48 MATRIX	216
49 MEM	226
50 MENU	228
51 MODE	231

52	MODEL	234
53	MULPRT	241
54	OLS	247
55	OPEN	251
56	OPTION	257
57	PAUSE	273
58	PIPE	274
59	PLOT	277
60	PROCEDURE	288
61	PRT	295
62	R_EXPORT	307
63	R_FILE	309
64	R_RUN	310
65	REBASE	314
66	READ	316
67	RENAME	320
68	RESET	322
69	RESTART	323
70	RETURN	326
71	RUN	327
72	SERIES	331
73	SHEET	346
74	SIGN	354
75	SIM	356
76	SMOOTH	360
77	SPLICE	363
78	STOP	365
79	STRING	366
80	SYS	373
81	TABLE	375
82	TARGET	384
83	TELL	385
84	TIME	387
85	TIMEFILTER	390
86	TRANSLATE	393
87	TRUNCATE	395
88	UNFIX	396
89	UNLOCK	397
90	VAL	398

91	VAR	401
92	WRITE	403
93	X12A	406
94	XEDIT	409
Part IV	Gekko functions	412
1	Functions	414
Part V	Gekko solvers	447
1	Newton-Fair-Taylor	449
Part VI	Guided tours	456
1	Guided tour: modeling	457
	1. Installation and download	457
	2. Graphical interface etc.	458
	3. Historical simulation	461
	4. Multiplier analysis (shocks)	469
	5. Add-factors etc.	474
	6. Goal-search etc.	477
	7. Forward-looking	481
Part VII	Comparison with similar software	485
1	Compare with AREMOS	487
2	Compare with EViews	494
Part VIII	Appendix	498
1	AREMOS translator details	500
2	Gekko 1.8 translator details	502
3	Gekko 2.0 translator details	504
4	Assignments	505
5	Missings	506
	Index	515

Part I

1 Gekko 3.0 user manual

Last modified: 5/12 2019

Gekko 3.0 was released in April 2019 as a stable version, but some glitches are still to be expected. Note that in some sections on the help pages, there is a "[New in 3.0.x]" marker, which indicates that the functionality is new in that particular Gekko version in the 3.0 series. There are still some known issues regarding DECOMP and TIMEFILTER (regarding issues and stability, see [this](#) page).

Gekko Timeseries and Modeling Software is an open-source software system for handling and analyzing timeseries data, and for solving and analyzing large-scale economic models. See the Gekko homepage: www.t-t.dk/gekko. The current 3.0 version is a release version (stable version). See the page www.t-t.dk/gekko/gekkoversions regarding the different Gekko versions available at the moment, and how to choose between them.

The user manual contains the following chapters:

- [Introduction](#). Introductory chapter.
- [Gekko syntax basics](#). In this chapter, the syntax is described.
- [Gekko commands](#). This chapter describes in detail the purpose of the different Gekko commands, 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.
- [Gekko solvers](#). A description of some of the Gekko solvers.
- [Guided tours](#). Step-by-step examples with screenshots etc.
- [Comparison with similar software](#). Gekko is compared to AREMOS and EViews.
- [Appendix](#). More info on automatic translation.

The present introductory chapter contains the following sections:

- [New features](#). A list of changes in Gekko since the previous version.
- [Setup](#). How to setup Gekko on a pc.
- [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.
- [Filenames](#). How filenames and paths are handled in Gekko.
- [Function keys](#). A list of function keys like F1, F2, etc. 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.
- [Guided tour](#). Some step-by-step examples of doing stuff with Gekko (mostly how to solve models).

1.1 New features

The current version of Gekko 3.0 is a release version. This means that it is stable, has been tested thoroughly, and that the syntax and functionalities are fixed. Users of the Gekko 3.0 stable version are in general advised to upgrade to the 3.1.x series. The 3.1.x series is a kind of "stable" development series, where care is taken not to break anything relative to the 3.0 version. The 3.1.x versions contain some extra functionality, but also improved error messages, improved graphical interface, etc. Read more about Gekko versions her: www.t-t.dk/gekko/gekkoversions.

Gekko is under continuous development, so the features are augmented on a regular basis in the development versions (versions with an uneven second number, for instance 3.1.x). You can find more detailed descriptions [here](#) (development versions), or even more detailed in the [changelog](#). Regarding Gekko 2.0 and earlier, see [this page](#).

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) 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 commands, 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 commands [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 calculations are treated more like vector operations in Gekko 3.0, lags no longer accumulate period-for-period, if the left-hand side variable is present with lags on the right-hand side (so-called "lagged endogenous"). So a series expression like `x = x[-1] + 1;` no longer accumulates automatically (augments `x` with 1 for each period); instead the alternatives `x ^= 1;` or `x <d>= 1;` could be used. If accumulating behavior is needed, the `<dyn>` option can be set, for instance `x <dyn> = x[-1] + 1;;`, or for several series statements a [block](#) structure can be used: `BLOCK series dyn = yes; ... ; END;`. Setting dynamic mode affects speed negatively, and should therefore not be set 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 commands 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 command file 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()`, `hpfiler()`, `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 command 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 commands: `BLOCK`, `CUT`, `LOCAL`, `GLOBAL`, `MAP`, `VAR`.
- Removed commands: 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()`, `hpfiler()`, `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](#) command. 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 command 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](#) command completely overhauled, see demo graphs [here](#). Graphs can be controlled in a lot of new ways, either as options in the PLOT command, 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](#) commands 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](#) command improved, including linear restrictions on parameters.
- [INTERPOLATE](#) and [REBASE](#) commands implemented.
- [XEDIT](#) command to open up a dedicated and in-built xml editor (for graph and table templates).
- [MATRIX](#) command 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](#) command regarding new options in Gekko 2.2.

Regarding Gekko 2.0 and earlier, see [this page](#).

1.2 Note about Gekko 3.0

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, and some of the most important changes regarding syntax are listed below. At the end of this page, you will also find a list of components or commands that do not work. These minor issues will be fixed in the form of patches to version 3.0.

Regarding lists of new commands, 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.

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 calculations are treated more like vector operations in Gekko 3.0, lags no longer accumulate period-for-period, if the left-hand side variable is present with lags on the right-hand side (so-called "lagged endogenous"). So a series expression like `x = x[-1] + 1;` no longer accumulates automatically (augments `x` with 1 for each period); instead the alternatives `x ^= 1;` or `x <d>= 1;` could be used. If accumulating behavior is needed, the `<dyn>` option can be set, for instance `x <dyn> = x[-1] + 1;`, or for several series statements a [block](#) structure can be used: `BLOCK series dyn = yes; ... ; END;`. Setting dynamic mode affects speed negatively, and should therefore not be set 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 commands `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` command 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()`, `hpfiler()`, `fromseries()`.

Issues list

The following is a list of commands 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.

As noted above, some in-built functions have been changed regarding "signature".

Stability

Gekko 3.0 has been tested quite a lot by now, and the first users of the preliminary pre-alpha versions of 3.0 started in December 2017. Hence, many of the main components are well-tested, and in this sense, Gekko 3.0 should not feel unstable. Still, some glitches are still to be expected, though, and such glitches will be fixed in patches to Gekko 3.0. Gekko 3.0 validates a large number of test-cases taken from (and translated from) Gekko 2.4.

Unstability reasons

A major source of potential instability is the fact the the parser has been completely rewritten from version 2.4 to 3.0. In addition to this, databanks are now a lot more flexible, allowing all kinds of objects to be stored and retrieved, and this may produce glitches, too. The internals of array-series objects have been completely reworked, but this is well-tested.

Series objects are handled very differently in version 3.0 compared to 2.4. In 2.4 and all previous versions, series operations were implemented very differently from, say, scalar operations. In Gekko 3.0, series operations treat series more like vectors, so in a sense, the series addition $x + y$ is performed in *one* operation in Gekko 3.0, an operation that resembles the addition of two vectors in linear algebra. This difference has many ramifications in the Gekko source code, but the advantages are large, too.

Printing and plotting components have been rewritten from scratch, among other things in order to accommodate mixed frequencies and array-series. Still, PRT and PLOT are well-tested.

User functions and procedures are implemented in a different and better way, so these should work more as expected than the case was regarding Gekko 2.4.

The graphical interface is more or less untouched from 2.4. Also, databank handling (opening and closing databanks, search order etc.) has not been touched, except for the introduction of the new Local and Global databanks.

The internals of the solving facilities (sim-[mode](#) and [SIM](#)) have not been altered since Gekko 2.4 and should therefore be stable.

1.3 Setup

Installation

In order to install Gekko as a standalone package for economic analysis, go to www.t-t.dk/gekko, and choose 'Download' --> 'Installer (stable version)', or go directly to www.t-t.dk/gekko/installer. If you have problems installing, please consult the trouble shooting guide: www.t-t.dk/gekko/troubleshooting. 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 ADAM in any way, and is being used for other models, too.

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 (for instance by typing 'gekko' in the Total Commander command line), 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 commands 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.

1.4 Basic concepts

This document describes some of the basic concepts used in 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') or undated ('u'). [IMPORT](#) can handle (collapse) higher frequencies than months, if needed.

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 succession, where the first-position databank has number 1 on the databank list (cf. the F2 window: click F2).

Gekko can [READ](#)/[WRITE](#) such databanks as external 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 startup, the Work databank has the 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](#) commands for more on this. See the F2 window regarding 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 is altered).

If you need to import data into an existing (opened) 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](#) command is often practical. You may use [WRITE](#) to write the first-position databank to 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 databanks 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 command 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 bankless variables, since this databank is only used for comparison purposes, cf. [MULPRT](#), [PLOT](#)<m>, [COMPARE](#), and similar commands. You may refer to a variable in the reference databank with `ref:x` or the shorter `@x`;

Series

A timeseries (or just: series) can have frequency annual, quarterly, monthly, 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 `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 `+`. 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`.

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. List stores variables in a sequence (by numbers 1, 2, etc.), 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 for advanced users: 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. But maps and array-series are in reality not that different, it is mostly a question of syntax. For instance, `y['a', 'x']` could be emulated with the map call `#y['#a']['x']`, where `#y` is a map, `#a` is another map stored inside `#y`, and `x` is a normal series stored inside `#a`. But the array-series notation is more convenient, and array-series have special capabilities regarding summing, printing, etc.

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 function in the same way. Consider the list `#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. 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. If you need to put three series `a`, `b`, and `c` into a list, you should use `#m = (a, b, c);`. So defining a list while omitting parentheses on the right-hand side always produces a list of *strings*.

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 command, 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 commands, 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 command). 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](#) command 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 commands 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](#) command. This command 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 a cleanup-command: [RESTART](#). This command 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 commands, for instance MODEL and READ commands, that the user wishes to "survive" a RESTART). If you wish a clean state without any potential gekko.ini file, use [RESET](#).

Creation

In sim-mode you have to [CREATE](#) a series before you update its values/observations with the [SERIES](#) commands (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 command.

Periods

Note that commands 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](#) command. In assignments, the time period may be stated before or after the left-hand side variable, so both `<2020 2030> x = 100;` and `x <2020 2030> = 100;` are legal.

There are some details regarding periods. Most commands that involve timeseries use the global time period if no local time period is stated, for instance commands like PRT, IMPORT, EXPORT, etc. For some of these commands, 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 commands [COPY](#), [READ](#) and [WRITE](#), omitting a local time period does not entail the use of the global time period. These commands 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 commands, 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` commands (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 [TABLE](#)s 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](#) commands. Regarding equation syntax, you may consult the latter part of the [MODEL](#) help file, if you need more information on this. (Models are cached in binary form on the user's hard disk in order to load faster next time).

Files

Regarding file names, you may use relative paths like `'\subfolder\data.txt'`. Using relative paths makes it easier to move a system of command 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.

No blanks

Generally, sequences of elements are delimited by commas, not blanks. Gekko 3.0 has a number of capabilities regarding the transformation of such lists, for instance setting or removing commas instead of blanks, setting or removing quotes, etc. See the Gekko main window, under Edit --> Paste as.... [**not done yet**].

Command files/batch job

Gekko commands can either be run directly from the Gekko main window, or assembled in a command file (script file) for later execution. Command files can be run with the RUN command. 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.

1.5 Time periods

Many Gekko commands accept a local time period stated inside the <>-brackets. For such commands, 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 commands where lack of local period means all observations

COPY	Handling variable objects
READ, WRITE	External file storage

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 commands `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.

1.6 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 accessed.

Databanks search can be controlled with an option, or via the `MODE` command, cf. the next section. In general, databank search can be practical, but the user should be aware of the pitfalls, especially of several open databanks contain variables with the same names.

The remainder of this page tries to answer the following question: In commands 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, MODE 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 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](#) command. If not found in any of these, the Global databank is queried at last. Note that the Ref databank is never searchable, so a bankless variable x 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 command 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 command 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 identifiers 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 print 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).

Commands without databank search

A few commands disallow databank searching completely in order to avoid ambiguities. In these commands, `bk1:x` is still understood as `x` from the `bk1` databank, but the bankless `x` will be understood as `x` from the first-position databank, without looking elsewhere for the variable.

List of commands 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 commands, `*: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 command 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 on [MODE](#) etc.

1.7 Wildcards

Wildcards are used to search for variables in one or more databanks. Internally in 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.

Wildcards can also be used to search for variables in a list

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, same as `{'x*y'}`.

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 commands 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 commands 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 command 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 wildcards have 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:

```
[ '*' ] --> fy!a, fy!q, %y, #y
```

Everything is matched. This may seem ok, but then what about this:

```
[ '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 start:

```
[ 'f*' ] --> fy!a, fy!q
```

Suddenly the two series match again, because the star matches '!a' and '!q'. But if the star is first, we get:

```
[ '*y' ] --> %y, #y
```

Now '!a' and '!q' are not matched, but on the contrary, the star matches '%' og '#', 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:

```
[ '*' ] --> fy!a
```

Only the active frequency is matched (we assume it is annual). No starting '%' or '#' are matched.

```
[ '*!*' ] --> fy!a, fy!q
```

Here, all frequencies are matched.

```
[ '%*' ] --> %y
```

This is how to match scalars. Collections are ['#*'].

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'.

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'] }`.

1.8 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 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;`, and regarding lists of values, the parentheses may also be omitted, so that `y = (1, 2, 3);` may be replaced by the 'naked' `y = 1, 2, 3;`, where `y` is a series.

Important note: in many ways, a naked list definition is very similar to a normal list definition with enclosing parentheses, but here is one difference to keep in mind. 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'
```

Compare this with normal list definitions:

```
#m1 = ('b', 'c'); #m2 = ('a', #m2, 'd'); //result: 'a', ('b', 'c'), 'd'
```

Here, `#m2` becomes a nested list (you could use the `flatten()` function to unnest it).

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 *`.
- 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)
```

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.

1.9 Filenames

Gekko accepts relative paths, relative to the Gekko working folder. Consider, for instance, that you have a command file 'job.gcm' with the following READ-statement inside the 'job.gcm' file:

```
READ \banks\data;
```

Now, Gekko will add the sub-folder \banks to the Gekko working folder path. If the Gekko working folder is "C:\Projects\Modell1", the READ statement is translated into:

```
READ C:\Projects\Modell1\banks\data.gbk;
```

The extension .gbk is automatically added if missing in the READ command.

You may use strings to compose file paths and names:

```
%s1 = 'Projects';  
%s2 = 'Modell1';  
%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](#) command. Path's must use the '\' (backslash) or '/' (frontslash), 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;".

Frontslash is allowed too, for instance:

```
READ C:/Projects/Modell1/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' ;").

At some point it may be preferable to add the sub-folder to the path of the executing command file, rather than to the Gekko working folder. Choosing between the two ways of interpreting relative path's is not completely obvious, however.

1.10 Function keys, etc.

Function keys are used for quick access to specific Gekko commands. 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 command (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.
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.
Ctrl+Enter	New line in command prompt, without issuing the command line.
Mark+Enter	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 command file (.gcm) and executing them with RUN .
Ctrl+M	Jump to Main tab.
Ctrl+O	Jump to Output tab.
Ctrl+U	Jump to Menu tab.

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' 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, including matrices.

1.11 Help system

It may be difficult to remember all the commands and the exact syntax for each command. The function key F1 (or typing 'HELP') accesses the Gekko help system. If you cannot remember the exact syntax for a particular command, 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](#).

1.12 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 timeseries 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 and months), only constrained by working memory. All values and calculations are double-precision (64-bit) internally, and missing values are handled consistently. The timeseries 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 (command 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).

1.13 Guided tour

Instead of painstakingly reading through a lot of descriptions of commands etc., you might prefer to jump right into simulating a model in Gekko, and analyzing the results. For the purpose of this, a guided tour has been created (the tour will entail some typing though), where each step is explained, but without delving into too many details.

The guide can be seen here:

- [Gekko Guided tour](#) (external link, Gekko 2.0 simulation examples, not updated to 3.0 yet)

Part II

2 Gekko syntax basics

This chapter 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. The chapter contains the following sections:

- [Basic syntax rules](#). A section on syntax basics.
- [More about syntax](#). More details about how the syntax works.
- [Syntax diagrams](#). Diagrams that explain the basic components of the syntax.

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

Almost all commands start with a **command name**, for instance PRT (for printing). You can see the commands sorted into categories [here](#), or the alphabetical list of commands [here](#). Beware that user-defined [procedures](#) may look similar to commands. Assignments like [SERIES](#), [VAL](#), [LIST](#), etc. may omit the command name.

Many commands accept an **option field** right after the command 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 command. But many other options may be set, for instance `PRT <pch>` for percentage printing, or `PRT <filter=avg >`. In `PRT <filter = avg>`, the **option type** is 'filter' and **option value** is 'avg', whereas in `PRT <pch>`, the option type is 'pch', and the option should be understood as short-hand for 'pch = yes'. Many of the options are of yes/no-type (boolean), and instead of for instance 'pch = no', the user may use the shorter 'nopch'. In assignments, the option field may be stated before or after the left-hand side variable, so both `<2020 2030> x = 100;` and `x <2020 2030> = 100;` are legal.

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** (,).

All commands end with a **semicolon** (;), and the commands may span multiple lines. (If you need a multi-line command in the user interface, use Ctrl+Enter to add newlines, and then mark the whole block and press Enter).

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 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 symbols must also be stated on the left-hand side of assignments 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 shorter `#[d]` or `#m.d`). Matrices use two dimensions, for instance `#a[2..3, 1..%n]`. You can use brackets for strings, selecting 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 commands, 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 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, Gekko will first look for `pxa` in the first-position databank, and afterwards in the other open databanks (except Ref). 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).

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 in Gekko, but may be put inside Gekko strings (the string `'The name "Peter" has 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 `abc123` (without quotes). 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 command/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)`, 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`.

Names (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.

Details

Some syntax from the 2.0 series 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 a 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 be 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.** There are several reasons for this. 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 sigil `'%'`, 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 has been deprecated in Gekko 3.0, providing simpler logic and programs that are easier to read.
- 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, the user can use `%s in #m`, or `#m.contains(%s)`.
- 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 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', and 'c' into the list `#m` or `y = 1, 2, 3;` to put the three values 1, 2, and 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 "#m = (a, b, c);" is very different from "#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 just inserts three strings. In the former case, you may use `PRT #m;`, 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 (concatenating) 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.

2.2 More about syntax

Below, some of the main concepts of the Gekko 3.0 syntax are explained in more detail.

See the page with [syntax diagrams](#) if the basics of names, expressions, etc. is confusing.

Banks, symbols, names, frequencies, indexes

A variable may be stated in the following way:

```
[bank]  [:]  [symbol]  [name]  [!]  [freq]  [indexes]
```

For instance, `b1:x!q` refers to the quarterly (!q) series `x` in the `b1` databank. If the series is an array-series, `b1:x!q['a', 'b']` would refer to the sub-series `['a', 'b']` (that is, with two-dimensional indices 'a', 'b') of `x!q`. Frequencies are not used for non-series types.

The symbols are used in the following ways: series (including array-series) have no symbol. Scalars (value, date, string) start with `%` symbol, and collections (list, map, matrix) start with `#` symbol.

If the databank is omitted on a variable in a command 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 for `x1!q`.

For array-series, the array indexes may sometimes be omitted, so that you may write `PRINT x;` instead of `PRINT x[#i, #j];`, printing out all the elements.

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 type symbols in Gekko ('%' and '#' to start scalar and collection names), the single quotes sometimes be omitted in those cases where a series would not make sense as input. 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 (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). Using a trailing comma like `#m = a,;` makes it a list.

Indexes [...]

Regarding indexes of array-series or other variables, single quotes on a string can in general be omitted (both the following are valid):

```
PRINT x['a', 'b']; //strict
PRINT x[a, b];     //short
```

Indexes are often used on lists to pick out items (so-called slicing).

Name-substitution {...}

The `{}`-curlies are used for name-composition, and in general you may think of `{...}` as simply a sequence of 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, `{...}` curly braces removes 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, so instead of the

array-series `x['i1', 'j1']`, the user may use simply a series called `xi1j1`. If the lists `#i` and `#j` contain the *i*- and *j*-elements, you may print the series: `PRINT x[#i, #j];`, or with normal series: `PRINT x{#i}{#j}`.

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. 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` is equivalent to `#m[a]` or `#m['a']` (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`.

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.

Lists are defined like for instance `(1, 2)`, a two-element list. Note that a singleton list 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 (a column vector). A nested list could be stated like `((1, 2), (3, 4))`, which for a matrix would be `[1, 2; 3, 4]`. 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 `#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 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 Python's convention, 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 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 indices are 0-based in Python. Both `m[0][0]` and `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 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 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]]
```

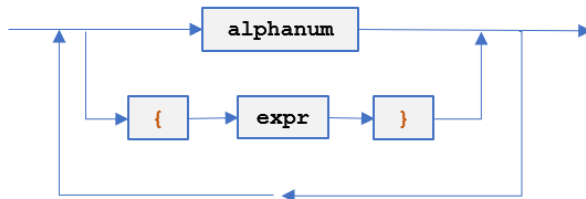
In this case, there is no difference, apart from the expected type error regarding `m5`.

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)



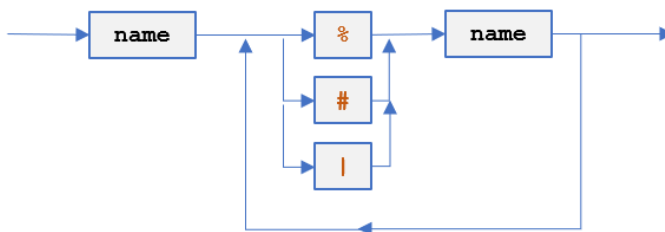
```

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. 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 command files, procedures and functions, it is often best to use normal `name` instead of `cname`, for readability and maintainability.

A variable name is a precise reference to an object residing in a particular Gekko databank. It may include type symbols `%` or `#`, or frequency `!`. The upper part of the diagram illustrates timeseries, which have no type symbols and may include a frequency. The lower part of the diagram illustrates scalars and collections, starting with a type symbol.

```
a
a%s
a!q
{%s1}!{%s2}
%s
#m
%{%s}
%(a%b)
```

A `varname` can reside in any databank (or [MAP](#)), and a `bankvarname` is hence designated as follows:

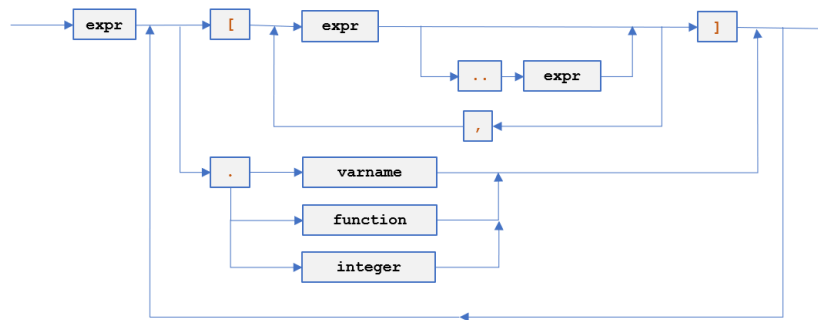
```

graph LR
    Start(( )) --> varname
    Start --> name
    name --> colon
    colon --> varname
    cname --> varname
    style Start fill:none,stroke:none
  
```

```
a
b:a
b:%s
b:#m
{%s1}:{%s2}
@x
@x!m
@%s
@#m
```

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)

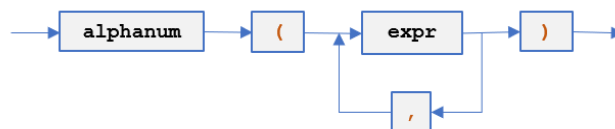


```
a[2020]
#m[1..2, 3..10]
(a + b)[2020q1]
a['b'][2020]
#m.x
#m.f('a')
a.1
a['b'].1
a!q[2020q1]
```

The dot (.) is used in three ways. The expression `#m.x` picks out the series `x` from the map `#m` (alternatively, `#m['x']` 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, lagging one period.

The function syntax is completely standard:

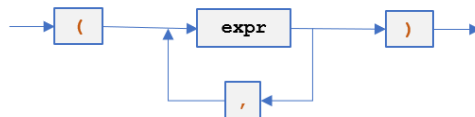
function (call of function)



```
f(100)
f('a', 100)
f(1+2, 3+4)
```

Lists are defined in the following way:

list (list definition)

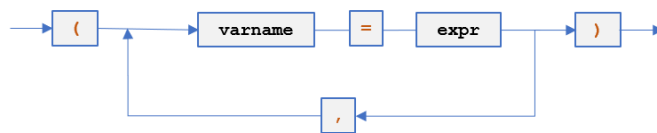


NOTE1: "(expr)" is not considered a list, use "(expr,)" for a one-element list. In general, a trailing comma is allowed. See also naked lists.

```
(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)

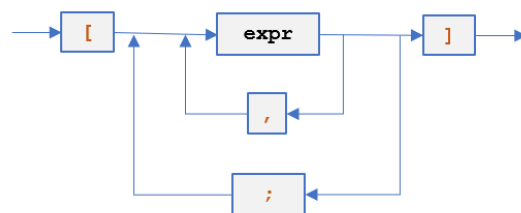


NOTE: Dollar or indexer may be used with varname. Type may also be indicated, for instance (VAL %v = 2). As for list, trailing comma is allowed

```
(%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:

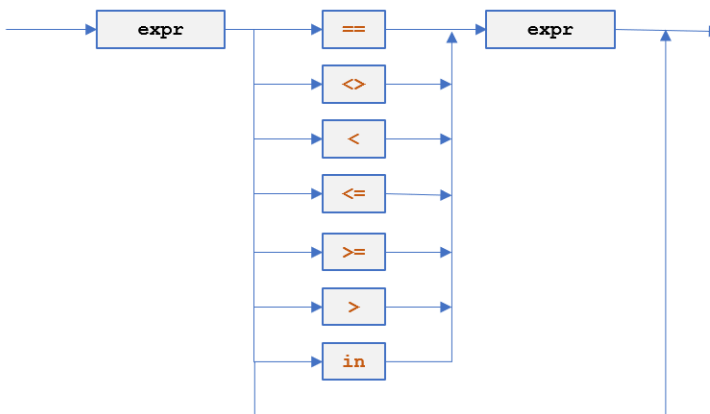
matrix (matrix definition)



```
[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.

Part III

3 Gekko commands

This chapter describes in detail the purpose of the different Gekko commands, the syntax to be used, the results produced, together with examples etc. Please select a command on the menu at the left.

Regarding general syntax, the reader may consult the short chapter on this [here](#).

Apart from the command sections, the chapter contains an overview:

- [Command overview](#). The commands are listed by category, and you may choose to see [MODE](#)-specific versions of the this list: [sim-mode](#) or [data-mode](#). See the chapter '[Gekko commands](#)' for an alphabetical list of commands, and the [functions](#) section to see functions.

3.1 Reading guide

The subjects in the chapter "Gekko commands" follow a general pattern similar to layout below. Regarding general syntax, you may read a short description [here](#).

<i>Introduction</i>	A brief description of the command.
<i>Syntax</i>	<p>The general syntax for the command. Many of the commands can use arguments or options.</p> <p>The following conventions are used in the description of the syntax:</p> <ul style="list-style-type: none">• Commands are in capital letters (for instance: PRT).• Predifined keywords are in capital letters (for instance: ROWS, or ROWS=yesno)• Elements that are defined elsewhere in the syntax definition are in <i>italics</i> (for instance: <i>period</i>). <p>Many commands accept a <i>period</i> argument. When using the period argument, the command will only be performed for the local time period, for instance</p> <pre>COMMAND < period > ... ; //for instance SIM <2015 2020></pre>
<i>Example</i>	The examples serve to illustrate the typical or common uses of the command.
<i>Note</i>	The notes concerns exceptions to the command and any specific features of the command. These notes may also include comparisons to other commands.
<i>Related options</i>	A list of related options.
<i>Related commands</i>	A list of commands with similar functions, or other commands typically used together with the specific command

3.2 Command overview

Note: You may consult the specialized overviews regarding sim- and data-[modes](#) here:

- [Sim-mode commands overview](#)
- [Data-mode commands overview](#).

Introduction

Below, all Gekko commands are listed, grouped together by functionality (regarding functions, see the chapter on these: '[Gekko functions](#)'). Before delving into the particular commands etc., you may prefer reading some introductory guides:

- [Setup](#)
- [Basic concepts](#)
- [Guided tour](#)

Databanks

At startup, Gekko operates with two databanks; 'Work' (first-position, working bank) and 'Ref' (reference, baseline bank). There are the following commands 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 command finds timeseries with missing values.
TA	
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 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 command 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 command 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	Opens up the decomposition window
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 commands 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.

Command files

Larger tasks can be run by means of command files (.gcm). There are the following commands related to such files:

RUN	Runs a .gcm command file. Use the EDIT command 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-command 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 command file or function definition.
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
---------------------	-----------------------------------

R integration

R_FILE	Starts a R session, with a particular R file as starting point
R_EXPORT	Decorates the R file with matrices from Gekko
R_RUN	Runs the decorated R file, and returns matrices back to Gekko

Miscellaneous

The following commands 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.

3.2.1 Sim-mode command overview

Note: You may consult the general overview regarding all commands here:

- [General command overview](#)

Introduction

Sim-mode (cf. [MODE](#)) is focused on solving models, comparing scenarios etc. Below, the different Gekko simulation related commands are listed, grouped together by functionality (regarding functions, see the chapter on these: '[Gekko functions](#)'). The commands listed below are the core commands regarding model simulation.

Databanks

At startup, Gekko operates with two databanks; 'Work' (first-position, working bank) and 'Ref' (reference bank). There are the following commands 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 command finds timeseries with missing values.
TA	
HDG	Inserts a heading (description) into a gbk databank

Timeseries

Timeseries exist as objects in a databank. Frequency can be annual, quarterly, monthly 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 command 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 command 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	Opens up the decomposition window
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 commands 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.

Command files

Larger tasks can be run by means of command files (.gcm). There are the following commands related to such files:

RUN	Runs a .gcm command file. Use the EDIT command 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-command 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 command 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 commands 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.

3.2.2 Data-mode command overview

Note: You may consult the general overview regarding all commands here:

- [General command overview](#)

Introduction

Data-mode (cf. [MODE](#)) is focused on databanks, handling of timeseries, data revision and similar purposes. Below, the different Gekko data related commands are listed, grouped together by functionality (regarding functions, see the chapter on these: '[Gekko functions](#)'). The commands listed below are the core commands regarding data handling.

Databanks

At startup, Gekko operates with two databanks; 'Work' (first-position, working bank) and 'Ref' (reference bank). There are the following commands 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 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 command 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 command 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

Command files

Larger tasks can be run by means of command files (.gcm). There are the following commands related to such files:

[RUN](#)Runs a .gcm command file. Use the [EDIT](#) command 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-command 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.

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 command file or function definition.
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
---------------------	-----------------------------------

R integration

R_FILE	Starts a R session, with a particular R file as starting point
R_EXPORT	Decorates the R file with matrices from Gekko
R_RUN	Runs the decorated R file, and returns matrices back to Gekko

Miscellaneous

The following commands 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.

3.3 ACCEPT

ACCEPT is used to input data to Gekko, during a session. See also [PAUSE](#).

Syntax

ACCEPT *type* *variable* *message*;

<i>type</i>	Choose between val, date or string. For string type, you do not need to enclose the input in quotes.
<i>variable</i>	The name of the variable
<i>message</i>	Text string to be displayed (please remember single quotes). You can use '\n' to insert a new line.

Examples

The command 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;
SERIES {%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.

Related commands

[RETURN](#), [STOP](#), [EXIT](#), [PAUSE](#)

3.4 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**;

<i>period</i>	(Optional). Local period, for instance 2010 2020, 2010q1 2020q4 or %per1 %per2+1.
<i>variables</i>	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 commands

[OLS](#)

3.5 BLOCK

A BLOCK structure is used to set the time period and/or other options temporarily. 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 *period*, *option1*, *option2*, ...;

<i>period</i>	(Optional). Local period, for instance <code>2010 2020</code> , <code>2010q1 2020q4</code> or <code>%per1 %per2+1</code> . The period must be first in the list of BLOCK options, and must include the TIME keyword, for instance <code>BLOCK time 2020 2030; ... ; END;</code>
<i>option</i>	A list of option settings (OPTION statements, without the 'OPTION' keyword).

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

```

Note

BLOCK can also be used to change frequency temporarily, for instance `TIME 2021 2023; BLOCK time 2021q1 2023q4, freq = q; y = 100; END;`. This will create the quarterly series `y!q` defined over 2001q1-2023q4. After the commands have been run, the time period will be back to annual 2021-23.

You can use any [OPTION](#) setting together with BLOCK, just omit the 'OPTION' keyword, and separate options with commas.

Related commands

[LOCAL](#), [OPTION](#), [TIME](#)

3.6 CHECKOFF

The command 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.
<i>variables</i>	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)
```

Currently ignored variables can be seen with

```
CHECKOFF ?;
```

There is no CHECKON command. The CHECKOFF command is non-additive (like the ENDO and EXO commands). To eliminate a CHECKOFF-variable, just remove it from the list given to the CHECKOFF command. To clear the CHECKOFF-list, issue a CHECKOFF command 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 command to work, "OPTION solve gauss conv ignorevars" must be set to 'yes' (which is its default value).

CHECKOFF is also related to the [ITERSHOW](#) command. 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 commands

[SIM](#), [OPTION](#), [ITERSHOW](#)

3.7 CLEAR

The CLEAR command is used to clear databanks in memory (that is, delete all variables inside the databanks).

Syntax

```
CLEAR ;  
CLEAR databank ;  
CLEAR <FIRST REF> ;
```

<i>databank</i>	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](#) command. To clear the entire workspace, see the [RESET](#) and [RESTART](#) commands.

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 commands

[DELETE](#), [RESET](#), [RESTART](#)

3.8 CLIP

CLIP has the same syntax and functionality as [SHEET](#), so please see this command.

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](#) command 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 excel decimalseparator = [comma|period].

Related commands

[SHEET](#), [PRT](#), [PLOT](#)

3.9 CLONE

The CLONE command 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 command in the following way:

```
MODEL m;  
READ data;  
TIME 2015 2050;  
SIM;  
CLONE;  
SERIES 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 command.

Note

The READ command 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 command is equivalent to READ<first> followed by CLONE.

Related commands

[READ](#), [OPEN](#), [MODEL](#), [MULPRT](#), [SIM](#), [DECOMP](#)

3.10 CLOSE

The CLOSE command 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=...> databanks;`

SAVE=	With <code>CLOSE <save=no></code> , Gekko will not write the databank to file, even if the databank contents has changed. See also <code>OPEN <save=no></code> .
databanks	The databank(s) to be closed. A star (*) indicates all open databanks opened by means of the OPEN command. You may provide a list of banks like <code>CLOSE db1, db2;</code>

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 command (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).

Closing of more than one databank (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](#) command.

Related commands

[OPEN](#), [CLEAR](#), [DELETE](#)

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 commands

[CLEAR](#), [RESTART](#), [RESET](#), [CUT](#)

3.12 COLLAPSE

COLLAPSE transform one higher-frequency timeseries to a lower-frequency timeseries, for instance converting quarterly data to annual data. Use [INTERPOLATE](#) to do the inverse transformation.

Syntax

```
COLLAPSE lf = hf method;
```

lf	Lower frequency timeseries. Frequency can be indicated with suffix !a, !q or !m. Banknames may be used.
hf	Higher-frequency timeseries. Frequency can be indicated with suffix !a, !q or !m. Banknames may be used.
method	(Optional). Choose between: <ul style="list-style-type: none"> • total: The higher-freq observations are summed. • 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'.

- 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](#)).

Example

Use this to convert frequency:

```
COLLAPSE fY!a = fY!q;
```

Since the method is 'total' as default, this will create the annual timeseries `fY!a` where each annual observation is the sum of the corresponding quarters in `fY!q`.

```
COLLAPSE fY!a = qbank:fY!q first;
```

With option 'first', the first quarter of each year would be used instead of summing the quarters. Here, the variable is taken from the databank `qbank`.

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.

See also [IMPORT](#)<collapse> regarding higher frequencies than quarters.

Related commands

[INTERPOLATE](#), [SERIES](#), [CREATE](#), [PRT](#)

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 command is an upgraded version of the same command in Gekko 2.4 and earlier. The Gekko 3.0 command fully replaces and improves the menu item 'Utilities' --> 'Compare two databanks...' in the Gekko user interface.

Syntax

```
COMPARE < period ABS=... DUMP REL=... SORT=... PCH=... > variables
FILE=... ;
```

period	(Optional). Local period, for instance <code>2010 2020</code> , <code>2010q1 2020q4</code> or <code>%per1 %per2+1</code> .
ABS=	Absolute differences smaller than the value are not shown, for instance <code><abs = 150></code> .
DUMP	If this option is set, a list <code>#dif</code> will be constructed, containing the list of different timeseries.
REL=	Relative differences smaller than the value are not shown, for instance <code><rel = 0.01></code> equivalent to 1%. You may alternatively use PCH for the same purpose.
SORT=	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=	Percentage differences smaller than the value are not shown, for instance <code><rel = 1.0></code> corresponding to 1%. You may alternatively use REL for the same purpose.

variables	A list of variable names. If no variables are given, the full databanks is compared. The names are separated by comma (like <code>x, y, z</code>), and a list <code>#x</code> of names should be used with <code>{}</code> -braces: <code>{#x}</code> . Regarding array-series, you may either indicate the name of the array-series itself (<code>x</code>), in which case all sub-series are checked, or you may state individual elements (like <code>x[a, k]</code>).
FILE=	Filenames may contain an absolute path like <code>c:\projects\gekko\myfile</code> , a relative path like <code>\gekko\myfile.gbk</code> , or be stated without a path. Filenames containing blanks and special characters should be put inside quotes. 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 command-driven.

Related commands

[MULPRT](#), [PRT](#)

3.14 COPY

The command is used to copy variables, either inside a databank, or between databanks.

Note that 'naked' [wildcards](#) are allowed in this command, so you may for instance use the shorter `a*b` instead of `{'a*b'}`.

Syntax

```
COPY < period RESPECT FROMBANK=... TOBANK=... ERROR=... PRINT >
names1 TO names2;
```

<i>period</i>	(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.
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. You may optionally use AS instead of 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.
<i>names1</i>	Variable name(s) or list(s) (wild-cards are allowed). You may prepend a databank name as bank:variable.
TO	(Optional). The TO part of the COPY command is optional. If omitted, the variables will be copied to the first-position databank (with the same names).
<i>names2</i>	(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.

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 or 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](#) command), 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 command 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 command:

```
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 commands

[CLONE](#), [RENAME](#), [INDEX](#), [DELETE](#)

3.15 COUNT

The command is used to search for variables in databanks, using wildcards.

The COUNT command is essentially a compact [INDEX](#) command without the output.

Note that 'naked' [wildcards](#) are allowed in this command, so you may for instance use the shorter `a*b` instead of `{'a*b'}`.

A wildcard like '*' does not match everything in Gekko: it only matches (in the first-position databank) variables with no '%' and '#' symbols, and only matches the current frequency. You may use the special wildcard '**' to match all variables in a databank, or '***' to match all variables in all databanks.

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.
<i>wildcard</i>	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 <code>a*x</code> to find all variables starting with 'a' and ending with 'x', or <code>a?x</code> 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 command 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](#) command 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 commands

[LIST](#), [INDEX](#)

3.16 CUT

Closes any open PLOT or DECOMP windows.

This can also be done via the button 'Close all PLOT and DECOMP windows' 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).

Syntax

`CUT ;`

Related commands

[CLS](#), [RESET](#), [RESTART](#)

3.17 CREATE

This command 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 ? ;

<i>variables</i>	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 commands

[SERIES](#), [DELETE](#)

3.18 DATE

The DATE command 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](#) command.

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)	<p>Converts the date d into a new date with frequency f (string), and option opt (string). The option can be 'start' or 'end'.</p> <p>When converting from a higher frequency to a lower frequency, the result does not depend upon the option opt.</p> <p>Returns: date</p>	<pre>%d = 2020q2; PRT %d.date('m', 'start'); //2020m4 PRT %d.date('m', 'end'); //2020m6 PRT %d.date('a', 'start'); //2020 PRT %d.date('a', 'end'); //2020</pre>
date(y, f,	Constructs a new quarterly	<pre>%d = date(2020, 'q',</pre>

sub)	<p>or monthly date from y (integer), frequency (string), and subperiod (integer).</p> <p>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.</p> <p>Returns: date</p>	<pre>2); //2020q2</pre>
fromExcelDate(v)	<p>Converts an Excel date (the val v, counting the number of days since January 1, 1900) to year, month and day (hours etc. are not converted). The year, month and day are returned as a map with the values %y, %m, %d.</p> <p>WARNING: this function will soon return a Gekko date instead. See also toExcelDate(). [New in 3.0.7]</p> <p>Returns: map.</p>	<p>See examples regarding the toExcelDate() function.</p>
getFreq(d)	<p>Extracts the frequency of a date</p> <p>Returns: string</p>	<pre>%d = 2020q2; PRT %d.getfreq(); //'q'</pre>
getMonth(d)	<p>Extracts the month number from a date. More specific than getSubPer(), and will fail if the date is not monthly.</p> <p>Returns: val</p>	<pre>%d = 2020m2; PRT %d.getmonth(); //2</pre>
getQuarter(d)	<p>Extracts the quarter number from a date. More specific than getSubPer(), and will fail if the date is not quarterly.</p> <p>Returns: val</p>	<pre>%d = 2020q2; PRT %d.getquarter(); //2</pre>

getSubPer(d)	Extracts the sub-period from a date (1 if annual or undated, the quarter if quarterly, and the month if monthly). Returns: val	<pre>%d = 2020q2; PRT %d.getsubper(); //2</pre>
getYear(d)	Extracts the year from a date. Returns: val	<pre>%d = 2020q2; PRT %d.getyear(); //2020</pre>
toExcelDate(y, m, d)	Converts year, month and day (integers) into an Excel date (counting the number of days since January 1, 1900). See also fromExcelDate(). Excel dates can be subtracted to obtain days. [New in 3.0.7] Returns: val.	<pre>%v1 = toExcelDate(2019, 11, 12); %v2 = toExcelDate(2019, 12, 3); PRT %v1, %v2; //43781 and 43802 PRT %v2 - %v1; //21 days in between #x = fromExcelDate(%v1 + 100); //100 days from %v1: Feb. 20, 2020. PRT #x.%y, #x.%m, #x.%d;</pre>

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 command 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

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 commands

[STRING](#), [VAL](#), [FOR](#), [IF](#), [TIME](#)

3.19 DECOMP

DECOMP of variables (equations) only works properly on simulated values, where the left-hand sides and the right-hand sides are equal. So for simulated values, or for comparing simulated values, DECOMP is ok. This restriction will be fixed in a patch to Gekko 3.0.

DECOMP opens a special window with an Excel-like sheet showing the contributions etc. The DECOMP command can decompose in two ways:

- **Variable:** an existing model equation can be decomposed, analyzing how the changes in the left-hand side of the equation can be decomposed into contributions from variables on the right-hand side of the equation. The decomposition is carried out on the differences between current and lagged values (time decomposition), or on the differences between the first-position and reference databanks (multiplier decomposition).
- **Expression:** DECOMP can decompose a user-provided expression. This can be thought of as anything legal in a PRT statement (with some limitations).

In the DECOMP window, regarding the list of variables shown in the first column, endogenous (left-hand side) variables are marked in blue. You may click on these to track an effect further (in that case, a new DECOMP window opens). Cells can be copy-pasted to Excel or other spreadsheets (use Ctrl-A til select all cells). If variable labels are present/loaded (cf. [MODEL](#)), these will be shown when the mouse hovers over variables in the first column.

There is an "Update table" button for updating the table if the underlying data changes. For instance, after a new simulation or after a [READ](#) statement.

It should be noted that DECOMP only decomposes an expression into contributions from series or values ([VAL](#)). So in a multiplier decomposition, do not expect Gekko to calculate contributions from matrices or lists of values, if these are different between the first-position and reference databanks (but contributions from VAL scalars will be identified).

Syntax

```
DECOMP < period > variable;
DECOMP < period > expression;
```

<i>period</i>	(Optional). Local period, for instance 2010 2020, 2010q1 2020q4 or %per1 %per2+1.
<i>variable</i>	The name of the endogenous variable to be decomposed.
<i>expression</i>	An expression: anything legal in for instance a PRT statement.

- If no period is given inside the <...> angle brackets, the global period is used (cf. [TIME](#)).

Details

The DECOMP window consists basically of a selector at the top, a table in the middle, and the equation/expression at the bottom. The selector consists of three parts: time-change selector, multiplier selector, and some auxiliary options at the right.

In the time-change and multiplier selectors, you may choose to either see 'raw' (non-decomposed) or 'decomp' (decomposed) values. The raw values are really just tabelling the relevant variables, optionally transforming them via the operators n , d , p , dp for time-changes, or n , m , q , mp for multipliers (see the [PRT](#) command for a list of these so-called 'short' operators). Raw values seldom sum up, so the first row is not usually equal to the sum of the rest of the rows (this only holds for simple sums in levels).

In contrast, the decomposed values for time-change or multiplier will sum up, so that the first row is equal to the sum of the rest of the rows. In that way, you may get an idea of why the left-hand side (or expression) changes relative to the previous period or relative to the reference databank.

In the auxiliary options at the upper right of the window, you may indicate that you want to see values from the reference databank instead of the first-position databank, or that you prefer the decomposition output scaled so that the first line is 100 (%) and the other lines sum up to 100. The number of decimals shown can be changed, and the table can be updated by pressing the 'Update table' button (in case you wish to have changes in the databanks reflected in the table).

The decomposition is done by means of linearizing the equation (for time-changes: in the previous period, and for multiplier changes: for reference databank values) and using this linearization to forecast how much the left-hand side is expected to change due to the changes in the right-hand side ($dy = \beta_1 * dx_1 + \beta_2 * dx_2 + \dots$). This may be more or less precise, depending upon how non-linear the equation is. If there is an error, that is, the contributions do not add up to 100% of the change in the left-hand side, the contributions are adjusted proportionally so that they sum up anyway.

Regarding decomposition of model equations, there is a further source of potential imprecision too, namely if the databank values of the first row (the dependent variable) do not correspond to the equation. If this is so, for instance for historical data, a further proportional adjustment is applied, so that the contributions sum up. (This problem does not exist regarding decomposition of an expression).

Clicking 'Show errors' allows you to inspect possible decomposition and data errors. If, for instance, you have selected 'Abs. time change' (row) and 'Decomp' (column), you may click both 'Show as shares' and 'Show errors' at the same time. This gives a good idea of any decomposition or data errors. If the decomposition error shows for

instance 2.50% for a particular period, this means that only 97.50% of the change in the right hand side (or expression) can be explained by means of the linearization. The smaller the decomposition error is, the more confidence can be put into the decomposed contributions (for linear equations, the decomposition error would be 0 in the absence of rounding errors).

Examples

After performing a multiplier analysis, you may want to decompose an usercost expression like the following:

```
DECOMP <2010 2020> (1-t)*i + b - (1-b)*rpi + 0.2*t;
```

If a model has been loaded with [MODEL](#), and the usercost variable is called `uc`, you may instead use:

```
DECOMP <2010 2020> uc;
```

This will look up the `uc` equation (the equation with `uc` on the left-hand side) and decompose that equation.

Note

[DISP](#) has a similar functionality, allowing to trace variables through model equations. If you need to decompose a long expression, you can mark the lines and hit [Enter] to execute the lines as one block of code. (Or use a command file).

You can only indicate one variable or expression in the DECOMP command. This is to avoid the command potentially opening up a lot of DECOMP windows at the same time.

The 'decimalseparator' option listed below controls how the cells are copied to the clipboard (for pasting in a spreadsheet), when the user uses copy-paste of cells in the DECOMP window.

Related options

OPTION interface excel decimalseparator = period; [period, comma]

Related commands

[READ](#), [CLONE](#), [MULPRT](#), [DISP](#), [CUT](#)

3.20 DELETE

DELETE is used to remove variables from 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 \bar{y} for income. Now, imagine that the definition and contents of the variable is changed to \bar{y}_2 in both the databank and model. If the old variable \bar{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 command, so you may for instance use the shorter `a*b` instead of `{'a*b'}`.

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 commands

[CREATE](#), [SERIES](#), [RESTART](#), [RESET](#)

3.21 DISP

The command 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.

If a model is loaded, the DISP command 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.

When displaying an array-series, the dimensions, possible domains, etc. are shown.

DISP of other variable types than series works like [PRINT](#).

If you have loaded a GAMS model with MODEL<gms>, you must set `OPTION model type = gams` to DISP the equations properly. For GAMS models, there are special options regarding how to identify which variable a given equation determines.

Syntax

```
DISP < period INFO > variables ;
DISP 'search string' ;
```

<i>period</i>	(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 <code>OPTION solve failsafe = yes</code> .
<i>variables</i>	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 command, so you may for instance use the shorter `a*b` instead of `{'a*b'}`.

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 command 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>` command 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.

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 commands

[PRT](#), [MULPRT](#), [PLOT](#), [DECOMP](#), [TELL](#)

3.22 DOC

The command is used to 'manually' change meta information fields in a timeseries.

The meta information is shown in the [DISP](#) command.

Syntax

```
DOC variables LABEL=... SOURCE=... UNITS=... STAMP=... ;
DOC <browser>;
```

<i>variables</i>	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.

- If a variable is stated without databank, the databank is assumed to be the first-position databank.

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. This inner workings of this system will be documented later on, if needed before then, please contact the Gekko editor. Essentially 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='11-01-2015';
```

To clear the label, use an empty string:

```
DOC fy label='';
```

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 commands

[READ](#), [IMPORT](#), [WRITE](#), [EXPORT](#), [DISP](#)

3.23 DOWNLOAD

At the moment, the command 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 command regarding other online databanks. Note that you can import a px file with [IMPORT](#)<px> or IMPORT<px array>.

The data is downloaded into the first-position databank.

Syntax

DOWNLOAD < **ARRAY** > *url* *filename* **DUMP=...**;

<i>ARRAY</i>	(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.
<i>url</i>	Url (web address) to the databank. Note: the web address should be in quotes.
<i>filename</i>	Filename of the JSON file defining what data to download.
<i>DUMP=</i>	(Optional). Name of the file in which to store the contents of the download (in this case, a px-file).

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` command, these two timeseries are available in the first-position (Work) databank. The above procedure 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> data;
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]`).

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. [New in 3.0.3].
- **CODES(...)**. Gets dimension names and dimension elements from this (*), for use in the timeseries names. For instance, the name part "brancheDB07_BC", where the first is the dimension name, and the last is the dimension element.
- **VALUES(...)**. Only used for metadata in the timeseries (timeseries labels).
- **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!

(*) If there is a .json file involved, the often shorter dimension names from the .json are used instead.

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).

Note

For more advanced px reading, you may take a look at the [pxr](#) package in [R](#).

Related commands

[IMPORT](#), [READ](#), [OPEN](#)

3.24 EDIT

The EDIT command uses Notepad to open up the designated file. The command is practical for editing command 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 command files, cf. `OPTION interface remote = yes|no;`.

Syntax

EDIT filename ;

filename

Filenames may contain an absolute path like `c:\projects\gekko\myfile`, a relative path like `\gekko\myfile.gbk`, or be stated without a path. Filenames containing blanks and special characters should be put inside quotes. 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 forecast1.gcm from the working folder:

```
EDIT forecast1;
```

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.

Related options

[OPTION](#) interface remote = no; [yes|no]

Related commands

[SYS](#), [XEDIT](#)

3.25 ELSE

An [ELSE](#) statement is used in conjunction with an [IF](#) condition and an [END](#) statement.

Related commands

[IF](#), [ELSE](#)

3.26 END

An END statement concludes a [FOR](#) (loop), [IF](#) (condition) or [FUNCTION/PROCEDURE](#) statement. Gekko will fail if the END statement is missing.

To execute for instance a loop in the command 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 command file.

Related commands

[FOR](#), [IF](#), [FUNCTION](#), [PROCEDURE](#)

3.27 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 command works differently depending upon `OPTION model type`.

- 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>, ... ;
```

<i>variables</i>	Default type: The variables are simple series names, or lists of these, for instance <code>x2</code> , or <code>{#m}</code> . Gams type: The variables are series or array-series names, for instance <code>x2</code> or <code>x2[a, b]</code> . For array-series, lists may be used, for instance <code>x2[a, #i, #j]</code> , where <code>#i</code> and <code>#j</code> are lists of strings.
<i>period0</i>	A period is a Gekko time interval like <code><2020 2030></code> or <code><2020q1 2030q4></code> . The general period can be set in the <code>period0</code> field, and this period will be used for the variables, unless specific periods are given in the <code>period1</code> , <code>period2</code> , etc.
<i>period1</i> , <i>period2</i> , ...	A period is a Gekko time interval like <code><2020 2030></code> or <code><2020q1 2030q4></code> . These specific periods will overrule the general period (<code>period0</code>).

Examples

Default type

If you need to exogenize a variable `fy`, and endogenize a variable `tg`, use this:

```

OPTION model type = default; //is default
MODEL forecast; //a model must be loaded beforehand
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;
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	1.0000	M	1.0000
	1.0000			
2024	1.0000	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;
#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 $2 \times 2 = 4$ elements (subseries) regarding `exo_x1` and `endo_x2`:

	exo_x1[a1, k1]	exo_x1[a1, k2]	exo_x1[a2, k1]
2020	exo_x1[a2, k2]		
	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]
2020	endo_x2[a2, k2]		
	M	M	M
2021	M	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 default type, the ENDO and EXO statements are non-cumulative, so all endogenized/exogenized variables should be present in the same ENDO/EXO statement.

With gams type, 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 commands

[EXO](#), [SIM](#), [UNFIX](#)

3.28 EXIT

The command 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 commands

[STOP](#), [RETURN](#)

3.29 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 command works differently depending upon `OPTION model type`.

- 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](#) command.

Related options

`OPTION model type = default; //default | gams`

Related commands

[ENDO](#), [SIM](#), [UNFIX](#)

3.30 EXPORT

The command 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 command files.

Excel note: When constructing 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 set, 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 ;
```

<i>period</i>	(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.
<i>format</i>	File format. Choose between CSV, FLAT, GCM, GDX, GNUPLOT, PRN, R, TSD, TSP, XLS/XLSX (regarding gbk, see the WRITE command). <ul style="list-style-type: none"> • 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

	<p>timeseries into the file data.gcm. Alternatively, you may use <code>^=</code>, <code>%=</code>, <code>+=</code> or <code>*=</code> operators, for instance <code>EXPORT<gcm op='%= '> {#vars} file=data;</code>. With the latter operators, you must enclose them in single quotes ('). You may use <code>EXPORT<gcm></code> to export in levels (corresponding to operator <code>n</code>). A .gcm file is imported simply with RUN. See the FLAT format for a faster version of this format.</p> <ul style="list-style-type: none"> • 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. • GNUPLLOT: 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. • R: Exports matrices as a R script file. The syntax is a bit convoluted, since matrices and not series are exported, and the <code>EXPORT<r></code> syntax is expected to change at some point. To export several matrices in one go, you need to state the matrix names as list items, for instance like this: <code>#m1 = [1, 2; 3, 4]; #m2 = [11, 12; 13, 14]; #matrices = ('#m1', '#m2'); EXPORT<r> {#matrices} file=matrix.r;</code> Exporting a single matrix is more simple: <code>#m = [1, 2; 3, 4]; EXPORT<r> #m file=matrix.r;</code> 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). • 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 command. 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 <code><tsd caps=no></code> .

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.
DATEFOR MAT= DATETYPE =	(Optional). These options control the date format for .xlsx and .csv files. <code>DATEFORMAT</code> can be either 'gekko' (default) or a format string like 'yyyy-mm-dd', and the latter may contain a <i>first</i> or <i>last</i> indicator, for instance 'yyyy-mm-dd last', which indicates for quarterly or monthly data that the <i>last</i> day of the quarter or month is used. <code>DATETYPE</code> 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. [New in 3.0.5].
<i>variables</i>	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 <code>EXPORT <csv> x* to *_old file = test;</code> , where Gekko will look for variables starting with <code>x</code> , and the found variables will acquire a <code>_old</code> suffix. This logic is similar to the <code>COPY</code> and <code>RENAME</code> commands.
<i>filename</i>	Filenames may be contain an absolute path like <code>c:\projects\gekko\myfile</code> , a relative path like <code>\gekko\myfile.gbk</code> , or be stated without a path. Filenames containing blanks and special characters should be put inside quotes. 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 command 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.

Related options

[OPTION](#) `folder bank = [empty];`

[OPTION](#) `interface csv decimalseparator = period; [period|comma]`

Related commands

[IMPORT](#), [READ](#), [WRITE](#), [SHEET](#)

3.31 FINDMISSINGDATA

This command finds timeseries with missing values (only the timeseries with frequency matching the global frequency setting).

Please note that the command 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 ;

<i>period</i>	(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.
<i>variables</i>	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 command

```
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 command is convenient when developing new models or changing existing models.

Related commands

[COMPARE](#), [DELETE](#), see also 'Utilities' --> 'Compare two databanks' (same as COMPARE)

3.32 FOR

The FOR command 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.

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 <code>a, b, c</code> instead of <code>('a', 'b', 'c')</code> , similar to how a list may be defined using short form. You may also use for instance lists (<code>#mylist</code>) or wildcards (e.g. <code>fx*</code>). You may also use a list of values, for instance <code>(1, 2, 3)</code> , or a list of dates, for instance <code>(2001q1, 2001q2, 2001q3)</code> .
Note that you may use parentheses, for instance <code>FOR([type] %x = items)</code> , like the IF command.	

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 command.

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 <code>BY step</code> if not needed. You may use STEP instead of BY if preferred.
TO	You may use '..' (range) instead: for instance <code>FOR date %d = 2015q1 .. 2020q4;</code>
Note that you may use parentheses <code>FOR (date %d = date1 TO date2 BY step)</code> , like the IF command. If one or both of <code>date1</code> and <code>date2</code> are positive integers, they will be interpreted as annual dates.	

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 ;
```

<i>v</i>	The loop variable %v.
<i>val1</i>	Start value. Can be any number or expression.
<i>val2</i>	End value. Can be any number or expression.
<i>step</i>	An optional stepsize (default step: 1). Can be any number or expression, and may be negative. Omit <code>BY step</code> if not needed. You may use <code>STEP</code> instead of <code>BY</code> if preferred.
<code>TO</code>	You may use <code>'..'</code> (range) instead: for instance <code>FOR val %v = 1 .. 100;</code>
Note that you may use parentheses <code>FOR(val %v = val1 TO val2 BY step)</code> , like the <code>IF</code> command.	

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](#) command. 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  
    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 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;
```

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 result is the following:

```
a, x  
b, y  
c, z
```

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 (this code 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;
```

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]);
    SERIES y[%d] = x0[%d];
  ELSE;
    SERIES 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};
  SERIES 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`.

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 command (which has mandatory parentheses).

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 commands

[END](#), [STRING](#), [DATE](#), [VAL](#), [IF](#)

3.33 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, you may consider using a [procedure](#) instead. A procedure is essentially the same as a user functions 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.

How to use a library of Gekko functions/procedures in 3.0?

In Gekko 3.0, the `OPTION library file = ...;` is obsolete. Instead, you can just put your user-defined functions/procedures in for instance a file called `lib.gcm`. Afterwards, you can define a [gekko.ini](#) file containing the line `RUN lib.gcm;` so that `lib.gcm` is always run at Gekko startup, or after a [RESTART](#). See the `lib.gcm` example on the [RESTART](#) help page. In Gekko 3.0, user functions/procedures are always available after they have been defined, as long as the use is chronologically after the definition.

Function hints

If a function has syntax errors, you may try to out-comment the FUNCTION statement and corresponding END statement for better error messages. Function arguments do not reside in any databanks, so if you have a function like `FUNCTION void f(series x); RUN data.gcm; END;` you cannot expect to use `x` inside the `data.gcm` command file, for instance expecting it to reside in the first-position databank (regarding function arguments, in many cases using the `name` type is more practical than the `series` type).

Syntax

```
FUNCTION type funcname(<date t1, date t2>, type1 var1 label1 = default1,
type2 var2 label2 = default2, ...);
  expressions... ;
END;
```

The function body must contain at least one [RETURN](#) statement, returning a variable.

<i>t1, t2</i>	(Optional). You may state optional time period parameters inside <code><></code> -brackets, for instance <code>FUNCTION series f(<date %t1, date %t2>, series x);</code> after which <code>%t1</code> and <code>%t2</code> are assigned to for instance 2020 and 2030 in the call <code>f(<2020 2030>, z)</code> . If the function is called without <code><></code> -brackets, for instance <code>f(z)</code> , the parameters <code>%t1</code> and <code>%t2</code> are assigned to the local/global time period instead. Using a <code><></code> -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.
<i>type1, ...</i>	Types of incoming and outgoing variables: <code>series</code> , <code>val</code> , <code>date</code> , <code>string</code> , <code>list</code> , <code>map</code> , <code>matrix</code> . You may also use the special name <code>type</code> for parameters, which behaves 100% as a <code>string</code> inside the function, but where the single quotes are omitted when calling the function from outside (the shorter call <code>f(y)</code> is used instead of <code>f('y')</code>). If the function does not return anything, use <code>void</code> as type.
<i>var1, ...</i>	The parameters/variables/expressions
<i>label1, ...</i>	(Optional). A label for the parameter, used if the function is prompting (called with <code>f?(...)</code>). See more about prompting below.
<i>default1, . ..</i>	(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 <code>f?(...)</code>) and the parameter is omitted, the default value is shown in the dialog box. In the dialog box, <code>Enter</code> or <code>Escape</code> will return the default value, and fire up the next dialog box (for the next optional parameter). If a <code>;</code> 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 (<code>'</code>) in the input box is optional. At the moment, only <code>val</code> , <code>date</code> and <code>string</code> types can be used for prompting input boxes.
<i>funcname</i>	The function name
<i>body</i>	The function body, that is, the commands to be performed. Use <code>RETURN</code> 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 particular line, try inserting a line with a non-existing function like for instance `stop();`. This will abort the program in a clean way and make it possible to inspect variables 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; SERIES 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.

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;
  SERIES <%d1 %d2> {%n} = %v;
  RETURN;
END;
//-----
load('extral', 'Helper variable', 1980, 2020, 100);
load('vat', 'VAT rate', 1980, 2020, 0.25);
disp extral, 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:
//
//      2001          z1          z2
//      2002          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.

Prompt 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 options

[OPTION](#) library file = [filename];

Related commands

[PROCEDURE](#), [RUN](#)

3.34 GLOBAL

The GLOBAL command 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 command 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](#) command regarding different types of databanks in Gekko.

See also the similar [LOCAL](#) command, for local variables.

Syntax

```
GLOBAL varnames;
GLOBAL <all>;
```

varname s	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 <code>x</code> that you would like to keep in another databank despite using a <code>GLOBAL<all></code> , you may use <code>first:x</code> or another bank designation to circumvent <code>GLOBAL<all></code> .

Examples

```
GLOBAL x, %y, #z;
```

After this, any use of `x`, `%y`, or `#z` (in the present command 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](#) commands, 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 commands

[LOCAL](#)

3.35 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 command 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 accidentally 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 superfluous, 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 commands

[TARGET](#)

3.36 HDG

HDG (heading) will put the heading into a databank file. This only works for .gbk files.

Syntax

HDG *heading* ;

<i>heading</i>	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 commands

[WRITE](#), [READ](#)

3.37 HELP

The HELP command (or F1) provides access to Gekko help system. Through the HELP command it is possible to get quick help on a command 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 `command`;

command

(Optional). The command on which help is needed. If the command 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 command, you may look directly for this topic in the help file:

```
HELP series;
```

If you cannot remember that the name of the relevant command is SERIES (for instance), you may write

```
HELP;
```

In the section "Gekko commands", there is a page called "Command overview" where the commands 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 commands

[OPTION](#)

3.38 IF

The IF command is used for conditional execution of different blocks of statements. The IF statement works with [strings](#), [dates](#), and [values](#) (or for instance single timeseries observations like `x[2010]`).

You may sometimes need to explicitly convert the variables in order to compare them (by means of the functions `val()`, `date()` or `string()`).

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.

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.

Syntax

```
IF ( expression ) ;
    statements1... ;
ELSE ;
    statements2... ;
END ;
```

<i>expression</i>	<p>A logical expression involving strings, dates or values (or single timeseries observations like <code>x[2010]</code>), in addition to the logical operators AND, OR, NOT, <code><</code>, <code><=</code>, <code>==</code>, <code>>=</code>, <code>></code>, <code><></code>. Please note that logical equivalence uses <code>==</code> (and not <code>=</code> which is assignment) , and that you must use <code><></code> for logical difference, not for instance <code>!=</code>.</p> <p>If the expression is a scalar value, the statements are not executed if the scalar value is 0, and are executed otherwise.</p>
<i>statements1</i>	Gekko commands to be executed if <i>expression</i> is true.
<i>statements2</i>	(Optional). Gekko commands to be executed if <i>expression</i> is 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 choose between more choices (and test validity of %write), you may use:

```
%write = 'yes';
IF (%write == 'yes');
    TELL 'Yes chosen';
ELSE; IF (%write == 'no');
    TELL 'No chosen';
ELSE; IF (%write == 'maybe');
    TELL 'Maybe chosen';
ELSE;
    TELL 'ERROR: The scalar variable should be yes, no or maybe';
END; END; END;
```

This is a little awkward with the three ending ENDS (a 'real' ELSEIF statement will be provided later on to provide easier syntax for several cases).

```
%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 much simpler way, using the `$`-conditional.

```
SERIES y = 2 $ (x[2015] == 100);
```

Note

- Note the use of two equal signs (`==`) in IF-commands, 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 `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 commands

[FOR](#), [END](#), [STRING](#), [DATE](#), [VAL](#)

3.39 IMPORT

The IMPORT command 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 monthly, quarterly or annual 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 command 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. 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.

Syntax

```
IMPORT < period format ALL ARRAY COLS REF SHEET=... CELL=...
NAMECELL=... DATECELL=... COLLAPSE=... METHOD=... DATEFORMAT=...
DATETYPE=... > 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 CSV, FLAT; GDX, PCIM, PRN, PX, TSD, TSP, XLS, XLSX.

- 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).
- 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 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](#) command.
- TSD: For interchange with AREMOS and others.
- TSP: Imports [TSP](#) records.
- XLS and XLSX: Tabular format with rows/cols consisting of names and dates. If you need to pick out Excel data from particular cells, see [SHEET](#)<import>. 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 = ...;`.

<i>filename</i>	<p>Filenames may contain an absolute path like <code>c:\projects\gekko\myfile</code>, a relative path like <code>\gekko\myfile.gbk</code>, or be stated without a path. Filenames containing blanks and special characters should be put inside quotes. See more on filenames here.</p> <p>If the filename is set to '*', you will be asked to choose the file in Windows Explorer.</p> <p>The extension <code>.gbk</code> is automatically added, if it is missing.</p>
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 <code>.csv</code> or Excel files, this indicates whether the timeseries are running downwards in columns. Note that for <code>.prn</code> files, you indicate this in the first 'cell' (date/name).
TO	<p>(Optional). If "TO <i>bankname</i>" is indicated, Gekko will put the data into a separate 'named' databank alongside the Work and Ref databanks. For instance, after <code>IMPORT <xlsx> adambk TO a;</code>, you may refer to the variables by means of colon, for instance <code>PRT a:var1;</code>. If you use <code>IMPORT <xlsx> adambk TO *;</code>, 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 command)</p>
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).
CELL=	(Optional). For Excel files: the first cell of the data section. Defaults to 'B2'.
DATECELL=	(Optional). For Excel files: the first cell of the dates labels. Calculated from CELL location if not provided.
NAMECELL= =	(Optional). For Excel files: the first cell of the names labels. Calculated from CELL location if not provided.
COLLAPSE=	(Optional). For Excel files with Excel-dates that are going to be collapsed, this option can be set to either m, q or a and indicates the frequency that the data points are being collapsed into. A data

	<p>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.</p>
METHOD=	<p>(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 command. 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.</p>
DATEFORM AT= DATETYPE=	<p>(Optional). These options control the date format for .xlsx and .csv files. DATEFORMAT can be either 'gekko' (default) 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 or monthly data that the last day of the quarter or month 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. [New in 3.0.5].</p>

- 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>](#) command.

Example, collapse

Excel data may be collapsed from higher frequencies than months (for instance from daily or weekly 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](#) command on the resulting monthly timeseries).

If needed, the [X12A](#) command 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!

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.

Related options

[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](#) sheet engine = internal; //use 'excel' for the older .xls format

Related commands

[READ](#), [WRITE](#), [EXPORT](#), [OPEN](#), [CLONE](#), [DOWNLOAD](#), [COLLAPSE](#)

3.40 INDEX

The command is used to search for variables in databanks, using wildcards or ranges. The result of the search may be put into a list.

Note that 'naked' [wildcards](#) are allowed in this command, so you may for instance use the shorter `a*b` instead of `{'a*b'}`.

A wildcard like `*` does not match everything in Gekko: it only matches (in the first-position databank) variables with no `%` and `#` type symbols, and only matches the current frequency. You may use the special wildcard `**` to match all variables in a databank, or `***` to match all variables in all databanks.

Beware: 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., is explained more generally on the [wildcards](#) page.

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 command is essentially an <code>INDEX<mute></code> .
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.

<i>wildcard</i>	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 <code>a*x</code> to find all variables starting with 'a' and ending with 'x', or <code>a?x</code> to match only one character.
<i>listname</i>	(Optional). The list name where the result is stored. The listname may be for instance <code>#m</code> , or <code>#(listfile m)</code> . 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 command 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 command 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 command types, for instance `#m = *:f*` or `PRT *:f*` would fail with an error. Do not expect "naked" wildcards to work in commands that accept mathematical expressions.
- The INDEX command 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'.

Note

If you use variable names without wildcards or ranges, an 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 commands

[LIST](#), [COUNT](#)

3.41 INI

The INI command 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](#) command is in reality a [RESET](#) command followed by an INI command. 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 command 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 commands

[RESTART](#), [RESET](#)

3.42 INTERPOLATE

INTERPOLATE transforms one lower-frequency timeseries to a higher-frequency timeseries, for instance converting annual data to quarterly data. Use [COLLAPSE](#) to perform the inverse transformation.

Syntax

```
INTERPOLATE vars1 = vars2 method;
```

<i>vars1</i>	Higher frequency timeseries. Frequency can be indicated with suffix <code>!a</code> , <code>!q</code> or <code>!m</code> . Banknames may be used. Lists can be use like for instance <code>{#m}</code> .
<i>vars2</i>	Lower-frequency timeseries. Frequency can be indicated with suffix <code>!a</code> , <code>!q</code> or <code>!m</code> . Banknames may be used. Lists can be use like for instance <code>{#m}</code>
<i>method</i>	<p>(Optional). Choose between:</p> <ul style="list-style-type: none"> <code>repeat</code>: Repeats the lower-freq observation (<code>rep</code> may be used as synonym) <code>prorate</code>: Also repeats, but divides the result so that the sum of the higher-freq observations corresponds to the lower-freq observation. <p>Note: default is <code>repeat</code>. More methods may be added by popular demand.</p>

- 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](#)).

Example

Use this to convert frequency:

```
INTERPOLATE x!q = x!a;
```

Since the method is `repeat` as default, this will create the quarterly timeseries `x` where each quarterly observation in `x!q` is the same as the corresponding annual observation in `x!a`.

```
INTERPOLATE qbank:x!q = abank:x!a prorate;
```

With option `prorate`, the quarters will sum up to `x!a` instead of just being repeated.

Note

If a frequency indicator is omitted, Gekko will use the current frequency.

More to come by popular demand, for instance using patterns, splines, etc., to create the high-frequency series.

Related commands

[COLLAPSE](#), [SERIES](#), [CREATE](#), [PRT](#)

3.43 ITERSHOW

The command ITERSHOW show details of previous Gauss-Seidel iterations (cf. [SIM](#)).

Syntax

```
ITERSHOW < period > variables;
```

<i>period</i>	(Optional). Local period, for instance 2010 2020, 2010q1 2020q4 or %per1 %per2+1.
<i>variables</i>	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 command 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 command 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 commands

[SIM](#)

3.44 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. See examples.

Upgrade note

In Gekko 2.x, a list could be stated like for instance `LIST m = #m1, a, #m2, b;`. Lists could only contain strings, so the command meant taking the strings from `#m1`, adding the string `'a'`, adding the strings from `#m2`, and finally adding the string `'b'`. In Gekko 3.0, a list can be added to another list in two ways: either adding the list itself (creating a nested list), or adding the list elements one by one (like Gekko 2.x). The former operation is called `append()` in Gekko 3.0, and the latter operation is called `extend()`, so the statement would be translated into `#m = #m1.append('a').extend(#m2).append('b');`. Instead of this, you may use the `+` operator, so `#m = #m1 + ('a',) + #m2 + ('b',);` will work, too. Note the use of `('a',)`, which is a single-item list, where the comma cannot be omitted (alternatively: use `list('a')`). Note also that `#m = (#m1, 'a', #m2, 'b');` is different in 3.0, creating a nested list.

These changes may seem cumbersome, but Gekko lists are much more powerful in the 3.0 version, and the syntax changes are necessary to support that. To alleviate some of the syntax burden, so-called naked lists like `#m = a, b, c;` are allowed in 3.0, cf. below. Using a naked list, the above example can be written as `#m = {#m1}, a, {#m2}, b;`. This is perhaps the easiest way to concatenate lists and strings like in Gekko 2.x.

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 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[value]           //picks out an element (by number 1, 2, 3, etc.)
#m[value..value]    //range/slice, returns a list.
#m[... , ...]       //matrix-like selections, using comma
{#m}                //use braces {...} to refer to variables
corresponding to string elements
```

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['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)

Use `#x.length()` or `length(#x)` to get the number of elements in `#x`, `#x[0]` cannot be used for this anymore. Use `#x.contains('a')` to check whether `'a'` is a member of `#x`. In **IF** statements, you can use `IF(#x.contains('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.

The LIST keyword is optional and can typically be omitted. Use REP to repeat items. For the last item, you may use `REP *` which has special capabilities in relation to timeseries. 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 <code>#m[3..3]</code> . Returns: var	<code>#m[3]; //the third element</code>
[x1..x2]-index	Index: picks out a range of elements. You may omit x1 or x2. Returns: list	<code>#m[3..5]; //the third to fifth elements</code>
[x1, x2]-index	For a nested list of lists, <code>#m[3, 5]</code> will return the same element as <code>#m[3][5]</code> , so this is just convenience to make a nested list accessible like a matrix . See more here . Returns: variable [New in 3.0.6].	<code>#m = ((1, 2), (3, 4)); PRT #m[2, 1], #m[2] [1]; //same</code>
[x1..y1, x2..y2]-index [x1..y1, x2]-index [x1, x2..y2]-index	For a nested list of lists, <code>#m[2..3, 2..4]</code> will select the given "rows" and "columns", corresponding to selecting a submatrix from a matrix . Beware that in general, <code>#m[2..3, 2..4]</code> is completely different from	<code>// 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-</code>

	<p><code>#m[2..3][2..4]</code>. See more here.</p> <p>Returns: list [New in 3.0.6].</p>	<pre>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!</pre>
<p><code>append(x1, x2)</code> <code>append(x1, i, x2)</code></p>	<p>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 <code>extend()</code>.</p> <p>If used with i argument, x2 is inserted at index i, instead of at the end. See also <code>extend()</code>.</p> <p>To prepend, use <code>append(x1, 1, x2)</code>.</p> <p>Returns: list</p>	<pre>#y = #x1.append(#x2); //or: append(#x1, #x2) #y = #x1.append(2, #x2); //insert at position 2</pre>
<p><code>contains(x1, x2)</code></p>	<p>Checks if the list of strings x1 contains the string x2. Returns 1 if true, 0 otherwise. You may alternatively use <code>x2 in x1</code>, see the last example. See also the <code>count()</code> and <code>index()</code> functions. The comparisons are case-insensitive.</p> <p>Returns: val</p>	<pre>%v = #x1.contains(%s); if(#x1.contains(%s) == 1); tell 'yes'; end; if(%s in #x1); tell 'yes'; end;</pre>
<p><code>count(x1, x2)</code></p>	<p>Counts the number of times the string x2 is present in the list of strings x1. See also the <code>contains()</code> and <code>index()</code> functions.</p> <p>Note: to obtain the number of elements in a list, use</p>	<pre>%v = #x1.count(%s); //or: count(#x1, %s)</pre>

	<p>the length() function. The comparisons are case-insensitive.</p> <p>Returns: val</p>	
data(x)	<p>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.</p> <p>Returns: list</p>	<pre>#m = data('1.0 2.0 1.5');</pre>
dates(x)	<p>Tries to convert each element of the list x to a date.</p> <p>Returns: list</p>	<pre>#y = dates(#x);</pre>
except(x1, x2)	<p>The except() function subtracts x2 from x1. You may alternatively use the operator -. Only works for lists of strings. See also intersect() and union().</p> <p>Was called difference() in Gekko 2.0. See also extend().</p> <p>Returns: list</p>	<pre>#y = #x1.except(#x2); //or: except(#x1, #x2) #y = #x1 - #x2; //same #y -= #x1; //subtract from itself</pre>
extend(x1, x2) extend(x1, i, x2)	<p>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.</p> <p>For two lists x1 and x2, you may alternatively use the + operator. See also except() and append().</p> <p>To pre-extend, use extend(x1, 1, x2).</p>	<pre>#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</pre>

	Returns: list	
flatten(x)	<p>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).</p> <p>Returns: list</p>	<pre>#m1 = (1, (2, 3)); #m2 = #m1.flatten(); //or: flatten(#m1).</pre>
index(x1, x2)	<p>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.</p> <p>Returns: val</p>	<pre>%i = #x1.index(%s); //or: index(#x1, %s)</pre>
intersect(x1, x2)	<p>The intersect() function finds the common elements of the two list of strings x1 and x2. You may alternatively use the operator &&. Only works for lists of strings. See also except() and union().</p> <p>Returns: list</p>	<pre>#y = #x1.intersect(#x2); //or: intersect(#x1, #x2) #y = #x1 && #x2;</pre>
length(x)	<p>Returns the number of elements in the list x. You may use len() instead of length().</p> <p>Returns: val</p>	<pre>%v = #x.length(); //or: length(#x). %v = #x.len(); //the same</pre>
list(x1, x2, ...)	<p>Returns a list of the variables x1, x2, etc. The function is handy for lists with only 0 or 1 elements. See examples.</p> <p>Returns: list</p>	<pre>#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</pre>

lower(x)	Returns string elements in the list as lower-case. Returns: list	<pre>#y = #x1.lower(); //or: lower(#x1)</pre>
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	<pre>#y = #x1.pop(2); //or: pop(#x1, 2) #y = #x1.pop(); //last element #y = #x1.pop(1); //first element</pre>
preextend(x1, x2)	Same as extend(x1, 1, x2), putting the elements of x2 in the first position of x1.	<pre>#y = #x1.preextend(#x2); //insert at position 1</pre>
prefix(x1, x2)	If x1 is a list of strings, each element has the string x2 prefixed (prepended) Returns: list	<pre>#y = #x1.prefix(%s); //or: prefix(#x1, %s);</pre>
prepend(x1, x2)	Same as append(x1, 1, x2), putting x2 in the first position of x1.	<pre>#y = #x1.prepend(#x2); //insert at position 1</pre>
sort(x)	Returns a sorted list of strings, provided that x is a list of strings. Sorting is case-insensitive. Returns: list	<pre>#y = #x.sort(); //or: sort(#x)</pre>
remove(x1, x2)	Removes any string x2 from the list of strings x1. See also the except() function. Returns: list	<pre>#y = #x1.remove(%s); //or: remove(#x1, %s);</pre>
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 occurrences of x2 inside the string replaced with x3. The replacements may be limited via the max argument.	<pre>#y = #x1.replace(%x2, %x3); //or: replace(#x1, %x2, %x3) #y = #x1.replaceinside(%x2, %x3); //or: replace(#x1, %x2, %x3, 'inside')</pre>

	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	<code>#y = strings(#x);</code>
suffix(x1, x2)	If x1 is a list of strings, each element has the string x2 suffixed (appended) Returns: list	<code>#y = #x1.suffix(%s); //or: suffix(#x1, %s);</code>
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)	<code>#m = ((1, 2), (3, 4)); p #m, t(#m);</code>
union(x1, x2)	The union() function adds the two lists (only adds unique elements in x2 that are not in x1), or you may use the operator <code> </code> . Alternatively, you may use <code>x + y</code> , but that may introduce dublets. Only works for lists of strings. See also except() and intersect(). Returns: list	<code>#y = #x1.union(#x2); //or: union(#x1, #x2) #y = #x1 #x2;</code>
unique(x1)	Retains only those elements of list x1 that are unique (list of strings only). Returns: list	<code>#y = #x1.unique(); //or: unique(#x1)</code>
upper(x)	Returns string elements in the list as upper-case. Returns: list	<code>#y = #x1.upper(); //or: upper(#x1)</code>
vals(x)	Tries to convert each element of the list x to a	<code>#y = vals(#x);</code>

	value Returns: list	
--	------------------------	--

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
```

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`. [New in 3.0.3].

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;                               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,
abcd                                     //result: 'abd', 'abcd'
#m2 = #m1['a*d'];                         //result: 'abd'
#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                    //result: 'abd'
#m3 = ['a?d'];
```

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); //array-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:

i, x[#i]	sum(#i, x[#i])			
(#i in #i.remove('b'))	x[a]	x[b]	x[c]	um(#i, x[#i])
2020	1.0000	2.0000	3.0000	6.0000
2021	1.0000	2.0000	3.0000	6.0000
2022	1.0000	2.0000	3.0000	6.0000

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](#) command. 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.

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 commands

[MAP](#), [MATRIX](#), [FOR](#)

3.45 LOCAL

The LOCAL command 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 command 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](#) command for global variables, and the [BLOCK](#) structure for temporary settings.

Syntax

```
LOCAL varnames;  
LOCAL <all>;
```

varname s	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 be practical for functions/procedures where no temporary variables are supposed to exist after the function/procedure has been left. For a variable <code>x</code> that you would like to keep despite using a <code>LOCAL<all></code> , you may use <code>first:x</code> or another bank designation to circumvent <code>LOCAL<all></code> .

Example

```
LOCAL x, %y, #z;
```

After this, any use of `x`, `%y`, or `#z` (in the present command file, function or procedure) will be interpreted as `local:x`, `local:%y`, and `local:#z`, respectively. And after Gekko leaves the current context (command 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:%perl = 2010;` first, and referring to `local:%perl` later on is possible, too. In that sense, the `LOCAL` keyword is just for convenience, especially if `%perl` is used several times.

Local variables survive [READ](#), [CLEAR](#), etc., but do only live in their local context (command 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 commands

[GLOBAL](#)

3.46 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](#) command.

Syntax

```
LOCK databank;
```

Examples

```
LOCK mybank;
```

This locks `mybank` (sets it non-editable, provided that it is unlocked already).

Related commands

[UNLOCK](#), [OPEN](#)

3.47 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 elements 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 command 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 coincidental: 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 commands

[LIST](#)

3.48 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.

NOTE: Instead of the former SHOW command 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.
<i>name</i>	The name of the matrix.
<i>expression</i>	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](#) command (SHEET<import matrix>). To export a matrix `#m` from Gekko to Excel, you can use `EXPORT <xlsx> #m file = matrix.xlsx;`

Examples

The MATRIX command 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 definition. 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 printing 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 $n \times 1$ 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 3×2 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, bull;
SERIES <1998 2010> lna1 = 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') ;
SERIES <1998 2010> pcp = 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') ;
SERIES <1998 2010> bull = 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 = bull;
s4 = bull.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 commands 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. Returns: matrix	<code>#m2 = avgc(#m1);</code>
avgr(x)	Average over rows Returns: matrix	<code>#m2 = avgr(#m1);</code>
chol(x) chol(x, type)	Cholesky decomposition of matrix x. Accepts type (string), either 'upper' or 'lower'. Returns: matrix	<code>#m2 = chol(#m1, 'upper');</code>

<code>cols(x)</code>	Returns the number of columns of <code>x</code> Returns: val	<code>%v = cols (#m);</code>
<code>det(x)</code>	Determinant of a matrix. Returns: val	<code>%v = det (#m);</code>
<code>diag(x)</code>	Diagonal. If <code>x</code> is a <code>n x n</code> symmetric matrix, the method returns the diagonal as a <code>n x 1</code> matrix. If <code>x</code> is a <code>n x 1</code> column vector, the method returns a <code>n x n</code> matrix with this column vector on the diagonal (and zeroes elsewhere). Returns: matrix	<code>#m2 = diag (#m1);</code>
<code>divide(x1, x2)</code>	Element by element division of the two matrices. If <code>x2</code> is a row vector, each <code>x1</code> column will be divided with the corresponding value from the row vector. And if <code>x2</code> is a column vector, each <code>x1</code> row will be divided with the corresponding value from the column vector. Returns: matrix	<code>#x = divide (#x1, #x2);</code>
<code>i(n)</code>	Returns a <code>n x n</code> identity matrix. Returns: matrix	<code>#m = i (10);</code>
<code>inv(x)</code>	Inverse of matrix <code>x</code> Returns: matrix	<code>#m2 = inv (#m1);</code>
<code>maxc(x)</code>	Max over cols Returns: matrix	<code>#m2 = maxc (#m1);</code>
<code>maxr(x)</code>	Max over rows Returns: matrix	<code>#m2 = maxr (#m1);</code>
<code>minc(x)</code>	Min over cols Returns: matrix	<code>#m2 = minc (#m1);</code>

<code>minr(x)</code>	Min over rows Returns: matrix	<code>#m2 = minr(#m1);</code>
<code>m(r, c)</code> or <code>miss(r, c)</code>	Returns a $n \times k$ matrix filled with missing values. Cf. also <code>m()</code> function for values. Returns: matrix	<code>#m = m(5, 10);</code>
<code>multiply(x1, x2)</code>	Element by element multiplication of the two matrices. If <code>x2</code> is a row vector, each <code>x1</code> column will be multiplied with the corresponding value from the row vector. And if <code>x2</code> is a column vector, each <code>x1</code> row will be multiplied with the corresponding value from the column vector. Returns: matrix	<code>#x = multiply(#x1, #x2);</code>
<code>ones(n, k)</code>	Returns a $n \times k$ matrix filled with 1's Returns: matrix	<code>#m = ones(5, 10);</code>
<code>pack(v1, v2, ...)</code> <code>pack(<t1 t2>, v1, v2, ...)</code>	Using period <code>t1-t2</code> , the timeseries <code>v1, v2, ...</code> are packed into a $n \times k$ 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	<code>#m = pack(<2020 2030>, x, y, z);</code> Returns: a 11×3 matrix <code>#m</code> with the values.
<code>rows(x)</code>	Returns the number of rows of <code>x</code> . Returns: val	<code>%v = rows(#m);</code>
<code>sumc(x)</code>	Sum over cols Returns: matrix	<code>#m2 = sumc(#m1);</code>
<code>sumr(x)</code>	Sum over rows Returns: matrix	<code>#m2 = sumr(#m1);</code>
<code>t(x)</code>	Returns the transpose of a	<code>#m2 = t(#m1);</code>

	matrix. Returns: matrix	
trace(x)	Returns the trace of a matrix. Returns: val	<code>%v = trace(#m);</code>
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	<code>//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</code>
zeros(n, k)	Returns a n x k matrix filled with 0's. Zeroes() can be used as alias. Returns: matrix	<code>#m = zeros(5, 10);</code>

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 command 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 commands

[VAL](#), [SERIES](#), [LIST](#), [SHEET](#)

3.49 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 command. 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](#) command.

Related commands

[STRING](#), [DATE](#), [VAL](#)

3.50 MENU

MENU is not a command, 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 command 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 (command 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 <font
color="gray"> S1 </font></a></li>
      <li><a href="s2.gtb">Productivity <font
color="gray"> S2 </font></a></li>
      <li><a href="main.html">***MAIN MENU***
        Back</a></li>
    </ul>
  </body>
</html>
```

You may run command 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 to 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 (``) 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 `` for instance. To navigate to a parent folder, use '..', for instance ``. 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](#) command 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 commands

[TABLE](#), [EDIT](#)

3.51 MODE

The MODE command 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 commands. The Ref (reference) databank is essential in sim mode (to show simulation differences/multipliers). See [here](#) regarding an overview of commands that have primarily *sim* flavour.
- **Data mode:** Focused on [OPEN](#), [COPY](#), [IMPORT](#), [SERIES](#), [PRT](#), [CLOSE](#) and similar commands. The reference bank is often not used at all in this mode. See [here](#) regarding an overview of commands 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 commands 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 command. 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 command 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 command `mode sim;` in the same folder as the gekko.exe file.

Syntax

```
MODE mode;
MODE ?;
```

<i>mode</i>	Choose between <code>sim</code> , <code>data</code> or <code>mixed</code> . Default is <code>sim</code> mode.
<code>?</code>	Shows the current mode.

Example

This command 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 commands

[RESET](#), [RESTART](#), [INI](#)

3.52 MODEL

The MODEL command 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.

Regarding Gekko model files (i.e., files with extension .frm, containing model equations), see at the bottom of this help file regarding equation syntax etc. Please put a MODEL statement 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`. With this option, Gekko will stop when the first missing value is encountered, and report equation that produced the error.

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 gms 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;
```

	<p>This tells Gekko that equations <code>e_3</code> and <code>e_10</code> designate <code>p</code> as dependent variable, whereas equation <code>e_2</code> designates <code>q</code> 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 <code>e_1</code> was determining <code>q</code> instead of <code>p</code>, the <code>q</code>-line of the listfile could be <code>q; e_1; e_2.</code> instead of just <code>q; e_2.</code></p> <p>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 <code>OPTION model gams dep current = yes</code>, after which only current (non-lagged and non-leaded) variables can be identified as dependents.</p> <p>Note: The identification of "dependents" is only used in the DISP and DECOMP commands, not elsewhere. [New in 3.0.2]</p>
DUMP	<p>(Optional). With this option together with the <code><gms></code> option, a GAMS model will be dumped in Gekko form, as the file <code>dump.gcm</code>. This file can be used to inspect how Gekko translates GAMS-equations, and which variables are identified as dependents in each equation (and if they are identified via a DEP list). These equations may not run in Gekko, at the moment <code><dump></code> is only for debugging. [New in 3.0.2]</p>
GMS	<p>(Optional). With this option, Gekko will read GAMS equations from a <code>.gms</code> file (extension <code>.gms</code> will be added if missing). Gekko will search the <code>.gms</code> file for equations which can be shown with the <code>DISP</code> command. Use together with <code>OPTION model type</code> if you want to DISP the equations.</p>
<i>filename</i>	<p>Filenames may contain an absolute path like <code>c:\projects\gekko\myfile</code>, a relative path like <code>\gekko\myfile.gbk</code>, or be stated without a path. Filenames containing blanks and special characters should be put inside quotes. See more on filenames here. If the filename is set to <code>'*'</code>, you will be asked to choose the file in Windows Explorer.</p> <p>The extension <code>.frm</code> is automatically added, if it is missing.</p>

Example

If the model file name is `adam2.frm`, you may load the file as follows:

```
MODEL adam2;
```

Note that extension '.frm' is automatically added if it is missing (you may alternatively use `MODEL *;` to choose the model in Windows Explorer).

```
MODEL othermodels\adam3;
```

This will look for `adam3.frm` in the subfolder `othermodels`, relative the the Gekko working folder. When a model is loaded, it is first parsed. During this phase, you will get errors if parentheses are missing etc. Gekko will also warn against dublets (for instance two equations with the same left-hand side variable).

The MODEL statement also orders the equations, and compiles the model. The ordering splits the model into three 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 in order to speed up simulations (hence, PCIM's 'AFTER\$' statement is ignored if it is present in a model). 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.

After ordering, Gekko 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`, `#exottrue` and `#exoz` with lists of different kinds of variables. The lists are located in the Global databank, so that they survive [READ](#) statements.

Model files (.frm files) consist of equations (there is no limit on the number of equations other than available RAM). The equations have the following syntax:

FRML *frmlcode* *variable* = *expression*;

<i>frmlcode</i>	May be used to indicate auto-generated D-, J- and Z-variables.
<i>variable</i>	A variable, or an expression on a variable: <code>log(var)</code> , <code>dlog(var)</code> , <code>dif(var)</code> or <code>pch(var)</code>
<i>expression</i>	Any mathematical expression

An example to illustrate, we will consider a particularly simple equation:

```
FRML _GJDD dif(y) = dif(x) ;
```


Ignoring for a moment the formula code, we first note that the equation uses the `dif()` function, given as `dif(x) = x - x[-1]`. Hence, we get this equivalent equation:

```
FRML y = y[-1] + x - x[-1];
```

Next comes the formula code (`_GJDD`). This means that an add factor (JD) will be added, with the name JDy. So our equation is augmented into

```
FRML y = y[-1] + x - x[-1] + JDy;
```

Last, exogenization dummies are added, too. This is done in order to ease exogenization of the equation, if needed. In general, such variables are added in the following way:

```
FRML y = (1-Dy) * (y[-1] + x - x[-1] + JDy) + Dy * Zy;
```

As you can see, two variables are added, `Dy` and `Zy`. If `Dy = 0`, the two variables have no effect, but if `Dy = 1`, the equation reduces to `y = Zy`. I.e., the equation is given by the exogenous Z-variable `Zy` (or in other words, the variable `y` will always be set to this value, corresponding to exogenizing it). This is the equation used internally in Gekko. In addition to this equation, Gekko adds two more "reverted" equations regarding `JDy` and `Zy`. These equations are not part of the "real" model, but are calculated after each simulation:

```
Reverted1: JDy = y - (y[-1] + x - x[-1]);
Reverted2: Zy = y;
```

If `Dy = 0`, it is easy to see that `JDy` will not change its value (together with the above FRML, the first reverted equation reduces to `JDy = JDy`). But if `Dy = 1`, `JDy` will get the value that corresponds to what `Zy` might have been set to, `JDy = Zy - (y[-1] + x - x[-1])`. This way, you may set `Dy = 1`, and `Zy` to some chosen value. When you simulate, `y` will assume that value. Later on, you may reset `Dy = 0` and simulate. You will notice that `y` still assumes the chosen value. So these exogenization dummies can be used to change the levels of endogenous variables, but keeping them endogenous if a multiplier is run on top of that.

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 become more complicated, but the basic idea is the same.

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, either before or after the FRML statement(s) using it. Scalar variables in .gcm command files are not used: all such variables must be present in the model file itself.

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
dify(x) or diffy(x)	Yearly absolute time-difference	yes
dlog(x)	Logarithmic time-differences	yes
dlogy(x)	Yearly logarithmic time-differences	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
movavg(x, lags)	Moving average	no
movsum(x, lags)	Moving sum	no
pch(x)	Percent time-difference	yes
pchy(x)	Yearly percent time-difference	yes

Variable list

You may add a variable list with variable explanations at the end of the model file (.frm). 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 $qJ\{j\}\{i\}$ $j = t, e$ $i = a, b$ (mere 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: $qJ\{i\}\{j\}\{k\}\{m\}$ $i=a,b;j=1,2;k=x,y;m=n,m,nk$.
- 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 exist in a stand-alone file called `varlist.dat` (without the 'VARLIST;' line).

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.

You may put meta-information into the model file (.frm). As of now, `Info`, `Date`, 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
// 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](#) command), 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](#) command.

If you have a variable that is defined implicitly, 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 where `x` cannot be isolated. If you choose `OPTION solve method = newton`, an equation like the above should solve just fine with [SIM](#).

Related options

[OPTION](#) folder model = [empty];
[OPTION](#) model type = default; //default | gams
[OPTION](#) model gams dep current = no; //yes | no

Related commands

[SIM](#), [SERIES](#), [READ](#)

3.53 MULPRT

The MULPRT command 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](#) command. The key difference between the two print commands is that the operators are different:

- Short operators like `d`, `p`, `m`, `q` etc. cannot be used in the MULPRT command
- The long operators have different interpretation than in the PRT command.
- 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:	
<i>period</i>	(Optional). Local period, for instance 2010 2020, 2010q1 2020q4 or %per1 %per2+1.
<i>operators</i>	<i>operator operator ...</i>
<i>operator</i>	lev, abs, pch, gdif, v (note: v overrides any other operators)
<i>decimals</i>	DEC=number NDEC=number PDEC=number
<i>width</i>	WIDTH=number NWIDTH=number PWIDTH=number
<i>elements</i>	<i>element, element, ...</i>
<i>element</i>	variable 'label' < operators decimals width >
details:	

<i>operators</i>	Long operators: <code>lev</code> , <code>abs</code> , <code>pch</code> or <code>gdif</code> . These can be switched off by means of prefix <code>no</code> (for instance <code>nopch</code>), or added to existing default operators by means of underscore (for instance <code>_lev</code>). Default operators are <code>abs</code> and <code>pch</code> (absolute and percentage multiplier is printed). Finally, you may use the <code>v</code> operator to get verbose (detailed) output. Also see <code>OPTION print mulprt</code> (The so-called 'short' operators cannot be used in MULPRT).
<i>element</i>	The variable can be a variable name, a list (for instance <code>{#m}</code>), 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 <code>OPTION print fields ndec....</code>
PDEC=	Sets number of decimals for percentage numbers. See also <code>OPTION print fields pdec....</code>
WIDTH=	Sets width, will apply to all kinds of numbers.
NWIDTH=	Sets width for non-percentage numbers. See also <code>OPTION print fields nwidth....</code>
PWIDTH=	Sets width for percentage numbers. See also <code>OPTION print fields pwidth....</code>
ROWS	If set, the result will be transposed, i.e., with variables running downwards.
FILTER=	A timefilter can be activated or deactivated (see TIMEFILTER command). With <code><FILTER></code> or <code><FILTER=yes></code> , the current timefilter is used. With <code><NOFILTER></code> or <code><FILTER=no></code> , any filtering is deactivated. The filter type can also be changed locally, for instance <code><FILTER=hide></code> hides the out-filtered periods, whereas <code><FILTER=avg></code> averages the out-filtered periods. See <code>OPTION timefilter....</code>
BANK	(Optional). A bankname where variables are looked up. For instance <code>MULPRT <bank = b1> x;</code> is equivalent to <code>MULPRT</code>

	<code>b1:x;</code> . See also <code><REF = ...></code> . These options can be convenient instead of opening and closing banks.
REF	(Optional). A bankname where reference variables are looked up. For instance <code>MULPRT <bank = b1 ref = b2 m> x;</code> uses banks <code>b1</code> and <code>b2</code> for the multiplier. See also <code><BANK = ...></code> . 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: <code>x</code>
abs	Absolute multiplier: <code>x-@x[-1]</code>
pch	Relative multiplier: <code>(x/@ -1)*100</code>
gdif	Multiplier in growth rate: <code>(x/x[-1] -1)*100 - (@x/@x[-1] -1)*100</code>
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 command 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` command.

Related options

[OPTION](#) freq a; [a|q|m]

[OPTION](#) print filewidth = 130;

[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 commands

[PLOT](#), [SHEET](#), [CLIP](#), [PRT](#), [DISP](#), [DECOMP](#)

3.54 OLS

The OLS command 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>`.

Syntax

```
OLS <period CONSTANT=... DUMP=... DUMPOPTIONS=...> name leftside =
var1, var2, ... IMPOSE=... ;
```

<i>period</i>	(Optional). Local period, for instance 2010 2020, 2010q1 2020q4 or %per1 %per2+1.
CONSTANT =	With <code><constant = no></code> , a constant term is not added automatically.
DUMP=	(Optional). Dumps the results as a FRML equation for use in models. You may use <code>OLS<dump></code> to produce a <code>ols.frm</code> file. <code>OLS<dump=eqs.frm></code> will use the filename <code>eqs.frm</code> 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 <code>+</code> or <code>-</code> . This will introduce superfluous parentheses in expressions like <code>a * (b + c)</code> or <code>exp(a - b)</code> etc. [New in 3.0.6]
DUMPOPTIONS=	(Optional). If you use <code>OLS<dump=eqs.frm dumpoptions='append'></code> , the results will be appended to an existing <code>eqs.frm</code> file. These options will be augmented with styling, FRML code, etc. [New in 3.0.6]
<i>name</i>	(Optional). A name for the equation, used to name the results. If no name is given, <code>ols</code> is used as name.
<i>leftside</i>	The leftside variable (may be an expression)
<i>var1, ...</i>	A list of variable names or expressions. A constant term is added automatically, unless you use option <code><constant = no></code> .

IMPOSE

(Optional). You can impose linear restrictions on the parameters, via a suitable matrix. One restriction per row of the matrix, cf. example below.

Results:

Note that if a name is given, `ols` is replaced with that name.

<i>ols_predict</i>	A timeseries with the predicted values
<i>ols_residual</i>	A timeseries with the residuals
<i>#ols_param</i>	A matrix with estimated parameters
<i>#ols_se</i>	A matrix with standard errors on parameters
<i>#ols_t</i>	A matrix with t-values on parameters
<i>#ols_covar</i>	A matrix with the variance-covariance matrix (of parameters)
<i>#ols_corr</i>	A matrix with the correlation matrix (of parameters)
<i>#ols_stats</i>	<p>A matrix containing different measures (analogous to the .stats matrix in AREMOS):</p> <ol style="list-style-type: none"> 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 <p>(At some point, a map will be used instead for these measures).</p>

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 lnal, pcp, bull;
SERIES <1998 2010> lnal = 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') ;
SERIES <1998 2010> pcp = 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') ;
SERIES <1998 2010> bull = 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(lnal) = dlog(pcp), dlog(pcp.1), bull, bull.1;
```

The commands produce the following screen output:

```
OLS estimation 2000-2010 (n = 11)
dlog(lnal)
```

Variable	Estimate	Std error	T-stat
dlog(pcp)	0.144517	0.227011	0.64
dlog(pcp.1)	0.613875	0.236473	2.60
bull	0.186740	0.202534	0.92
bull.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` command.

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](#) command):

```
#r = [1, 1, 0, 0, 0, 0.80; 0, 0, 1, -1, 0, 0];
OLS <2000 2010> dlog(lnal) = dlog(pcp), dlog(pcp.1), bull, bull.1
IMPOSE = #r;
```

If the parameters are called $b_{\{i\}}$, the first restriction is equivalent to $1*b_1 + 1*b_2 + 0*b_3 + 0*b_4 + 0*b_5 = 0.80$, or $b_1 + b_2 = 0.80$. The second restriction is equivalent to $0*b_1 + 0*b_2 + 1*b_3 + (-1)*b_4 + 0*b_5 = 0$, or $b_3 = b_4$. 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(lnal)
-----
Variable           Estimate           Std error           T-stat
-----
dlog(pcp)           0.167642           0.180625           0.93
dlog(pcp.1)         0.632358           0.180625           3.50
bull                -0.0863480         0.0794747          1.09
bull.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. 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 parameters, 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/errors to Excel or other spreadsheets.

Related commands

[ANALYZE](#), [MATRIX](#), [MODEL](#), [R_RUN](#)

3.55 OPEN

The OPEN command 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 command (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 command 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](#) command.

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 a list of available in-memory databanks (including Work and Ref) 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=... > filename1 AS alias1, filename2 AS
alias2, ... ;
```

EDIT	The databank is opened in first position, as editable. (See also <code>OPEN<create></code>).
FIRST	The databank is opened in first position, as non-editable (protected)
LAST	The databank is opened in last position, as non-editable (protected). This corresponds to the way databanks are opened with the OPEN command in Gekko versions 2.1.3 and later.

POS=	(Optional) Indicate an integer value. The databank is opened in the n-th position, as non-editable (protected)
SAVE	With <code>OPEN<save=no></code> , 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 <code>CLOSE<save=no></code> .
CREATE	If this option is set, Gekko will accept an <code>OPEN<create>b1</code> , even if b1.gbk does not exist beforehand. Regarding a non-existing file, this is similar to <code>OPEN<edit></code> , the difference is that <code>OPEN<edit></code> puts the bank in first position, as editable. The reason why <code><create></code> is not the default way of opening a databank is to avoid errors if a user misspells a databank name.
<i>format</i>	(Optional). Can be <code>tsd</code> , <code>gbk</code> , <code>pcim</code> , <code>csv</code> , <code>prn</code> , <code>xls</code> , <code>xlsx</code> . The default file format is <code>gbk</code> .
<i>filename</i>	<p>Filenames may contain an absolute path like <code>c:\projects\gekko\myfile</code>, a relative path like <code>\gekko\myfile.gbk</code>, or be stated without a path. Filenames containing blanks and special characters should be put inside quotes. See more on filenames here.</p> <p>If the filename is set to '*', you will be asked to choose the file in Windows Explorer.</p> <p>The extension <code>.gbk</code> is automatically added, if it is missing.</p>
COLS	(Optional). Only for <code>.xlsx</code> and <code>.csv</code> files: this indicates whether the timeseries are running downwards in columns. Note that for <code>.csv</code> files, you indicate this in the first 'cell' (date/name).
DATEFORM AT= DATETYPE=	(Optional). These options control the date format for <code>.xlsx</code> and <code>.csv</code> files. <code>DATEFORMAT</code> can be either <code>'gekko'</code> (default) or a format string like <code>'yyyy-mm-dd'</code> , and the latter may contain a <code>first</code> or <code>last</code> indicator, for instance <code>'yyyy-mm-dd last'</code> , which indicates for quarterly or monthly data that the <i>last</i> day of the quarter or month is used. <code>DATETYPE</code> can be either <code>'text'</code> or <code>'excel'</code> . In the former case, the dates are understood as text strings (for instance <code>'2020q3'</code> or <code>'2020-09-30'</code> for a quarterly date), and in the latter case (not relevant for <code>.csv</code> 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. [New in 3.0.5].
<i>alias</i>	(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` command puts databank in the last position on the list. The following yields the same:

```
OPEN bank1, bank2;
```

You may use `OPEN<first>`, `OPEN<sec>`, `OPEN<pos=n>`, `OPEN<last>` to open a databank in a specific position on the databank list (F2). The command `OPEN<edit>mybank;` is the same as `OPEN<first>mybank;` `UNLOCK mybank;` -- and in that case, a subsequent `SERIES xx1 = 100;` will put the series `xx1` into `mybank`:

```
OPEN <edit> mybank;  
SERIES xx1 = 100;  
CLOSE mybank;
```

After the `CLOSE` command, 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` command, 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](#) command 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.

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 commands

[READ](#), [WRITE](#), [CLONE](#), [UNLOCK](#), [LOCK](#)

3.56 OPTION

The command sets various global 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).

Note that Gekko provides suggestions (a small popup-window) for the OPTION command (see "OPTION interface suggestions = option"), to aid the user navigate the option tree.

Syntax

OPTION *type* = *value*;

<i>type</i>	One or more space-delimited options
<i>value</i>	Boolean (e.g.: <code>yes no</code>) Integer (e.g.: 200) Floating point number (e.g.: 0.45) String (eg.: <code>newton</code>)

Note: the '=' may be omitted: `OPTION freq q;` and `OPTION freq = q;` are equivalent.

Below the full list of options are provided, with their default values. You may set temporary options via the [BLOCK](#) structure.

Options

OPTION databank create auto = no; [yes|no]

If `yes`, timeseries will be auto-created. For instance, `SERIES 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](#) command.

OPTION databank create message = yes; [yes|no]

A message is issued when a new timeseries is created.

OPTION databank compare tabs = 1.0;

OPTION databank compare trel = 0.0001;

These options set limits regarding the [COMPARE](#) command (comparing the first-position and reference databanks).

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 in many instances speed up reading/opening files on network drives.

OPTION databank file gbk compress = yes;
OPTION databank file gbk internal = 'databank.data';
OPTION databank file gbk version = 1.2; [1.0|1.1|1.2]

Some options regarding the .gbk format, applying only to writing or closing databanks ([WRITE/CLOSE](#)). Without compression there is 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 `databank.data`, but can be changed here (it used to be `databank.bin`). Regarding the .gbk version, there is normally no need to change this.

OPTION databank search = no;

If `yes`, variables will be searched for in all open databanks except Ref (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](#) command, and the [databank searching](#) page. Note that the Local or Global databanks are always searchable, independent on [MODE](#) etc.

OPTION decomp maxlag = 10;
OPTION decomp maxlead = 10;

When [decomposing](#), effects via lags are restricted within these values (larger values = slower decomp).

OPTION folder = yes; [yes|no]

If `yes`, Gekko will look for files in predefined folders (is switched on per default).

OPTION folder bank = [empty];
OPTION folder bank1 = [empty];
OPTION folder bank2 = [empty];

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 = [empty];
OPTION folder command1 = [empty];
OPTION folder command2 = [empty];

While looking for command 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 = [empty];

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 = [empty];

Folder where Gekko looks for menu files (.html files).

OPTION folder model = [empty];

Folder where Gekko looks for model files (.frm files).

OPTION folder pipe = [empty];

Folder where Gekko pipes out text output files (see the [PIPE](#) command). If no folder is indicated, piped files will end up in the working folder.

OPTION folder table = [empty];

OPTION folder table1 = [empty];

OPTION folder table2 = [empty];

Folders where Gekko looks for table files (.gtb files). Gekko will first look in `table`, then `table1`, and last `table2` folders.

OPTION folder working = [empty];

This changes the Gekko working folder. (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|u]

Sets frequency of timeseries to 'a' (annual), 'q' (quarterly), 'm' (monthly), or 'u' (undated). You may use a string inside {}-bracket, for instance `%f = 'q';` `OPTION freq = {%f};`

OPTION gams exe folder = [empty];

This option starts out empty, and if so, GAMS will try to auto-detect the location of the executable for GAMS (gams.exe). 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 (low-level API). If this poses problems, try the more robust normal API by setting `OPTION gams fast = no;`.

OPTION gams time freq = 'a'; [a, q, m, u]

OPTION gams time set = 't';

OPTION gams time prefix = '';

OPTION gams time offset = 0;

OPTION gams time detect auto = no;

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. In this case, you may also set `OPTION gams time detect auto = yes;`. If so, any dimension element with the pattern 't' + integer will be understood as a time period. This may happen if a variable/parameter does not use a strict set (name 't' in this case, cf. `OPTION gams time set`) for the time dimension, and Gekko

may in that case create a lot of timeless timeseries with dimensions like `x['2020']`, `x['2021']`, etc.

OPTION interface alias = no; [yes|no]

When this option is set, Gekko will look for a 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;` will be 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](#) command, (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> local option, where the databank is written as a csv (comma separated) file. The decimal separator in the .csv file will correspond to the indicated setting.

OPTION interface csv delimiter = semicolon; [semicolon|comma]

Used with [IMPORT](#)<csv> and [EXPORT](#)<csv>. The field delimiter is `;` per default, but can be set to `,` 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 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 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 `'#NAVN?'` for missing values (alternatively, it uses `'#NAME?'`).

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](#) command). 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 = sim; [sim|data|mixed]

Sets some interface messages/warnings etc. regarding different modes (see the [MODE](#) command). This option is not intended for direct use: please use the MODE command 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](#) is somewhat similar. The option is primarily set in command files/procedures/functions to avoid voluminous screen output blocking and slowing the user interface. [New in 3.0.6].

OPTION interface remote = no; [yes|no]

If this is set to 'yes', Gekko will look for a file named `remote.gcm` 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]

If active, Gekko will play a sound when a command file finishes (or has an error). See also `OPTION interface sound type` and `OPTION interface sound wait` for more options regarding sounds.

OPTION interface sound type = bowl; [bowl|ding|notify|ring]

The sound type, choose between `bowl`, `ding`, `notify`, `ring`.

OPTION interface sound wait = 60;

The minimum duration of the job (in seconds), before for Gekko plays a sound. Used to avoid too much noise regarding fast-executing jobs.

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 '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 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 cache is used.

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 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 gams dep current = no; [yes|no]

When reading a GAMS 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 = lhs; [lhs|eqname]

When reading a GAMS 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](#). [New in 3.0.3].

OPTION model type = default; [default|gams]

If `type = gams`, Gekko will handle ENDO, EXO, UNFIX and DISP differently to ease the interaction with GAMS.

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 accidentally 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. Intended for Gekko 2.0 graphs, but also works for Gekko 2.2. graphs.

OPTION plot using = [empty];

With this option, a global `.gpt` PLOT template file can be set, for instance `OPTION plot using = m:\common\gekko.gpt;`. Subsequent PLOT commands will then use that template (unless the PLOT command itself has a `using` argument).

OPTION plot xlabel annual = at; [at|between]

OPTION plot xlabel 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 xlabel 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 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](#) command. Only applies to `OPTION print freq = pretty;`.

OPTION print disp maxlines = 3;

The number of lines of data shown per default in the [DISP](#) command. You may choose -1 for infinite, 0 means that no data are shown.

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 freq = pretty; [pretty|simple]

If this option is set to `pretty`, timeseries of frequencies quarterly and annual are [printed](#) with years and quarters/months separated, for better readability. If the option is set to `simple`, the dates are shown without such separation.

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 non-percent fields, and the two last for percent fields. The `ndec` and `nwidth` settings also affect printing of matrices.

OPTION print filewidth = 130;

Line width (number of characters) when printing to a file

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](#) command. 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 command.

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` command. 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 command.

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 the global 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 subseries exist or not.

OPTION print width = 100;

Line width (number of characters) when printing to screen

OPTION r exe folder = [empty];

This option starts out empty, and if so, Gekko will try to auto-detect the location of the executable folder for R (`R.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'. You should state the path to an executable file (`R.exe`), but if you omit a trailing '`R.exe`' or '`\R.exe`', Gekko will fill these in for you. So in the above example, `OPTION r exe folder = 'c:\Program Files\R\R-3.0.0\bin';` would have been equivalent.

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 subseries). 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. The four options above that also handle missings deal with the question of what to do if a normal series or array-subseries does not exist at all. In contrast, 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 expression `x = x[-1] + 1;` accumulate over time, because the expression is run 'dynamically', one period at a time consecutively. The option will only be active, if the right-hand side of the expression 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 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 command 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 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 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 quarterly and annual are [printed](#) with years and quarters/months separated, 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]

[NOT USED YET]. These options set the default way of printing with the SHEET<mul> (not implemented yet) command. 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 command.

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 command. Per default, `abs` is chosen, so SHEET shows the absolute level (`abs`). These can be overridden via the option fields in the SHEET command.

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](#) command. Only applies to `OPTION sheet freq = pretty;`.

OPTION sheet cols = no; [yes|no]

OPTION sheet rows = yes; [yes|no]

Per default, the SHEET command 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 solve data create auto = yes; [yes|no]

If yes, when a databank is read by means of the general [READ](#) command (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](#) command.

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 lead 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 lead 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 `conv1` (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 lead endogenous variables. If set to `fair`, the Fair-Taylor algorithm is used, and if set to `nfair`, the Newton-Fair-Taylor method is used. If set to `none`, no forward-looking method is used. In that case, the lead 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 itemax = 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 command [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](#) command 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](#) command. 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 system code split = 20; [0 or positive integer]

This is a very technical option related to how Gekko compiles command files. If > 0, long non-looping command 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. 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 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 commands, 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.

3.57 PAUSE

PAUSE is for pausing the command flow: the command will wait for the user to press [Enter]. See also [ACCEPT](#).

Syntax

PAUSE info;

<i>info</i>	(Optional). Text string to be displayed when a command flow is paused. You can use '\n' inside the string to insert a new line.
-------------	---

Example

The command may contain text inside single quotes:

```
PAUSE 'This is the first part of the scenario';
```

Related commands

[RETURN](#), [STOP](#), [EXIT](#), [ACCEPT](#)

3.58 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 <code><body></code> 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: <code>TELL '<p>Hello</p>';</code>). 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.
<i>filename</i>	(Optional, not used with PAUSE, CONTINUE, STOP options). Filenames may be contain an absolute path like <code>c:\projects\gekko\myfile</code> , a relative path like

\gekko\myfile.gbk, or be stated without a path. Filenames containing blanks and special characters should be put inside quotes. 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;

3.59 PLOT

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.

You can create plots of variables with the `PLOT` command, see [demo graphs](#). The `PLOT` arguments have the same syntax as `PRT/MULPRT`, `SHEET` and `CLIP`. You can store plot options/settings in external `.gpt` files, so that you can easily reuse the styling. `PLOT` uses the free open-source [gnuplot 5.1](#) internally, and the settings/options of the `PLOT` command and corresponding `.gpt` files are named to match gnuplot naming conventions. `PLOT` can create an `.emf`, `.svg`, `.png` or `.pdf` file silently, if the `FILE=` option is used. The `.emf` files are practical for MS Office applications, including Word. 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](#) command. You can use a global `.gpt` file via `"OPTION plot using = ... ;"`.

Note that Gekko 3.0 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.

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):

<i>period</i>	(Optional). Local period, for instance 2010 2020, 2010q1 2020q4 or %per1 %per2+1.
<i>operator</i>	(Optional). 'Long': abs, dif, pch, gdif, or 'short': n, d, p, dp, m, q, mp, r, rd, rp, rdp
<i>variables</i>	Name of the variable(s) printed. Several variables can be printed at once using var1, var2, var3, You may also use lists or expressions.
PLOTCODE =	(Optional). The contents of this string must be gnuplot code (for instance 'set' commands separated by semicolon), and the contents is sent to gnuplot and inserted just before the gnuplot 'plot' statement.
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 command: 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 subseries and missing data values like GAMS, treating them as zero for sums and mathematical expressions, or skipping the printing of a subseries 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=	<p>(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 <code>OPTION plot using = ...</code>.</p> <p>Filenames may be contain an absolute path like <code>c:\projects\gekko\myfile</code>, a relative path like <code>\gekko\myfile.gbk</code>, or be stated without a path. Filenames containing blanks and special characters should be put inside quotes. See more on filenames here.</p>
FILE=	<p>(Optional). Use extension <code>emf</code> (default), <code>png</code>, <code>svg</code> or <code>pdf</code>. 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 <code>.png/.svg/.pdf</code> instead, and PLOT will produce the file in that format. The .svg files are very useful for insertion into html document.</p> <p>Filenames may be contain an absolute path like <code>c:\projects\gekko\myfile</code>, a relative path like <code>\gekko\myfile.gbk</code>, or be stated without a path. Filenames containing blanks and special characters should be put inside quotes. See more on filenames here.</p>

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 command, you may also use element-specific operators (for instance `PLOT unemp, gdp<p>;`). See the [PRT](#) command 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;`, not `PLOT;`. If in doubt, try using quotes.

type	<p>The most used of the following are the line-types <code>linespoints</code> and <code>lines</code>, together with <code>boxes</code> (column chart).</p>
------	--

	<ul style="list-style-type: none"> • <code>linespoints</code>, <code>lines</code>: Normal lines, with or without point markers. • <code>boxes</code>: bar chart/histogram. If several timeseries are boxes, these will be clustered, unless the stacked option is set. • <code>filledcurves</code>: lines where the area below each line is an area. If the stacked option is set, the areas are stacked instead of overlaid. • <code>steps</code>: Step-wise lines, a bit box-like. • <code>points</code>: Just the point markers, no lines. • <code>dots</code>: Just tiny dots. • <code>impulses</code>: vertical lines instead of boxes. <p>Quotes may be omitted.</p>
<code>dashtype</code>	'1':normal, '2':dashed, '3':dotted, ... (default: '1'). More . Quotes should be used.
<code>linewidth</code>	A number. Default: 3.
<code>linecolor</code>	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.
<code>pointtype</code>	<p>An integer. These points are shown for each period, if the <code>linetype</code> is <code>linespoints</code> or <code>points</code>.</p> <ul style="list-style-type: none"> • 1:'+' • 2:'x' • 3:'*' • 4:box • ... etc. <p>Default: 7 (circle). More.</p>
<code>pointsize</code>	A number. The size of the points. Default: 0.5.
<code>fillstyle</code>	String. Only relevant for <code>linetype boxes</code> . 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 <code>linetype -1</code> '. Default: 'solid'. More . Quotes should be used.
<code>title</code>	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 <code>OPTION plot decimalseparator =</code>
grid	Choose between yes no xline yline. If <code>yes</code> , both vertical and horizontal lines are shown. If <code>xline</code> , only vertical lines are shown. If <code>yline</code> , only horizontal lines are shown. Default: <code>yes</code> . Quotes can be omitted.
gridstyle	<p>This sets how gridlines are formatted, stated in gnuplot syntax. Quotes should be used.</p> <ul style="list-style-type: none"> • Default is the following, which are light grey dashed lines: <code>'linecolor rgb "#d3d3d3" dashtype 3 linewidth 1.5'</code>. • The gridstyle <code>'linecolor black dashtype 2 linewidth 2.0'</code> will provide black dashed lines which look ok in Word. • To emulate the solid grey style of Gekko 2.2.4 and earlier, use <code>'linecolor rgb "#f0f0f0" linetype 1 linewidth 1'</code>. <p>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.</p>
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:

	<p>'out horiz bot center Left reverse'. Quotes should be used.</p> <p>To remove the key completely, you can use gnuplot-code <code>'set key off'</code>: <code>PLOT<plotcode='set key off'>;</code>.</p>
palette	<p>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.</p>
stack	<p>If the element is set active (<code><stack></code>), <code>boxes</code> are stacked instead of clustered, and <code>filledcurves</code> are stacked instead of overlaid. Default: <code>no</code>.</p>
boxwidth	<p>The width of the boxes. Set to 1 for max width. Default: 0.75.</p>
boxgap	<p>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.</p>
separate	<p>With the <code>separate</code> 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 <code>y2</code> options regarding the lines. This functionality is Gekko-specific and does not yet work for stacked boxes (option <code>boxstack</code>), where the scaling will not be precise (this will be fixed in a future version). Default: <code>no</code>.</p>
xline	<p>Vertical line at the given period. Several lines may be given. For instance: <code><xline>2020q2</xline></code>.</p>
xlinebefore	<p>Vertical line between the given period and the period before. Several lines may be given. For instance: <code><xlinebefore>2020q2</xlinebefore></code>.</p>
xlineafter	<p>Vertical line between the given period and the period after. Several lines may be given. <code><xlineafter>2020q2</xlineafter></code></p>
x(2)zeroaxis	<p>The <code>xzeroaxis</code> is the horizontal axis corresponding to <code>y=0</code>, and the <code>x2zeroaxis</code> is the axis corresponding to <code>y2=0</code> (the right-</p>

	hand side y-axis). These <code>xzeroaxes</code> will only be shown if the y or y2 values change sign. Default for <code>xzeroaxis</code> is <code>yes</code> , and default for <code>x2zeroaxis</code> is <code>no</code> .
<code>ymirror</code>	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 <code>ymirror</code> setting is ignored.
<code>y(2)title</code>	A title for the y or y2 axis. Quotes should be used. If the title should break, you may use a '\n', for instance <code>PLOT<ytitle='Balance\nof payments'>;</code> .
<code>y(2)line</code>	Horizontal line at the given y- or y2-value. Several lines may be given. For instance: <code><yline>150</yline></code> .
<code>y(2)max</code>	Fixed max for the y or y2 values. Will overrule any <code>maxhard</code> or <code>maxsoft</code> values. Can cut off data points.
<code>y(2)maxhard</code>	All values > <code>maxhard</code> are filtered out, but all values < <code>maxhard</code> are shown. This setting is practical for filtering out outliers. Think of 'hard' as being capable of cutting off data points.
<code>y(2)maxsoft</code>	All values are shown, but the axis will not scale down below <code>yminsoft</code> . 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.
<code>y(2)min</code>	Fixed min for the y or y2 values. Will overrule any <code>minhard</code> or <code>minsoft</code> values. Can cut off data points.
<code>y(2)minhard</code>	All values < <code>minhard</code> are filtered out, but all values > <code>minhard</code> are shown. This setting is practical for filtering out outliers. Think of 'hard' as being capable of cutting off data points.
<code>y(2)minsoft</code>	All values are shown, but the axis will not scale up above <code>yminsoft</code> . 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.
<code>label</code>	A free-floating label is inserted at the given position. Several labels may be given. Quotes should be used. [Not available yet]
<code>arrow</code>	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 command

```
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 command, 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 ';'.

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](#) command (if used, choose 'View' --> 'Expand All' to unfold all XML nodes). You may also use Notepad (cf. the [EDIT](#) command), 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 '<', '>', and '&');).

If the file is not in valid XML syntax, Gekko will complain that the file is invalid and abort.

Note

PLOT produces a graph by means of the open-source program [gnuplot](#) as an underlying engine. You do not need to install anything in order to use PLOT (the Gekko installation files contain gnuplot).

In the graph window, you may change the so-called operators by clicking on the radio buttons (or multiplier checkbox). 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' button. There are also options to save the graph as a emf/svg/png/pdf files.

You may close the PLOT graph window by pressing the 'Esc' button.

Per default, PLOT will place annual and undated data at the x-tics, and quarterly and monthly data between x-tics. See `OPTION plot xlabel` ... 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 an .emf file, and the same .emf file may look a bit different when imported into Word (or converted to pdf or printed from Word). Also, a .svg version of the graph may look different in html. The differences apply to, for instance, dashed lines and fonts. 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.
- Indexed data, for instance showing all lines as index 2015=1, perhaps using operator <i> for index.
- Options <a>, <ad>, and <ap> For instance, "PLOT<a>x1, x2;" would be the same as "PLOT x1, @x1, x2, @x2;", showing values from the Reference databank.
- Multi-plots.
- 3D-plots.
- Scatterplots.
- Free-floating labels and arrows.
- Outputting in more file formats like latex, etc.
- It is the intention to make it possible to use R (using its plot function) as the graphing engine, too. Outputting R code instead of gnuplot code would not be too difficult. But gnuplot works ok, and is a small program that can be easily bundled with the Gekko package.

Related options

OPTION plot decimalseparator = period; [comma|period]
OPTION plot lines points = yes; [yes|no]
OPTION plot new = yes; [yes|no]
OPTION plot using = [filename];
OPTION plot xlabel annual = at; [at|between]
OPTION plot xlabel nonannual = between; [at|between]
OPTION plot xlabel digits = 4; [2|4]

Related commands

[PRT](#), [SHEET](#), [CLIP](#), [TABLE](#), [XEDIT](#), [EDIT](#), [CUT](#)

3.60 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 commands. A procedure without arguments is similar to running a command file, cf. [RUN](#).

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;
```

<i>t1, t2</i>	(Optional). You may state optional time period parameters inside <code><></code> -brackets, for instance <code>PROCEDURE f <date %t1, date %t2>, series x;</code> after which <code>%t1</code> and <code>%t2</code> are assigned to for instance 2020 and 2030 in the call <code>f <2020 2030 z;</code> . If the procedure is called without <code><></code> -brackets, for instance <code>f z;</code> , the parameters <code>%t1</code> and <code>%t2</code> are assigned to the local/global time period instead. Using <code><></code> -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.
<i>type1, ...</i>	Types of variables: <code>series</code> , <code>val</code> , <code>date</code> , <code>string</code> , <code>list</code> , <code>map</code> , <code>matrix</code> . You may also use the special <code>name</code> type for parameters, which behaves 100% as a <code>string</code> inside the procedure, but where the single quotes are omitted when calling the procedure from outside (the shorter call <code>f y</code> is used instead of <code>f 'y'</code>).
<i>var1, . .</i>	The parameters/variables/expressions
<i>label1, ...</i>	(Optional). A label for the parameter, used if the procedure is prompting (called with <code>f?</code>). See more about prompting below.
<i>default 1, ...</i>	(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 <code>f?</code>) and the parameter is omitted, the default value is shown in the dialog box. In the dialog box, <code>Enter</code> or <code>Escape</code> will return

	the default value, and fire up the next dialog box (for the next optional parameter). If a <code>;</code> 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 (<code>'</code>) in the input box is optional. At the moment, only <code>val</code> , <code>date</code> and <code>string</code> types can be used for prompting input boxes.
<i>procname</i>	The procedure name. The name cannot be the same as an existing Gekko command name.
<i>body</i>	The procedure body, that is, the commands to be performed. If the body contains a RETURN statement, the procedure will abort at that point (just like a normal command file, cf. RUN).

Tip: if you need to stop execution at a particular line, try inserting a line with a non-existing function like for instance `stop();`. This will abort the program in a clean way and make it possible to inspect variables etc.

When calling the procedure, the arguments are separated with blanks, not commas.

How to use a library of Gekko functions/procedures in 3.0?

In Gekko 3.0, the `OPTION library file = ...;` is obsolete. Instead, you can just put your user-defined functions/procedures in for instance a file called `lib.gcm`. Afterwards, you can define a [gekko.ini](#) file containing the line `RUN lib.gcm;` so that `lib.gcm` is always run at Gekko startup, or after a [RESTART](#). See the `lib.gcm` example on the [RESTART](#) help page. In Gekko 3.0, user functions/procedures are always available after they have been defined, as long as the use is chronologically after the definition.

Procedure hints

If a procedure has syntax errors, you may try to out-comment the PROCEDURE statement and corresponding END statement for better error messages. Procedure arguments do not reside in any databanks, so if you have a procedure like `PROCEDURE f series x; RUN data.gcm; END;` you cannot expect to use `x` inside the `data.gcm` command file, for instance expecting it to reside in the first-position databank (regarding procedure arguments, in many cases using the `name` type is more practical than the `series` type).

Negative arguments: beware that when calling a procedure, the arguments are blank-separated, for instance `PRINTVARS x y z;`. This will look for a procedure `PRINTVARS` with 3 arguments. But if, for instance, one of the arguments is negative, some care must be taken. For instance, `PRINTVARS x -y z;` will look for a procedure `PRINTVARS` with 2 arguments, since `x-y` is interpreted as one variable

(expression). In such cases, you may consider using a [FUNCTION](#) without return value, like `PRINTVARS (x, -y, z) ;`, or use a parenthesis to avoid the subtraction, as in `PRINTVARS x (-y) z ;`.

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:
//
//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 options

[OPTION](#) library file = [filename];

Related commands

[FUNCTION](#), [RUN](#)

3.61 PRT

The PRT command prints variables or expressions (you may use P, PRI or PRINT as synonyms for PRT). Regarding [series](#), 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](#) command is very similar to PRT, but compares data in the first-position and reference databanks. You may use the more detailed [DISP](#) command, 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 decimals width ROWS FILTER=... TITLE=...
  COLLAPSE=... NOMAX BANK=... REF=... SPLIT MISSING=... > 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:

<i>period</i>	(Optional). Local period, for instance 2010 2020, 2010q1 2020q4 or % per1 %per2+1.
<i>operator s</i>	<i>operator operator ...</i>
<i>operator</i>	'Long': abs, dif, pch, gdif, or 'short': n, d, p, dp, m, q, mp, l, dl, r, rd, rp, rdp, rl, rdl
<i>decimals</i>	DEC=number NDEC=number PDEC=number
<i>width</i>	WIDTH=number NWIDTH=number PWIDTH=number
<i>elements</i>	<i>element element ...</i>
<i>element</i>	variable label < <i>operators decimals width</i> >

details:

<i>operator s</i>	(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. See <code>OPTION print prt</code>
<i>element</i>	The variable can be a variable name, a list (for instance {#m}), or an expression. A label can be provided (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.
DEC	Sets number of decimals, will apply to all kinds of numbers.
NDEC	Sets number of decimals for non-percentage numbers. See also <code>OPTION print fields ndec....</code>

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 command). 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];. It only works when 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 <code>PRT <split> x[#i], @x[#i];</code> . In that case, you may prefer to use for instance the <code><missing = m></code> option, so that all columns (<code>#i</code>) are shown (and are hence aligned), regardless of whether the subseries exist or not.
MISSING =	(Optional). With <code><missing = ignore></code> , PRT will deal with missing array subseries and missing data values like GAMS, treating them as zero for sums and mathematical expressions, or skipping the printing of a subseries if it does not exist. The following options are set locally and reverted afterwards: <code>option series array print missing = skip;</code> <code>option series array calc missing = zero;</code> <code>option series data missing = zero.</code> See also the appendix page on missings .
FILE	(Optional). A filename that the print is put into. Filenames may contain an absolute path like <code>c:\projects\gekko\myfile</code> , a relative path like <code>\gekko\myfile.gbk</code> , or be stated without a path. Filenames containing blanks and special characters should be put inside quotes. 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

<i>abs</i>	Absolute level: x
<i>dif</i>	Absolute time change: $x - x[-1]$
<i>pch</i>	Growth rate: $(x/x[-1] - 1) * 100$
<i>gdif</i>	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

<code>n</code>	Absolute level: x . Equivalent to no use of operators.
<code>d</code>	Absolute time change: $x - x[-1]$
<code>p</code>	Growth rate: $(x/x[-1] - 1) * 100$
<code>dp</code>	Change in growth rate: $(x/x[-1] - 1) * 100 - (x[-1]/x[-2] - 1) * 100$
<code>m</code>	Absolute multiplier: $x - @x$
<code>q</code>	Relative multiplier: $(x/@x - 1) * 100$
<code>mp</code>	Multiplier in growth rate: $(x/x[-1] - 1) * 100 - (@x/@x[-1] - 1) * 100$
<code>l</code>	Log: $\log(x)$. [New in 3.0.3].
<code>dl</code>	Log-difference: $\log(x) - \log(x[-1])$. [New in 3.0.3].
<code>r</code>	Absolute level in reference databank: $@x$. Code ' <code>rn</code> ' is equivalent.
<code>rd</code>	Absolute time change in the reference databank: $x - x[-1]$
<code>rp</code>	Growth rate in reference databank: $(@x/@x[-1] - 1) * 100$
<code>rdp</code>	Change in growth rate in reference databank: $(@x/@x[-1] - 1) * 100 - (@x[-1]/@x[-2] - 1) * 100$
<code>rl</code>	Log: $\log(@x)$
<code>rdl</code>	Log-difference: $\log(@x) - \log(@x[-1])$

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'. For instance:

```
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 as you like, and mixed frequencies work for PLOT, too.

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';
```

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

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. 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

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[-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 r p 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](#) command (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 some commands, 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 commands by means of marking the block before pressing [Enter]. Alternatively, and better, RUN the commands from a command 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 command 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]
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 filewidth = 130;
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 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 commands

[MULPRT](#), [PLOT](#), [SHEET](#), [CLIP](#), [DISP](#), [DECOMP](#)

3.62 R_EXPORT

The R interface has three logical parts: [R_FILE](#), [R_EXPORT](#), and [R_RUN](#), and these commands should appear in that order. The [R_FILE](#) command points to an existing R file, into which data is inserted. [R_EXPORT](#) inserts Gekko data (matrices) into the R file. And [R_RUN](#) runs the R file, and returns R-matrices from R back to Gekko.

[R_EXPORT](#) exports matrix data from Gekko to R. If you need to export timeseries data, you must first convert the timeseries to matrices by means of the [pack\(\)](#) [function](#).

Syntax

```
R_EXPORT < TARGET=... > matrix1, matrix2, ... ;
```

TARGET =

(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)

Example

To export the matrices `#m1` and `#m2`, inserting them at the top of the R file (pointed to in the [R_FILE](#) command), you may use the following:

```
R_EXPORT #m1, #m2;
```

If you need to insert the matrices at a particular location in the R file, you may use the following:

```
R_EXPORT <target = 'data1'> #m1, #m2;
```

This will insert the matrix data at the exact location in the R file, where there is a line starting with `gekkoimport data1`. See a practical example under [R_RUN](#).

Note

You can also use [EXPORT](#)<r> to export matrices to a file suitable for R.

Related commands

[R_FILE](#), [R_RUN](#), [OLS](#), [MATRIX](#)

3.63 R_FILE

The R interface has three logical parts: [R_FILE](#), [R_EXPORT](#), and [R_RUN](#), and these commands should appear in that order. The `R_FILE` command points to an existing R file, into which data is inserted. `R_EXPORT` inserts Gekko data (matrices) into the R file. And `R_RUN` runs the R file, and returns R-matrices from R back to Gekko.

The `R_FILE` command points to an existing R file (with extension `.r`) to be used when R is called.

Syntax

```
R_FILE filename ;
```

filename

Filenames may be contain an absolute path like `c:\projects\gekko\myfile`, a relative path like `\gekko\myfile.gbk`, or be stated without a path. Filenames containing blanks and special characters should be put inside quotes. See more on filenames [here](#).

Example

To initiate a R session with the R file `ols.r` located in the working folder, use:

```
R_FILE ols.r;
```

Inside this `.r` file, you can now use special `gekkoimport` and `gekkoexport` statements, please see the practical example under [R_RUN](#).

Related options

[R_EXPORT](#), [R_RUN](#), [OLS](#), [MATRIX](#)

3.64 R_RUN

The R interface has three logical parts: [R_FILE](#), [R_EXPORT](#), and [R_RUN](#), and these commands should appear in that order. The [R_FILE](#) command points to an existing R file, into which data is inserted. [R_EXPORT](#) inserts Gekko data (matrices) into the R file. And [R_RUN](#) runs the R file, and returns R-matrices from R back to Gekko.

So the [R_RUN](#) command starts up R, runs (in R) the R file pointed to by the [R_FILE](#) command, and returns to Gekko while obtaining back matrix data from R.

If you just need to export matrices to R, try the [EXPORT](#)<r> command.

Syntax

R_RUN <MUTE> ;

MUTE

(Optional). With this option set, R is run silently in Gekko. Alternatively, R output is shown in the Gekko main window.

Examples

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.

First, put the following R file into your working folder:

```
----- ols.r
-----
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
-----
```

Next, you can run the following program in Gekko:

```

RESET; CLS;
CREATE lna1, pcp, bull;
SERIES <1998 2010> lna1 = 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') ;
SERIES <1998 2010> pcp = 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') ;
SERIES <1998 2010> bull = 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;
CREATE s0, s1, s2, s3, s4;
SERIES s0 = dlog(lna1);
SERIES s1 = dlog(pcp);
SERIES s2 = dlog(pcp.1);
SERIES s3 = bull;
SERIES s4 = bull.1;
MATRIX #x = pack(%t1, %t2, s1, s2, s3, s4);
MATRIX #y = pack(%t1, %t2, s0);
R_FILE ols.r;
R_EXPORT <target = 'data1'> #x, #y; //puts the data into the
'gekkoimport data1' section in ols.r
R_RUN; //returns matrices #beta and
#yfit from R
PRT #beta;
SERIES s0fit = #yfit[... 1].unpack(%t1, %t2);
PLOT s0, s0fit;

```

The program prints R output on the screen, and plots actual and predicted values. 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 in `R_EXPORT` 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. This interface is more stable than COM-based automation, and interchange of values, text, etc. could also be provided.

Note

Note that with `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.

Related options

[OPTION](#) r exe folder = ... ; //you may use this, if the auto-detection of the location of R fails

Related commands

[R_FILE](#), [R_EXPORT](#), [OLS](#), [MATRIX](#), [EXPORT](#)<r>

3.65 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).

Syntax

REBASE < BANK=... PREFIX=... INDEX=... > *variables* *date1* *date2* ;

<i>variables</i>	A list of variable names (may include bank names)
<i>date1</i>	Starting date for the observations.
<i>date2</i>	(Optional). Ending date for the observations. If not indicated, <i>date1</i> is used as ending date.
BANK	(Optional). A databank name indicating where the timeseries are located.
PREFIX	(Optional). A prefix name for 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.

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;`.

Example

```
RESET;
MODE data;
TIME 2010 2012;
y = 2, 3, 4;
REBASE <prefix=i1> y 2011;
```

```
PRT <n> y, i1y;  
REBASE <prefix=i2> y 2011 2012;  
PRT <n> y, i2y;
```

The result is the following:

Rebased 1 variables

	y	i1y
2010	2.0000	66.6667
2011	3.0000	100.0000
2012	4.0000	133.3333

Rebased 1 variables

	y	i2y
2010	2.0000	57.1429
2011	3.0000	85.7143
2012	4.0000	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).

Related commands

[SERIES](#), [SPLICE](#)

3.66 READ

The READ command 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 commands, and are in that sense long-lived.

It should be noted that if a model has been loaded, the READ command will auto-create any model variables not present in the .gbk file (and fill these variables with missing values). Because of this, in command files it may often be convenient to put the MODEL statement before the READ statement.

READ is intended for .gbk files, and can be thought of as a strong 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>`.

Syntax

```
READ < period ALL FIRST REF MERGE > filename TO bankname;
```

<i>filename</i>	<p>Filenames may contain an absolute path like <code>c:\projects\gekko\myfile</code>, a relative path like <code>\gekko\myfile.gbk</code>, or be stated without a path. Filenames containing blanks and special characters should be put inside quotes. See more on filenames here. If the filename is set to <code>'*'</code>, you will be asked to choose the file in Windows Explorer. The extension .gbk is automatically added, if it is missing.</p>
<i>period</i>	<p>(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.</p>

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 <code>TO bankname</code> is indicated, Gekko will put the data into a separate 'named' databank alongside the Work and Ref databanks. For instance, after <code>READ adambk TO a;</code> , you may refer to the variables by means of colon, for instance <code>PRT a:var1;</code> . If you use <code>READ adambk TO *;</code> , 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 (<code>READ ... TO ...</code> is essentially the same as an OPEN command)

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 exact 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 command.

Note

When reading, extension `.gbk` is automatically added if it is missing. Global time settings do not affect the READ command, 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 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) and comes in the following versions:

gbk file format versions

1.0	(July 2011). The file extension is <code>.tsdx</code> . Inside this zip-file there is a <code>.tsd</code> 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 <code>.tsdx</code> or <code>.gbk</code> . Inside there is a binary protobuf file (either with extension <code>.bin</code> (older) or <code>.data</code> (newer)), and a xml file with meta-information, where <code>databankVersion = "1.1"</code> . Only supports timeseries. Can be read by Gekko 1.5.8 and later.
1.2	(November 2018). The file extension is <code>.gbk</code> . Inside there is a binary protobuf file (<code>databank.data</code>) and a xml file with meta-information (<code>DatabankInfo.xml</code>), where <code>databankVersion = "1.2"</code> . 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 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 `.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).

Related options

[OPTION](#) databank create auto = no; [yes|no]

[OPTION](#) databank create message = yes; [yes|no]

[OPTION](#) databank file copylocal = yes;

[OPTION](#) databank file gbk version = 1.1; [1.0|1.1]

[OPTION](#) folder bank = [empty];

[OPTION](#) folder bank1 = [empty];

[OPTION](#) folder bank2 = [empty];

[OPTION](#) solve data create auto = yes; [yes|no]

Related commands

[IMPORT](#), [EXPORT](#), [WRITE](#), [OPEN](#), [CLONE](#), [DOWNLOAD](#)

3.67 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 command, so you may for instance use the shorter `a*b` instead of `{'a*b'}`.

Syntax

```
RENAME < BANK=... > variables1 AS variables2;
```

BANK=	(Optional). A databank name indicating where the timeseries are located.
<i>variables1</i>	Variable name(s) or list(s)
<i>variables2</i>	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.

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](#) command for more examples. RENAME is in reality similar to a COPY where the original object is deleted after copying. Also see the [INDEX](#) command 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 commands

[COPY](#), [DELETE](#)

3.68 RESET

The RESET command 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).

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](#) command, 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](#) command can be used to run Gekko.ini files after a RESET (RESET followed by INI is equivalent to [RESTART](#)).

Related commands

[RESTART](#), [INI](#), [DELETE](#), [CLOSE](#)

3.69 RESTART

The RESTART command 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. The gekko.ini files may reload models, databanks, options, etc. (use [RESET](#) to avoid doing that).

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.

NOTE: RESTART is equivalent to `RESET; INI;`, and can be convenient in interactive sessions, where a RESTART command 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 command files, the user may put `RESET; RUN gekko.ini;` as the first line in the user's main Gekko command 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 commands 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](#) command first, set the [mode](#), and then [OPTION](#) commands including time settings. Finally [MODEL](#) and [READ](#) commands (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` command 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 commands (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](#) command clears up in the same way as RESTART, but will skip any existing `gekko.ini` file.

The [INI](#) command 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` command file in the working folder (cf. [OPTION](#)).

Related commands

[RESET](#), [INI](#), [DELETE](#), [CLOSE](#)

3.70 RETURN

RETURN returns from a command file or [function/procedure](#) (i.e., does not execute the remainder of the file). It will return to any 'parent' command file calling that particular 'child' command file (or to the command prompt if there are no 'mother' command files). If you wish to return from all command files at once, use the STOP command instead.

Syntax

```
RETURN ; //return from command file
```

If RETURN is used to return from a [FUNCTION](#) (that is, if the function returns one or more variables), it must use the following syntax:

```
RETURN expression ;
```

Note

If you wish to comment out a section of the command 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 commands

[STOP](#), [EXIT](#)

3.71 RUN

The RUN command runs a file Gekko commands (also called a command file). The default extension for command files is .gcm. You may also run commands from outside Gekko, either via `gekko.exe` or via remote control (cf. below).

See also [PROCEDURE](#), which can be thought of as a command file that also accepts arguments. If you need to call a command file many times, for instance from inside of a loop, calling a procedure instead will often run much faster because re-parsing and re-compiling can be skipped.

Syntax

RUN filename;

*filenam
e*

Filenames may be contain an absolute path like `c:\projects\gekko\myfile`, a relative path like `\gekko\myfile.gbk`, or be stated without a path. Filenames containing blanks and special characters should be put inside quotes. 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 particular line, try inserting a line with a non-existing function like for instance `stop();`. This will abort the program in a clean way and make it possible to inspect variables etc.

Example

You may run the command 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](#) command).

gekko.exe parameters

This is for more advanced users, but you may start up gekko.exe with parameters. Gekko accepts commands as arguments, so for instance Gekko can be started up like this (execute this from the system shell where gekko.exe is located, or from the .bat file starting up Gekko):

```
gekko.exe -folder:c:\adam\2019 -noini RUN scenario;
```

With the first parameter `-folder:` you can indicate the starting folder when starting up gekko.exe this way. Please avoid blanks after the `-folder:`, and if the path contains a blank, enclose the path in quotes ("`c:\my folder\adam`"). With the `-noini` parameter you can indicate that you do not wish any gekko.ini files to run when starting up Gekko. After this, you may write Gekko statements separated by semicolons (';'). So above, we are starting up Gekko in the `C:\adam\2010` folder, without running a possible ini file, and executing the statement `RUN scenario;` (that is, running the file `scenario.gcm` in the working folder). You may stop Gekko after running `scenario.gcm` by issuing an [EXIT](#) statement, for instance:

```
gekko.exe -folder:c:\adam\2010 -noini RUN scenario; EXIT;
```

Or alternatively end `scenario.gcm` with an EXIT statement. This way, the session can run completely without user intervention.

If you do not state a `-noini`, Gekko will first run the gekko.ini file, and then run the statements given as arguments to gekko.exe. [New in 3.0.3]

Remote control

With `OPTION interface remote = yes;`, Gekko may be remote-controlled from a special `remote.gcm` command 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

In older Gekko versions, command files typically had extension `.cmd`. In the versions leading up to Gekko 2.0, the extension was `.gek`.

If a Gekko `.gcm` file fails mysteriously, try setting the technical `OPTION code split = 0;` (this will cost a little bit of speed).

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).

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 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 option system code split = 10; //If Gekko fails mysteriously,
try setting this to 0 (costs a little bit of speed)
```

Related commands

[RETURN](#), [STOP](#), [PROCEDURE](#)

3.72 SERIES

The SERIES (or SER) command 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`.

Compatibility note regarding lags:

Since series calculations are treated more like vector operations in Gekko 3.0, lags no longer accumulate period-for-period, if the left-hand side variable is present with lags on the right-hand side (so-called "lagged endogenous"). So a series expression like `x = x[-1] + 1;` no longer accumulates automatically (augments `x` with 1 for each period); instead the alternatives `x ^= 1;` or `x <d>= 1;` could be used. If accumulating behavior is needed, the `<dyn>` option can be set, for instance `x <dyn> = x[-1] + 1;`, or for several series statements a [block](#) structure can be used: `BLOCK series dyn = yes; ... ; END;`. Setting dynamic mode affects speed negatively, and should therefore not be set unless needed.

Gekko has two kinds of timeseries: **normal** series and **array-series**. Array-series allow the use of multidimensional indexes, for instance `x['a', 'b']`, picking out a subseries 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.

Syntax

```
SERIES <period operator KEEP=... LABEL=... SOURCE=... UNITS=...
STAMP=... DYN MISSING=...> variable = 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/count series in open databanks
```

SERIES	SERIES keyword may be omitted
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, SERIES <2020 2025 m keep=p> x = 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 observations is the same as before the update.
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 endogenous variables like in the expression $x = x[-1] + 1$; accumulate over time. Entails a speed penalty, so please do not use if not needed ($x \wedge = 1$; or <d> $x = 1$; could be used instead).

MISSING=	(Optional). With <code><missing = ignore></code> , SERIES will deal with missing array subseries and missing data values like GAMS, treating them as zero for sums and mathematical expressions. The following options are set locally and reverted afterwards: <code>option series array calc missing = zero;</code> <code>option series data missing = zero.</code> 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](#)).

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;
```

The right-hand side in this example is a [list](#) of values. You can indicate a local time period in the <>-option field:

```
TIME 2020 2030;
SERIES <2021 2023 label='Gekko-variable'> x = 100, 110, 90;
<2021 2023 label='Gekko-variable'> x = 100, 110, 90; //the same:
the SERIES keyword may be dropped
x <2021 2023 label='Gekko-variable'> = 100, 110, 90; //the same:
the option field may be moved
```

The local time period overrules the global period. If the three values corresponded to quarters for a quarterly series x , the command $x!q = 100, 110, 90;$ 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 $y = 1, 2 \text{ rep } 2, 3;$ 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: $y <2021 2025> = 1, 2, 3 \text{ rep } *;$, 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):

	xa	b	xb	a	b	xb
2010	1.0000	200.0000	2.0000	100.0000	200.0000	2.0000
2011	1.0000	200.0000	2.0000	100.0000	200.0000	2.0000
2012	1.0000	200.0000	2.0000	100.0000	200.0000	2.0000

Array-series

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	<code>^=</code>	<code>x ^= 1200;</code>	<code>x = x[-1] + 1200</code>	Same as <code><d></code> . or <code>dif(x) = 1200;</code> . See also the <code><dyn></code> option.

Relative	<code>%=</code>	<code>x %= 3.5;</code>	<code>x = x[-1] * (1+3.5/100)</code>	Same as <code><p></code> or <code>pch(x) = 3.5;</code> . See also the <code><dyn></code> option.
Absolute	<code>+=</code>	<code>x += 1200;</code>	<code>x = x + 1200</code>	Same as <code><m></code> . You can also use <code>-=</code> to subtract values.
Relative	<code>*=</code>	<code>x *= 1.03;</code>	<code>x = x*1.03</code>	Similar to <code><q></code> . You can also use <code>/=</code> to divide with values.
Change in relative	<code>#=</code>	<code>x #= 2.1;</code>	<code>x = x[-1]*(x0/x0[-1] + 2.1/100)</code>	Same as <code><mp></code> . See also the <code><dyn></code> 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	<code><d></code>	<code>x <d>= 1200;</code>	<code>x = x[-1] + 1200</code>	Same as <code>^=</code> . or <code>dif(x) = 1200;</code> . See also the <code><dyn></code> option.
Relative	<code><p></code>	<code>x <p>= 3.5;</code>	<code>x = x[-1] * (1+3.5/100)</code>	Same as <code>%=</code> or <code>pch(x) = 3.5;</code> . See also the <code><dyn></code> option.
Absolute	<code><m></code>	<code>x <m>= 1200;</code>	<code>x = x + 1200</code>	Same as <code>+=</code> . You can also use <code>-=</code> to subtract values.
Relative	<code><q></code>	<code>x <q>= 3;</code>	<code>x = x*(1+3/100)</code>	Similar to <code>*=</code> . You can also use <code>/=</code> to divide with values.
Change in relative	<code><mp></code>	<code>x <mp>= 2.1;</code>	<code>x = x[-1]*(x0/x0[-1] + 2.1/100)</code>	Same as <code>#=</code> . See also the <code><dyn></code> option.
Log	<code><l></code>	<code>x <l>= 5;</code>	<code>x = exp(5)</code>	Same as <code>log(x) = 5;</code> .
Relative	<code><dl></code>	<code>x <dl>= 0.035;</code>	<code>x = x[-1] * exp(0.035)</code>	Same as <code>dlog(x) = 0.035;</code> .

Left-side functions:

Type	Option	Example	Result	Note
Absolute	<code>dif()</code> <code>diff()</code>	<code>dif(x) = 1200;</code>	<code>x = x[-1] + 1200</code>	Same as <code>^=</code> or <code><d>=</code> . See also the <code><dyn></code> option.

				You may use diff() as synonym.
Relative	pch()	pch(x) = 3.5;	x = 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 = 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: 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 command is functionally equivalent to the following:

```
tg <2020 2020> = %v
```

The `$`-operator can be used after any expression, and works like an implicit [IF](#)-statement. For instance:

```
y = 3 $ ('b' in #m and %v == 10)
```

In this case, `y` will be 3 if 'b' is a member of `#m` and `%v` has the value 10. Else, `y` will obtain the value 0. The `$`-operator can be used to switch between values inside a period, for instance:

```
RESET; MODE data;
TIME 2001 2005;
x = 10, 10, 11, 12, 10;
y1 = 110 $ (x == 10) + 111 $ (x <> 10);
y2 = iif(x, '==', 10, 110, 111);
```

The second-last SERIES (`y1`) illustrates the use of the `$`-operator for switching, and `y1` will contain the numbers 110, 110, 111, 111, 110 (the 10's are replaced with 110, and all other values are replaced with 111). The last SERIES (`y2`) illustrates how to perform the same operation using the `iif()` function. The operation could alternatively be performed with [FOR](#) and [IF](#) statements, looping explicitly over each period, but using the `$`-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 curlyes, #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 curlyes, the expression would fail, since a list does not implement the *= operator. Finally, in PRT {#m};, without the curlyes, 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](#) command), 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](#) command.

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
```

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'.

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 (subseries) 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.

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 2.4 and before, so for example two series are added in one operation, similar to adding two vectors. This affects the use of lags in expressions with "lagged endogenous":

```
TIME 2021 2024;
x = 100;
x <2022 2024> = x[-1] + 1; //result: 101, 101, 101, not 101, 102,
103
```

Here, we might have expected 101, 102, 103, but lags do not accumulate like this in Gekko 3.0. In this case, `x` can be thought of as the vector `[100, 100, 100, 100]` over the period 2021-24. The lag of this vector is then `[M, 100, 100, 100]`, where the elements are shifted one position to the right, and where 'M' is missing value. Add 1 to this: `[M, 101, 101, 101]`, and it is seen why the result is 101 over the period 2022-24. To produce the 'right' result, use the `^=` operator or `<d>` instead:

```
TIME 2021 2024;
x = 100;
x <2022 2024> ^= 1; //result: 101, 102, 103. Alternatively, the
<d> operator could be used.
```

The 'problem' (surprises) with such dynamic definitions only appears when the left-hand side variable itself appears with a lag on the right-hand side. So an expression like `x = y/y[-1] - 1` has no such problems.

To mitigate this issue, for instance when copy-pasting model equations containing "lagged endogenous" variables in some of the equations. For such cases, the option `<dyn>` can be used, or a `block` like `BLOCK series dyn = yes; ...; END;` can be used. Please only use this when relevant: setting the option entails a speed penalty. Example:

```
TIME 2021 2024;
x = 100;
x <2022 2024 dyn> = x[-1] + 1; //result: 101, 102, 103
```

Note

See the page with [syntax diagrams](#) if the basics of names, expressions, etc. is confusing.

In general, you may put the option field to the left of, or to the right of the left-hand side variable. These variants are all equal: `SERIES <2010 2020> y = 100;` `SERIES y <2010 2020> = 100;` `<2010 2020> y = 100;` `y <2010 2020> = 100;` The last one is perhaps more readable than the second-last one.

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 command 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 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](#) freq = a; [a|q|m|u]

[OPTION](#) databank create auto = no; [yes|no]

[OPTION](#) series array calc missing = error; [error|m|zero]

[OPTION](#) series dyn = no; [yes|no]

[OPTION](#) series normal calc missing = error; [error|m|zero]

Related commands

[CREATE](#), [DOC](#), [PRT](#), [VAL](#), [EXPORT](#)<gcm>, [EXPORT](#)<flat>

3.73 SHEET

You can transfer variables to Excel by means of the SHEET command. SHEET has the same syntax as the [PRT](#), [PLOT](#) and [CLIP](#) commands, including the use of operators. 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 DATEFORMAT=... DATATYPE=... BANK=...
      REF=... MISSING=... > variables FILE=... ;
```

<i>period</i>	(Optional). Local period, for instance 2010 2020, 2010q1 2020q4 or %per1 %per2+1.
IMPORT	<p>(Optional). Use SHEET<import> to obtain data from Excel. (If you need to obtain data arranged in rows/columns with labels, see IMPORT<xlsx>).</p> <p>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 $n \times k$ 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.</p>

	<p>LIST, MAP or MATRIX: You can state one collection name like for instance <code>#m</code>. Additionally, you may use <code>SHEET=...</code>, <code>CELL=...</code>, <code>ROWS/COLS</code> (select the orientation of the data, <code>COLS</code> is transposed), and <code>FILE</code> (the filename). After this, Gekko will read the data into the given collection:</p> <ul style="list-style-type: none"> • 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, <code>#m[2][3]</code> will represent row 2, column 3 of the sheet (that is, the cell <code>c2</code>). 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, <code>#m['%c2']</code> or <code>#m.%c2</code> will represent the cell <code>c2</code> (stored as the scalar <code>%c2</code>). • Matrix: The data is loaded as a matrix, where all the cells must be of value type. If you use the <code>MISSING</code> 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 (blanks at beginning and end are removed). • List and map: [New in 3.0.6].
<i>operator</i>	(Optional). 'Long': <code>abs</code> , <code>dif</code> , <code>pch</code> , <code>gdif</code> , or 'short': <code>n</code> , <code>d</code> , <code>p</code> , <code>dp</code> , <code>m</code> , <code>q</code> , <code>mp</code> , <code>r</code> , <code>rd</code> , <code>rp</code> , <code>rdp</code>
TITLE	(Optional). A title for the sheet. You can use <code>HEADING</code> as alias.
STAMP	(Optional). If 'yes', a time stamp is inserted at the top
<i>variables</i>	Name of the variable(s) printed. Several variables can be printed at once using, <code>var1</code> , <code>var2</code> You may also use lists or expressions.
FILE	<p>(Optional). <code>SHEET</code> will optionally create an Excel file silently without opening Excel (the filename will be <code>[filename].xls</code> or <code>[filename].xlsx</code>, and is put into the Gekko working folder)</p> <p>Filenames may be contain an absolute path like <code>c:\projects\gekko\myfile</code>, a relative path like <code>\gekko\myfile.gbk</code>, or be stated without a path.</p>

	<p>Filenames containing blanks and special characters should be put inside quotes. See more on filenames here.</p>
SHEET	<p>The name of the sheet for your data, for instance 'Data1'</p>
CELL	<p>The cell where data starts, for instance 'C4', default is 'A1'.</p>
DATES	<p>[yes no]: If 'yes', dates are shown (is 'yes' per default)</p>
NAMES	<p>[yes no]: If 'yes', names are shown (is 'yes' per default)</p>
COLORS	<p>[yes no]: If 'yes', colors are shown (is 'yes' per default)</p>
ROWS	<p>[yes no]: If 'yes', the timeseries are printed in rows (default), use the COLS option to transpose.</p>
COLS	<p>[yes no]: If 'yes', the timeseries are printed in columns (transposed).</p>
APPEND	<p>[yes no]: If 'yes', the table is appended to an existing Excel workbook (is 'no' per default)</p>
MATRIX	<p>[yes no]: Used with SHEET<import> to import a matrix, see example below.</p>
MISSING	<p>[yes no]: Used with SHEET<import matrix>. Cells with no content are set to missing instead of 0.</p>
DATEFORMAT= DATETYPE=	<p>(Optional). These options control the date format for .xlsx and .csv files. DATEFORMAT can be either 'gekko' (default) 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 or monthly data that the last day of the quarter or month 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</p>

	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. [New in 3.0.5].
BANK	(Optional). A bankname where variables are looked up. For instance <code>PRT <bank = b1> x;</code> is equivalent to <code>PRT b1:x;</code> . See also <code><REF = ...></code> . These options can be convenient instead of opening and closing banks.
REF	(Optional). A bankname where reference variables are looked up. For instance <code>PRT <bank = b1 ref = b2 m> x;</code> uses banks <code>b1</code> and <code>b2</code> for the multiplier. See also <code><BANK = ...></code> . These options can be convenient instead of opening and losing banks.
MISSING=	(Optional). With <code><missing = ignore></code> , SHEET will deal with missing array subseries and missing data values like GAMS, treating them as zero for sums and mathematical expressions, or skipping the printing of a subseries if it does not exist. The following options are set locally and reverted afterwards: <code>option series array print missing = skip;</code> <code>option series array calc missing = zero;</code> <code>option series data missing = zero.</code> 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` command, you may also use element-specific operators (for instance `SHEET unemp, gdp<p>;`). See the [PRT](#) command 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 command 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 command 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 EXPORT to export a matrix to Excel, `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 become `pm011` if found under imports, else `pe011`, and in addition we wish 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](#)) commands 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](#) 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 commands

[CLIP](#), [PRT](#), [PLOT](#), [EXPORT](#)

3.74 SIGN

This command 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 command 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 accidentally 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 $2^{(-128)} = 2.9\text{E-}39$ (that is, effectively zero).

Syntax

SIGN;

Examples

To obtain a signature for a (unsigned) model loaded with the [MODEL](#) command, 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 command.

Note

You may put other meta-information into the model file (.frm). As of now, `Info`, `Date`, and `Signature` fields are supported. For instance:

```
// Info: Model used for forecasting 2012-2030
// Date: 7-11-2012 15:37:00
// 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 command).

Related commands

[MODEL](#), [SIM](#)

3.75 SIM

SIM solves the 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](#) commands. 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](#) command), 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 >;

<i>period</i>	(Optional). Local period, for instance <code>2010 2020, 2010q1 2020q4</code> or <code>%per1 %per2+1</code> .
FIX	Tells Gekko to enforce any goals/means stated by the ENDO/EXO commands. It is mandatory to use <code>SIM<fix></code> 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 <code>OPTION solve static = yes</code> .
RES	The <code>SIM<res></code> command (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.

	<p>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. <code>SIM<res></code> 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).</p>
AFTER	<p><code>SIM<after></code> 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 <code>AFTER;</code> or <code>AFTER\$</code> line in a model file).</p> <p>Since Y-type variables are typically also J- and Z-variables, one can think of <code>SIM<after></code> as a way to compute all reversed J- and Z-variables independently of the SIM command (the SIM command computes these after simulation), in addition to computing table variables. See the MODEL command for more on J- and Z-variables.</p> <p>On a well-specified model, reading a historical databank, and issuing a <code>SIM<after></code> statement on the historical period should ideally make it possible to simulate on the historical period and replicate the endogenous variables.</p>

- 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](#) commands 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>` command (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 commands

[MODEL](#), [EXO](#), [ENDO](#), [READ](#), [WRITE](#), [CLONE](#), [MULPRT](#)

3.76 SMOOTH

SMOOTH replaces missing values inside a timeseries with values generated by means of a particular (user-chosen) method.

Syntax

SMOOTH <period> var1 = var2 type;

period	(Optional). Local period, for instance 2010 2020, 2010q1 2020q4 or %per1 %per2+1.
var1	The new corrected variable
var2	The variable that contains missings/holes
type	The type of smoothing, choose between: <ul style="list-style-type: none"> • LINEAR. Linear interpolation. • GEOMETRIC. Geometric interpolation. • REPEAT. Repeats last known observation. • SPLINE. Cubic splines. • OVERLAY. Insert another series into the holes.

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](#)).

Example

For instance:

```
CREATE ts, ts1; //only necessary in sim-mode
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;
```

As you can see, the timeseries `ts` has a hole in the middle, namely the observations 2005-2007 (inclusive). Using the `SMOOTH` command, these three observations are filled out. Below, the four different interpolation methods are shown (interpolated values in red):

	ts4	ts	ts5	ts1	ts2	ts3
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	6.1349	M	-2.0000	6.0000	5.2643	4.0000
2006	8.8696	M	-3.0000	8.0000	6.9282	4.0000
2007	11.1694	M	-4.0000	10.0000	9.1180	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` method, note that the growth rate of `ts2` is constant (31.61%) in the three interpolated years. In this case, `SPLINE` provides the most realistic hole-filling, since it takes the curvature of the `ts` timeseries into consideration.

Note

See the `hpfiler()` [function](#) regarding the smoothing of timeseries without holes.

Related options

[OPTION](#) `calc ignoremissingvars = no; [yes|no]`

Related commands

[SPLICE](#)

3.77 SPLICE

SPLICE is used to combine two [timeseries](#) into one.

Syntax

```
SPLICE var = var1 var2 ;
SPLICE var = var1 period var2 ;
```

var	The new timeseries made from the variables on the right-hand side of <code>=</code> .
var1, var2	The two timeseries that are to be spliced into the one stated on the left-hand side of <code>=</code> .
period	(Optional). The period that should be used as overlap to splice the two timeseries.

- 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](#)).

Example

The following example illustrates the use of SPLICE:

```
TIME 2002 2010;
CREATE ts1, ts2, ts0a, ts0b; //only necessary in sim-mode
ts1 <2002 2006> = 2, 3, 4, 5, 6;
ts2 <2004 2010> = 41, 42, 43, 44, 45, 46, 47;
SPLICE ts0a = ts1 ts2;
SPLICE ts0b = ts1 2006 ts2;
PRT <n> ts1, ts2, ts0a, ts0b;
PRT <p> ts1, ts2, ts0a, ts0b;
```

In this case, the `ts1` observations up to and including 2006 are used, whereas the observations from 2007-2010 are generated by multiplying the `ts2` values with a correction factor (so that their levels fit with the levels of `ts1`).

In the first SPLICE, all three common observations (2004, 2005 and 2006) are used to create the correction factor, whereas in the second SPLICE, only 2006 is used to compute the correction factor.

Levels:

	ts1	ts2	ts0a	ts0b
2002	2.0000	M	16.8000	14.3333
2003	3.0000	M	25.2000	21.5000
2004	4.0000	41.0000	41.0000	28.6667
2005	5.0000	42.0000	42.0000	35.8333
2006	6.0000	43.0000	43.0000	43.0000
2007	M	44.0000	44.0000	44.0000
2008	M	45.0000	45.0000	45.0000
2009	M	46.0000	46.0000	46.0000
2010	M	47.0000	47.0000	47.0000

Percentage growth:

	ts1	ts2	ts0a	ts0b
2002	M	M	M	M
2003	50.00	M	50.00	50.00
2004	33.33	M	62.70	33.33
2005	25.00	2.44	2.44	25.00
2006	20.00	2.38	2.38	20.00
2007	M	2.33	2.33	2.33
2008	M	2.27	2.27	2.27
2009	M	2.22	2.22	2.22
2010	M	2.17	2.17	2.17

Related commands

[SMOOTH](#)

3.78 STOP

The command returns from all command files immediately, returning to the command prompt. This command can be used for debugging, if the user wishes to run a system of command files up to a specific point, and examine the results up to that specific point.

To stop/abort a program while it is running, you can use the red stop button in the user interface.

Syntax

```
STOP ;
```

Example

```
STOP;
```

The similar [RETURN](#) command does not return from all command files. The [EXIT](#) command effectively issues a STOP, and then afterwards closes the Gekko application.

Related commands

[RETURN](#), [EXIT](#)

3.79 STRING

The STRING command 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	<pre>%s = 'abcd'; PRT %s[2]; // 'b'</pre>
[x1..x2]-index	Index: returns the range of characters from position x1 to x2 (both inclusive). You may omit x1 or x2. Returns: string	<pre>%s = 'abcd'; PRT %s[2..3]; // 'bc'</pre>
concat(s1, s2)	Appends the two strings: same as s1 + s2. Returns: string	<pre>%s = concat('He', 'llo'); Result: 'Hello'.</pre>
endswith(s1, s2)	Returns 1 if the string s1 starts with the string s2, else 0. The comparison is case-insensitive. Returns: val	<pre>%v = endswith('abcde', 'cde'); Returns: 1</pre>
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	<pre>%v = index('onetwothreetwo', 'two'); Returns: 4. %v = index('oneTWO', 'two'); Returns: 4.</pre>
isAlpha(s)	Returns 1 if all the characters are letters (alphabet). [New in 3.0.5].	<pre>%v = isAlpha('aBc'); Returns: 1</pre>
isLower(s)	Returns 1 if the string contains no uppercase characters. [New in 3.0.5].	<pre>%v = isLower('abc12'); Returns: 1</pre>
isNumeric(s)	Returns 1 if all the characters are of numeric value. [New in 3.0.5].	<pre>%v = isNumeric('123'); Returns: 1</pre>
isUpper(s)	Returns 1 if the string contains no lowercase characters. [New in 3.0.5].	<pre>%v = isUpper('ABC12'); Returns: 1</pre>

length(s)	The length of the string (number of characters). You may use len() instead of length(). Returns: val	<pre>%v = %s.length();</pre>
lower(s)	The string in lower-case letters. Returns: string	<pre>%s = lower('aBcD'); Result: 'abcd'.</pre>
prefix(s1, s2)	If s1 is a string, it has the string s2 prefixed (prepended). Returns: string	<pre>%s1 = %s2.prefix('a');</pre>
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	<pre>%s = replace(%s1, %s2); //or: replace(%s, %s1, %s2)</pre>
split(s1, s2) 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, 1 per default (set to 0 or 1), see examples. [New in 3.0.6]	<pre>%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')</pre>
startswith(s1, s2)	Returns 1 if the string s1 starts with the string s2,	<pre>%s = 'abcde'; %v = %s.startswith('abc'); Returns: 1</pre>

	else 0. The comparison is case-insensitive. Returns: val	
strip(s)	Removes blank characters from the start and end of the string. Returns: string	<pre>%s1 = %s2.strip(); //or: strip(%s1)</pre>
stripstart(s)	Removes blank characters from the start of the string. Returns: string	<pre>%s1 = %s2.stripstart(); //or: stripstart(%s1, %s2)</pre>
stripend(s)	Removes blank characters from the end of the string. Returns: string	<pre>%s1 = %s2.stripend(); //or: stripend(%s1, %s2)</pre>
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	<pre>%s = %s1.substring(3, 2); //or: substring(%s1, 3, 2) %s = %s1[3 .. 5]; //a slice from pos 3 to 5 (both inclusive)</pre>
suffix(s1, s2)	If s1 is a string, it has the string s2 suffixed (appended) Returns: string	<pre>%s1 = %s2.suffix('a');</pre>
upper(s)	The string with upper-case letters. Returns: string	<pre>%s = upper('aBcD'); Result: 'ABCD'.</pre>

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](#) command (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, prepend 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](#) chapter (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 commands

[DATE](#), [VAL](#)

3.80 SYS

The SYS command provides access to the system shell (in Windows sometimes called the DOS prompt). Gekko will wait until the SYS command finishes before it continues.

Syntax

SYS **<MUTE>** **commands** ;

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.
commands	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

The SYS command can be used for file managing:

```
SYS 'copy file1.txt file2.txt';
```

or for more complex tasks or jobs

```
SYS 'start /wait gnuplot pl.prg';
```

If you need to use a percent symbol that may look like a scalar variable, prepend the symbol '~' to indicate that Gekko should not try to in-substitute the scalar. For example:

```
SYS 'set path3 = ~%path2% \"source\"';
```

If ~ is not used, Gekko will complain that the scalar %path2 cannot be found.

```
SYS;
```

Opens up the system shell in a separate window.

Note

Note the use of single quotes when using SYS.

Scalar variables will be in-substituted in a SYS-command, unless a '~' is prepended before the '%'. Cf. also the [TELL](#) command.

3.81 TABLE

The TABLE command 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. 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](#) command.

Syntax

TABLE < *period* *operator* HTML WINDOW=main > *tablename* ;

<i>period</i>	(Optional). Time period.
<i>operator</i>	(Optional). Can be <code>m</code> for multiplier or <code>r</code> for Ref databank. The <code>m</code> code works as if two tables were subtracted, one made with no operator ('Work'), and one made with <code>r</code> operator (Ref). No other codes are possible at the moment (operator <code>n</code> can also be used, but is the same as omitting an operator).
HTML	(Optional). If <code>html</code> 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 <code>WINDOW=main</code> 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.
<i>tablename</i>	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
|
-----
+-----+-----+-----+-----+
-
|      2009      2010 |      2005      2006      2007      2008
+-----+-----+-----+-----+
-
| 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>
      <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 definition 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 columns, so the vertical border between column 2 and 3 will be hidden in this row. Next, there is a `<rowborder>`, and then the date row begins. This row is filled with two empty `<txt>` items (think of these as 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 `<colborderhide>inner<colborderhide>`, or `<colborderhide>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: `<colborderhide>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 containing data, whereas the *attributes* (for instance `txtalign="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="%perl"/>` 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</text> <text>BBB</text>` would make 'BBB' appear in the fourth column, so the `<text colspan="3">` tag really represents three `<text>` 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 `<text>`. Horizontal alignment of text in a `<text>` 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](#) commands).
- `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](#) command (if used, choose 'View' --> 'Expand All' to unfold all XML nodes). You may also use Notepad (cf. the [EDIT](#) command), 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 `<`, `>`, and `&`).

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 command 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 excel 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 [functions](#) `bankfilename()` and `banktime()` may 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 ` ` (if written in a text editor like Notepad), for instance three blanks: `<txt>Gdp </txt>`. If the code is written via [XEDIT](#), you should write the code as ` `. 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 commands

[MENU](#), [PLOT](#), [XEDIT](#), [EDIT](#)

3.82 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 commands

[GOTO](#)

3.83 TELL

The TELL command 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 < NOCR > message ;

message	(Optional). The text to print (remember single quotes). Can be an expression that can be evaluated as a string.
NOCR	(Optional). Indicates that no newline is inserted (no carriage return)

Examples

The TELL command 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.';
```

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, prepend 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](#) chapter (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 commands

[DISP](#), [PRT](#)

3.84 TIME

The TIME command sets global time used for subsequent time series operations etc. The command works for frequencies annual, quarterly or monthly (and undated). The TIME command is the only way to change global time.

Syntax

```
TIME periods ;  
TIME period ;  
TIME ? ;
```

<i>periods</i>	<i>per1 per2</i>
<i>per1</i>	Start period, for instance 2010 or 2010q1
<i>per2</i>	End period, for instance 2012 or 2012q4
<i>period</i>	A single date that will be used as both start and end period.
<i>?</i>	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;
```

Commands like [SERIES](#), [PRT](#) etc. respect global time if no time period is indicated in the local <>-option field. In all the relevant commands (i.e. all commands operating on time series), a local time period can be set inside the <> brackets, operating on only the particular command 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 command:

```
PRT <2000m2 2005m12> fy, enl;
```

When the frequency is quarterly or monthly, 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`.

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 translates 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](#) commands.

The [TIMEFILTER](#) command 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 commands

[TIMEFILTER](#), [COLLAPSE](#), [BLOCK](#), `OPTION freq = ...`

3.85 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 command TIMEFILTER is used to indicate periods that are to be omitted in output from TABLE, DISP, PRT and MULPRT. The commands PLOT, SHEET and CLIP are not affected by the filter.

Syntax

TIMEFILTER *filterperiods* ;

<i>filterperiods</i>	<i>periods, periods, ...</i> (note the comma)
<i>periods</i>	<i>singleperiod</i> <i>periodlist</i>
<i>singleperiod</i>	Single observation
<i>periodlist</i>	<i>singleperiod</i> .. <i>singleperiod</i> BY <i>step</i> (if you prefer, you may use TO instead of .., and STEP instead of BY)
<i>step</i>	An optional stepsize (default step: 1). Must be integer >= 1. Omit BY <i>step</i> 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 command, 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 command, without having to change the global options.

Note

TIMEFILTER does not alter the TIME period settings. Note that this command 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 equivalent (and the latter is more similar to the FOR loop over DATES).

Related commands

[TIME](#), [PRT](#)

3.86 TRANSLATE

The command translates command files (.cmd) from AREMOS or older versions of Gekko.

- Translate from AREMOS to Gekko 3.0. Details [here](#).
- Translate from Gekko 1.8 to Gekko 2.0. 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 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 GEKKO18 GEKKO20 > 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.
GEKKO18	(Optional). Translates command files (.cmd) from Gekko 1.8 syntax to Gekko 2.0 syntax. NOTE: the translator is <i>not</i> intended for .frm files (model files).
GEKKO20	(Optional). Translates command files (.gcm) from Gekko 2.0/2.2/2.4 syntax to Gekko 3.0 syntax. NOTE: the translator is <i>not</i> intended for .frm files (model files). If you prefer to omit parentheses around lists of numbers in SERIES statements, you can use TRANSLATE<remove> to remove these afterwards.
MOVE	(Optional). A simple translator that moves the <>-field in assignments (typically SERIES commands). 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 command.
REMOVE	(Optional). A simple translator that removes superfluous ()-parenteses around a comma-separated list of numbers in assignments (typically SERIES commands). For instance: y = (2,

	<code>3, 4);</code> becomes <code>y = 2, 3, 4;</code> . This translator does not touch anything else, so it is for existing 3.0 files where the user would like to remove such parentheses.
<i>filename</i>	Filenames may contain an absolute path like <code>c:\projects\gekko\myfile</code> , a relative path like <code>\gekko\myfile.gbk</code> , or be stated without a path. Filenames containing blanks and special characters should be put inside quotes. 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 2.0/3.0, 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).

In principle, a Gekko 1.8 program can be translated to 3.0 in two steps via the two Gekko translators. In that case, take extra care regarding the translated result.

Related commands

[RUN](#)

3.87 TRUNCATE

TRUNCATE shortens a timeseries, so that the observations outside the given period are discarded.

Syntax

TRUNCATE < *period* > *variables* ;

<i>period</i>	(Optional). Local period, for instance 2010 2020, 2010q1 2020q4 or %per1 %per2+1.
<i>variables</i>	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.

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 commands

[SERIES](#)

3.88 UNFIX

UNFIX works in conjunction with SIM<fix>. It removes any goals/means set with the [EXO](#) and [ENDO](#) commands.

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 commands

[ENDO](#), [EXO](#), [SIM](#)

3.89 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 commands

[LOCK](#), [OPEN](#)

3.90 VAL

The VAL command 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 value 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 commands

[DATE](#), [STRING](#), [SERIES](#), [FOR](#), [IF](#)

3.91 VAR

VAR is a more general version of assignment types like [SERIES](#), [VAL](#), [DATE](#), [STRING](#), [LIST](#), [MAP](#) and [MATRIX](#). Such type indicators 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. So the VAR keyword is in a sense superfluous, since it can always just be omitted. If there are local options associated with the command, using VAR may look aesthetically more pleasing:

```
VAR <2020 2030> x = 100;
```

But still, the VAR may be omitted:

```
<2020 2030> x = 100;
```

If preferred, you may generally move the option field to the other side of the left-hand side variable:

```
x <2020 2030> = 100;
```

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); //will become a list of values
#x = 1, 2; //allowed for simple numbers
#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 commands

[SERIES](#), [VAL](#), [DATE](#), [STRING](#), [LIST](#), [MAP](#), [MATRIX](#)

3.92 WRITE

The command writes variables to a Gekko databank file (.gbk).

Syntax

WRITE < *period* **RESPECT** > *filename* ;
WRITE < *period* **RESPECT** > *variables* **FILE=filename** ;

<i>period</i>	(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.
<i>variables</i>	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.
<i>filename</i>	Filenames may be contain an absolute path like <code>c:\projects\gekko\myfile</code> , a relative path like <code>\gekko\myfile.gbk</code> , or be stated without a path. Filenames containing blanks and special characters should be put inside quotes. 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` command 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.

Related options

[OPTION](#) folder bank = [empty];

Related commands

[READ](#), [IMPORT](#), [EXPORT](#), [SHEET](#), [MODEL](#)

3.93 X12A

The X12A (X12-arma) performs seasonal adjustment on quarterly or monthly data. The component uses a well-known program from the US Census Bureau, and is similar to the AREMOS command with the same name. (Later on, X13-arma-seats and Tramo-Seats might be provided in Gekko, perhaps via the JDemetra+ project).

Syntax

X12A < period **PARAM=...** **BANK=...** > **variables** ;

<i>period</i>	(Optional). If not stated, Gekko will use the global time period.
<i>variables</i>	Variables and/or list(s). Wildcards may be used.
PARAM=	A text string containing parameter values for X12A.
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.

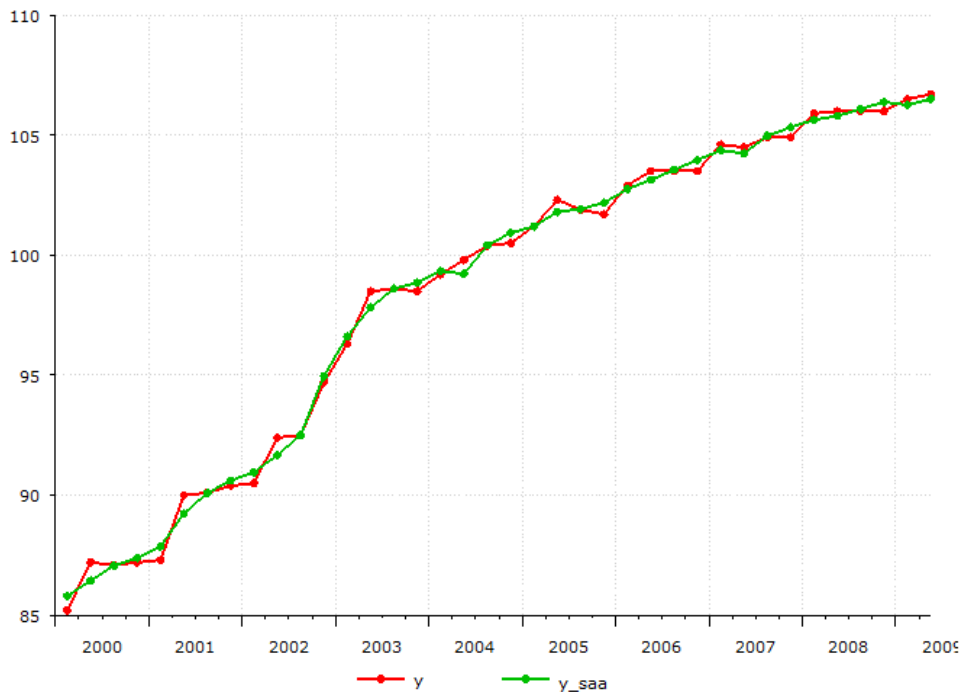
Example

You may try the following example:

```
RESET;
OPTION freq q;
CREATE y;
TIME 2000q1 2009q2;
SERIES 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 signalim=(1.50,2.50)
```

```
seasonalma=msr force=totals';  
X12A <param = %p> y ;  
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.

Note

For much more information on X12-arima, see <https://www.census.gov/ts/x12a/v03/x12adocV03.pdf>

Note that the frequency must be set right, before calling the X12A command.

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 this AREMOS command:

```
X12A y d10,d11,saa "mode=mult sigmalim=(1.50,2.50) seasonalma=msr  
force=totals print=alltables";
```

Related commands

[COLLAPSE](#), [SMOOTH](#)

3.94 XEDIT

The XEDIT command uses the open-source [XML Notepad](#) xml editor to open up the designated file. The command 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 around in XML Notepad.

Syntax

XEDIT *filename* ;

*filena
me*

Filenames may be contain an absolute path like `c:\projects\gekko\myfile`, a relative path like `\gekko\myfile.gbk`, or be stated without a path. Filenames containing blanks and special characters should be put inside quotes. 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 commands

[SYS](#), [EDIT](#), [PLOT](#), [TABLE](#)

Part IV

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](#) command.

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
<code>abs(x)</code>	Returns the absolute value of x (series, val or matrix). Returns: series/value/matrix	<code>%v1 = abs(%v2);</code>
<code>avg(x1, x2, ...)</code>	Returns the average of x_1, x_2, \dots etc. The input parameters may be series or value. Returns: series/value	<code>y = avg(x1, x2, x3);</code> <code>%v = avg(%v1, %v2, %v3);</code>
<code>avgt(x)</code> <code>avgt(<t1 t2>, x)</code>	Returns the time-average of the observations of the timeseries x over the local/global time period (or over t_1 to t_2 , if indicated) Returns: series	<code>y = avgt(x);</code> <code>y = avgt(<2020 2025>, x);</code> <code>y = x.avgt(<2020 2025>);</code> //same as above
<code>ceiling(x)</code>	Returns the the smallest integer which is greater than or equal to x (series, val or matrix). See also <code>int()</code> , <code>floor()</code> and <code>round()</code> . Returns: series/value/matrix	<code>PRT ceiling(-2.2);</code>
<code>dif(x)</code> or <code>diff(x)</code>	Absolute time-difference of series x : can also be used on left side of '='. Does not work on value. Returns: series	<code>y = dif(x);</code> <code>dif(y) = 100;</code>
<code>dify(x)</code> or <code>diffy(x)</code>	Yearly difference. Same as <code>dif(x)</code> , but will use 4 lags for quarterly data, and 12 lags for monthly data.	<code>y = dify(x);</code>

	Returns: series	
dlog(x)	Logarithmic time-difference of series x: can also be used on left side of '='. Does not work on value. Returns: series	<pre>y = dlog(x); dlog(y) = 0.02;</pre>
dlogy(x)	Yearly logarithmic time-difference. Same as dlog(x), but will use 4 lags for quarterly data, and 12 lags for monthly data. Returns: series	<pre>y = dlogy(x);</pre>
exp(x)	Returns the exponential value of x (series, value or matrix). Returns: series/value	<pre>y = exp(x); %v1 = exp(%v2);</pre>
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	<pre>PRT floor(-2.2);</pre>
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). You may alternatively use \$-conditionals, see examples under SERIES . See also the replace() function for series. Returns: series/value	<pre>TIME 2010 2012; in1 = 1, 2, 3; in2 = 3, 2, 1; y = iif(in1, '<= ', in2, 50, 100); Result: y = 50, 50, 100.</pre>

<code>int(x)</code>	Returns the integer value of <code>x</code> (series, val or matrix), discarding the fractional part (after the <code>.</code>). See also <code>floor()</code> , <code>ceiling()</code> and <code>round()</code> . Returns: series/value/matrix	<pre>PRT int(-2.2);</pre>
<code>isMiss(x)</code>	If the value <code>x</code> is missing, the function returns 1, else 0. Does not work if <code>x</code> is a series. Returns: value	<pre>PRT isMiss(1); PRT isMiss(miss());</pre>
<code>lag(x, lag)</code>	Lags series <code>x</code> a number of periods. Note the sign of the lag: <code>lag(x, 2) = x[-2]</code> . Can be used if <code>x</code> is an expression. Returns: series	<pre>y = lag(x, 2); //same as x[-2]</pre>
<code>log(x)</code>	Returns the natural logarithmic value of <code>x</code> (series, value or matrix). Can also be used on the left side of '='. Returns: series/value/matrix	<pre>y = log(x); %v1 = log(%v2); log(y) = a * log(b);</pre>
<code>movavg(x1, lags)</code>	Moving average of series <code>x1</code> . Returns: series	<pre>y = movavg(x, 3); y = (x + x[-1] + x[-2])/3; //same</pre>
<code>movsum(x1, lags)</code>	Moving sum of series <code>x1</code> , cf. <code>movavg()</code> . Returns: series	<pre>y = movsum(x, 3); y = x + x[-1] + x[-2]; //same</pre>
<code>pch(x)</code>	Percentage growth in series <code>x</code> : can also be used on left side of '='. Does not work on value. Returns: series	<pre>y = pch(x); pch(y) = 2;</pre>
<code>pchy(x)</code>	Yearly growth. Same as <code>pch(x)</code> , but will use 4 lags	<pre>y = pchy(x);</pre>

	for quarterly data, and 12 lags for monthly data. Returns: series	
<code>pow(x, y)</code>	The exponent must be a value or number, not a series. The function <code>pow(x, y)</code> is equal to <code>x**y</code> or <code>x^y</code> , that is, <code>x</code> in the <code>y</code> 'th power. You may use <code>power(x, y)</code> as synonym. Returns: series/value	<pre>y = pow(x1, %x2); %v = pow(%x1, %x2);</pre>
<code>rnorm(mean, var)</code> <code>rnorm(mean, vcov)</code>	Returns a random number from a normal distribution with mean and variance provided. If fed with a <code>nx1</code> matrix of averages, and a <code>nxn</code> covariance matrix, the function will return a <code>nx1</code> matrix of values. See also <code>rseed()</code> and <code>runif()</code> . Returns: value/matrix	<pre>%n = rnorm(0, 1); %n = rnorm(-100, 25**2); #n = rnorm(#mean, #vcovar);</pre>
<code>rseed(x)</code>	Given value <code>x</code> , it sets a random seed for <code>runif()</code> and <code>rnorm()</code> functions. The function returns the seed as a value. Returns: value	<pre>%v = rseed(12345);</pre>
<code>round(x, d)</code>	Rounds <code>x</code> (series, val or matrix) to <code>d</code> decimal places. See also <code>int()</code> , <code>floor()</code> and <code>ceiling()</code> . Returns: series/value/matrix	<pre>%v1 = round(%v2, 3);</pre>
<code>runif()</code>	Returns a random number from a uniform distribution between 0 and 1. See also <code>rseed()</code> and <code>rnorm()</code> . Returns: value	<pre>%v = runif();</pre>
<code>seq(start, end)</code>	Returns a list of integer values or dates between start and end (both included). Start and end	<pre>#m = seq(1, 100); #t = seq(2001q1, 2005q4);</pre>

	must be two values or two dates. Returns: list	
<code>sqrt(x)</code>	Returns the square root of <code>x</code> (series, value or matrix). Returns: series/value/matrix	<pre>y = sqrt(x); %v1 = sqrt(%v2);</pre>
<code>sum(x1, x2, ...)</code> <code>sum(list, x)</code>	Returns the sum of <code>x1</code> , <code>x2</code> , ... 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	<pre>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});</pre>
<code>sumt(x)</code> <code>sumt(<t1 t2>, x)</code>	Returns the time-sum of the observations of the timeseries <code>x</code> over the local/global time period (or over <code>t1</code> to <code>t2</code> , if indicated) Returns: series	<pre>y = sumt(x); y = sumt(<2020 2025>, x); y = x.sumt(<2020 2025>); //same as above</pre>

Conversions

Function name	Description	Examples
<code>date(x)</code>	Tries to convert the scalar <code>x</code> to date type. See also under "date combining functions". Returns: date	<pre>%d = date(2000+15); Result: %d = 2015.</pre>
<code>dates(x)</code>	Tries to convert each element of the list <code>x</code> into a date. Returns: list	<pre>#m1 = (2001, 2002, 2003); #m2 = dates(#m1); //or: #m1.dates()</pre>
<code>data(x)</code>	Converts a string <code>x</code> containing blank-separated numbers to a list of values.	<pre>#m = data('1 2 3'); x = data('1 2 3');</pre>

	Returns: list of values	
<code>format(x, code)</code>	<p>Formats the value/date/string <code>x</code> by means of the formatting code. The formatting code is as follows for values:</p> <pre>[width]: [format]' //note: no blanks</pre> <p>The width specifies that the string will be at least <code>[width]</code> characters wide. The <code>[format]</code> follows the conventions shown here or here, so you may either use a pattern like <code>0.000</code> or <code>0.###</code> (exactly three digits or at most three digits), or you may use a description like <code>F3</code> (floating point, three digits). So, <code>12:0.000</code> or <code>12:F3</code> both a 12 characters wide field, and a number with three decimals.</p> <p>If the <code>[width]</code> is positive, the number is right-aligned within the field, and if it is negative, it is left-aligned.</p> <p>You may also format strings or dates: in that case only the <code>[width]</code> is used (positive or negative). In this way, table-like alignment is quite straightforward.</p> <p>See also <code>OPTION string interpolate format val = ... ;</code> to set rules regarding <code>{...}</code> format in strings. Returns a string.</p>	<pre>%v = 12.3456; %d = 2020q1; %s = 'abc'; tell; tell '123456789012' + ' '; tell '-----' + ' '; tell format(%v, '0.000') + ' '; tell format(%v, '12:0.000') + ' '; tell format(%d, '12') + ' '; tell format(%s, '12') + ' '; tell format(%v, '-12:0.000') + ' '; tell format(%d, '-12') + ' '; tell format(%s, '-12') + ' '; tell '-----' + ' '; // 123456789012 // ----- // 12.346 // 12.346 // 2020q1 // abc // 12.346 // 2020q1 // abc // ----- </pre>
<code>string(x)</code>	Tries to convert the scalar <code>x</code> to string type. See also	<pre>%s = string(12) + string(34); Result: %s = '1234'.</pre>

	<p>under "string combining functions".</p> <p>If x is a list of strings, the string() function returns a comma-separated list of strings.</p> <p>Returns: string</p>	
strings(x)	<p>Tries to convert each element of the list x into a string.</p> <p>Returns: list</p>	<pre>#m1 = (1, 2, 3); #m2 = strings(#m1); //or: #m1.strings()</pre>
val(x)	<p>Tries to convert the scalar x to value type.</p> <p>Returns: value</p>	<pre>%v = val('12' + '34'); Result: %v = 1234.</pre>
vals(x)	<p>Tries to convert each element of the list x into a value.</p> <p>Returns: list</p>	<pre>#m1 = ('1', '2', '3'); #m2 = vals(#m1); //or: #m1.vals()</pre>

Date combining functions

Function name	Description	Examples
date(d, f, opt)	<p>Converts the date d into a new date with frequency f (string), and option opt (string). The option can be 'start' or 'end'.</p> <p>When converting from a higher frequency to a lower frequency, the result does not depend upon the option opt.</p> <p>Returns: date</p>	<pre>%d = 2020q2; PRT %d.date('m', 'start'); //2020m4 PRT %d.date('m', 'end'); //2020m6 PRT %d.date('a', 'start'); //2020 PRT %d.date('a', 'end'); //2020</pre>
date(y, f, sub)	<p>Constructs a new quarterly or monthly date from y (integer), frequency (string), and subperiod (integer).</p>	<pre>%d = date(2020, 'q', 2); //2020q2</pre>

	<p>Note: you may also use <code>date(x)</code>, where <code>x</code> can be a value or a string, and Gekko will try to convert the argument into a date.</p> <p>Returns: date</p>	
<code>fromExcelDate(v)</code>	<p>Converts an Excel date (the val <code>v</code>, counting the number of days since January 1, 1900) to year, month and day (hours etc. are not converted). The year, month and day are returned as a map with the values <code>%y</code>, <code>%m</code>, <code>%d</code>.</p> <p>WARNING: this function will soon return a Gekko date instead. See also <code>toExcelDate()</code>. [New in 3.0.7]</p> <p>Returns: map.</p>	See examples regarding the <code>toExcelDate()</code> function.
<code>getFreq(d)</code>	<p>Extracts the frequency of a date</p> <p>Returns: string</p>	<pre>%d = 2020q2; PRT %d.getfreq(); // 'q'</pre>
<code>getMonth(d)</code>	<p>Extracts the month number from a date. More specific than <code>getSubPer()</code>, and will fail if the date is not monthly.</p> <p>Returns: val</p>	<pre>%d = 2020m2; PRT %d.getmonth(); // 2</pre>
<code>getQuarter(d)</code>	<p>Extracts the quarter number from a date. More specific than <code>getSubPer()</code>, and will fail if the date is not quarterly.</p> <p>Returns: val</p>	<pre>%d = 2020q2; PRT %d.getquarter(); // 2</pre>
<code>getSubPer(d)</code>	<p>Extracts the sub-period from a date (1 if annual or undated, the quarter if quarterly, and the month if</p>	<pre>%d = 2020q2; PRT %d.getsubper(); // 2</pre>

	monthly). Returns: val	
getYear(d)	Extracts the year from a date. Returns: val	<pre>%d = 2020q2; PRT %d.getyear(); //2020</pre>
toExcelDate(y, m, d)	Converts year, month and day (integers) into an Excel date (counting the number of days since January 1, 1900). See also fromExcelDate(). Excel dates can be subtracted to obtain days. [New in 3.0.7] Returns: val.	<pre>%v1 = toExcelDate(2019, 11, 12); %v2 = toExcelDate(2019, 12, 3); PRT %v1, %v2; //43781 and 43802 PRT %v2 - %v1; //21 days in between #x = fromExcelDate(%v1 + 100); //100 days from %v1: Feb. 20, 2020. PRT #x.%y, #x.%m, #x.%d;</pre>

String combining functions

Function name	Description	Examples
[x]-index	Index: returns the character at position x. Returns: string	<pre>%s = 'abcd'; PRT %s[2]; // 'b'</pre>
[x1..x2]-index	Index: returns the range of characters from position x1 to x2 (both inclusive). You may omit x1 or x2. Returns: string	<pre>%s = 'abcd'; PRT %s[2..3]; // 'bc'</pre>
concat(s1, s2)	Appends the two strings: same as s1 + s2. Returns: string	<pre>%s = concat('He', 'llo'); Result: 'Hello'.</pre>
endswith(s1, s2)	Returns 1 if the string s1 starts with the string s2, else 0. The comparison is case-insensitive. Returns: val	<pre>%v = endswith('abcde', 'cde'); Returns: 1</pre>

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	<pre>%v = index('onetwothreetwo', 'two'); Returns: 4. %v = index('oneTWO', 'two'); Returns: 4.</pre>
isAlpha(s)	Returns 1 if all the characters are letters (alphabet). [New in 3.0.5].	<pre>%v = isAlpha('aBc'); Returns: 1</pre>
isLower(s)	Returns 1 if the string contains no uppercase characters. [New in 3.0.5].	<pre>%v = isLower('abc12'); Returns: 1</pre>
isNumeric(s)	Returns 1 if all the characters are of numeric value. [New in 3.0.5].	<pre>%v = isNumeric('123'); Returns: 1</pre>
isUpper(s)	Returns 1 if the string contains no lowercase characters. [New in 3.0.5].	<pre>%v = isUpper('ABC12'); Returns: 1</pre>
length(s)	The length of the string (number of characters). You may use len() instead of length(). Returns: val	<pre>%v = %s.length();</pre>
lower(s)	The string in lower-case letters. Returns: string	<pre>%s = lower('aBcD'); Result: 'abcd'.</pre>
prefix(s1, s2)	If s1 is a string, it has the string s2 prefixed (prepended). Returns: string	<pre>%s1 = %s2.prefix('a');</pre>
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	<pre>%s = replace(%s1, %s2); //or: replace(%s, %s1, %s2)</pre>

	<p>at most max times.</p> <p>Returns: string</p>	
<p>split(s1, s2) split(s1, s2, removeempty, strip)</p>	<p>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, 1 per default (set to 0 or 1), see examples. [New in 3.0.6]</p>	<pre>%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')</pre>
<p>startswith(s1, s2)</p>	<p>Returns 1 if the string s1 starts with the string s2, else 0. The comparison is case-insensitive.</p> <p>Returns: val</p>	<pre>%s = 'abcde'; %v = %s.startswith('abc'); Returns: 1</pre>
<p>strip(s)</p>	<p>Removes blank characters from the start and end of the string.</p> <p>Returns: string</p>	<pre>%s1 = %s2.strip(); //or: strip(%s1)</pre>
<p>stripstart(s)</p>	<p>Removes blank characters from the start of the string.</p> <p>Returns: string</p>	<pre>%s1 = %s2.stripstart(); //or: stripstart(%s1, %s2)</pre>
<p>stripend(s)</p>	<p>Removes blank characters from the end of the string.</p> <p>Returns: string</p>	<pre>%s1 = %s2.stripend(); //or: stripend(%s1, %s2)</pre>
<p>substring(s, start, length)</p>	<p>The piece of the string between character number start and length (these must be integer values).</p> <p>You can alternatively use a</p>	<pre>%s = %s1.substring(3, 2); //or: substring(%s1, 3, 2) %s = %s1[3 .. 5]; //a slice from pos 3 to 5 (both inclusive)</pre>

	'slice', using []-notation, see example. Returns: string	
suffix(s1, s2)	If s1 is a string, it has the string s2 suffixed (appended) Returns: string	<code>%s1 = %s2.suffix('a');</code>
upper(s)	The string with upper-case letters. Returns: string	<code>%s = upper('aBcD');</code> Result: '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	<code>#m[3];</code> //the third element
[x1..x2]-index	Index: picks out a range of elements. You may omit x1 or x2. Returns: list	<code>#m[3..5];</code> //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].	<code>#m = ((1, 2), (3, 4));</code> <code>PRT #m[2, 1], #m[2]</code> <code>[1];</code> //same

<p>[x1..y1, x2..y2]-index [x1..y1, x2]-index [x1, x2..y2]-index</p>	<p>For a nested list of lists, <code>#m[2..3, 2..4]</code> will select the given "rows" and "columns", corresponding to selecting a submatrix from a matrix. Beware that in general, <code>#m[2..3, 2..4]</code> is completely different from <code>#m[2..3][2..4]</code>. See more here. Returns: list [New in 3.0.6].</p>	<pre>// 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!</pre>
<p>append(x1, x2) append(x1, i, x2)</p>	<p>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 <code>extend()</code>.</p> <p>If used with i argument, x2 is inserted at index i, instead of at the end. See also <code>extend()</code>.</p> <p>To prepend, use <code>append(x1, 1, x2)</code>.</p> <p>Returns: list</p>	<pre>#y = #x1.append(#x2); //or: append(#x1, #x2) #y = #x1.append(2, #x2); //insert at position 2</pre>
<p>contains(x1, x2)</p>	<p>Checks if the list of strings x1 contains the string x2. Returns 1 if true, 0 otherwise. You may alternatively use <code>x2 in x1</code>, see the last example. See also the <code>count()</code> and <code>index()</code> functions. The comparisons are case-insensitive. Returns: val</p>	<pre>%v = #x1.contains(%s); if(#x1.contains(%s) == 1); tell 'yes'; end; if(%s in #x1); tell 'yes'; end;</pre>

count(x1, x2)	<p>Counts the number of times the string x2 is present in the list of strings x1. See also the contains() and index() functions.</p> <p>Note: to obtain the number of elements in a list, use the length() function. The comparisons are case-insensitive.</p> <p>Returns: val</p>	<pre>%v = #x1.count(%s); //or: count(#x1, %s)</pre>
data(x)	<p>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.</p> <p>Returns: list</p>	<pre>#m = data('1.0 2.0 1.5');</pre>
dates(x)	<p>Tries to convert each element of the list x to a date.</p> <p>Returns: list</p>	<pre>#y = dates(#x);</pre>
except(x1, x2)	<p>The except() function subtracts x2 from x1. You may alternatively use the operator -. Only works for lists of strings. See also intersect() and union().</p> <p>Was called difference() in Gekko 2.0. See also extend().</p> <p>Returns: list</p>	<pre>#y = #x1.except(#x2); //or: except(#x1, #x2) #y = #x1 - #x2; //same #y -= #x1; //subtract from itself</pre>
extend(x1, x2) extend(x1, i, x2)	<p>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</p>	<pre>#y = #x1.extend(#x2); //or: extend(#x1, #x2) #y = #x1 + #x2; //same as above #y = #x1.extend(2, #x2); //insert at position 2</pre>

	<p>x1.</p> <p>For two lists x1 and x2, you may alternatively use the <code>+</code> operator. See also <code>except()</code> and <code>append()</code>.</p> <p>To pre-extend, use <code>extend(x1, 1, x2)</code>.</p> <p>Returns: list</p>	<pre>#y += #x1; //add to itself</pre>
<code>flatten(x)</code>	<p>For at list x, the function returns a flattened version of the list. For instance, the list <code>(1, (2, 3))</code> is transformed into a non-recursive list of non-list elements: <code>(1, 2, 3)</code>.</p> <p>Returns: list</p>	<pre>#m1 = (1, (2, 3)); #m2 = #m1.flatten(); //or: flatten(#m1).</pre>
<code>index(x1, x2)</code>	<p>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 <code>count()</code> and <code>contains()</code> functions. The comparisons are case-insensitive.</p> <p>Returns: val</p>	<pre>%i = #x1.index(%s); //or: index(#x1, %s)</pre>
<code>intersect(x1, x2)</code>	<p>The <code>intersect()</code> function finds the common elements of the two list of strings x1 and x2. You may alternatively use the operator <code>&&</code>. Only works for lists of strings. See also <code>except()</code> and <code>union()</code>.</p> <p>Returns: list</p>	<pre>#y = #x1.intersect(#x2); //or: intersect(#x1, #x2) #y = #x1 && #x2;</pre>
<code>length(x)</code>	<p>Returns the number of elements in the list x. You may use <code>len()</code> instead of <code>length()</code>.</p> <p>Returns: val</p>	<pre>%v = #x.length(); //or: length(#x). %v = #x.len(); //the same</pre>

<code>list(x1, x2, ...)</code>	Returns a list of the variables <code>x1</code> , <code>x2</code> , etc. The function is handy for lists with only 0 or 1 elements. See examples. Returns: list	<pre>#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</pre>
<code>lower(x)</code>	Returns string elements in the list as lower-case. Returns: list	<pre>#y = #x1.lower(); //or: lower(#x1)</pre>
<code>pop(x1, i)</code> <code>pop(x1)</code>	Removes the element at position <code>i</code> in the list <code>x1</code> . Removes the last element if called with <code>pop(x)</code> . Returns: list	<pre>#y = #x1.pop(2); //or: pop(#x1, 2) #y = #x1.pop(); //last element #y = #x1.pop(1); //first element</pre>
<code>preextend(x1, x2)</code>	Same as <code>extend(x1, 1, x2)</code> , putting the elements of <code>x2</code> in the first position of <code>x1</code> .	<pre>#y = #x1.preextend(#x2); //insert at position 1</pre>
<code>prefix(x1, x2)</code>	If <code>x1</code> is a list of strings, each element has the string <code>x2</code> prefixed (prepended) Returns: list	<pre>#y = #x1.prefix(%s); //or: prefix(#x1, %s);</pre>
<code>prepend(x1, x2)</code>	Same as <code>append(x1, 1, x2)</code> , putting <code>x2</code> in the first position of <code>x1</code> .	<pre>#y = #x1.prepend(#x2); //insert at position 1</pre>
<code>sort(x)</code>	Returns a sorted list of strings, provided that <code>x</code> is a list of strings. Sorting is case-insensitive. Returns: list	<pre>#y = #x.sort(); //or: sort(#x)</pre>
<code>remove(x1, x2)</code>	Removes any string <code>x2</code> from the list of strings <code>x1</code> . See also the <code>except()</code> function. Returns: list	<pre>#y = #x1.remove(%s); //or: remove(#x1, %s);</pre>
<code>replace(x1, x2, x3)</code> <code>replaceinside(x1, x2, x3)</code> <code>replaceinside(x1, x2, x3,</code>	<code>replace()</code> : In the list of strings <code>x1</code> , if this string element is the same as <code>x2</code> , <code>x3</code> is inserted instead. <code>replaceinside()</code> : the string	<pre>#y = #x1.replace(%x2, %x3); //or: replace(#x1, %x2, %x3) #y = #x1.replaceinside(%x2, %x3); //or: replace(#x1, %x2, %x3, 'inside')</pre>

max)	<p>element has any occurrences of x2 inside the string replaced with x3. The replacements may be limited via the max argument.</p> <p>Returns: list</p>	
reverse(x)	To be done	
split(x, s)	To be done	
strings(x)	<p>Tries to convert each element of the list x to a string</p> <p>Returns: list</p>	<code>#y = strings(#x);</code>
suffix(x1, x2)	<p>If x1 is a list of strings, each element has the string x2 suffixed (appended)</p> <p>Returns: list</p>	<code>#y = #x1.suffix(%s); //or: suffix(#x1, %s);</code>
t(x)	<p>For a nested list of lists, the t() function returns the transpose, similar to transposing a matrix. [New in 3.0.6].</p> <p>Returns: list (of lists)</p>	<code>#m = ((1, 2), (3, 4)); p #m, t(#m);</code>
union(x1, x2)	<p>The union() function adds the two lists (only adds unique elements in x2 that are not in x1), or you may use the operator <code> </code>. Alternatively, you may use <code>x + y</code>, but that may introduce dublets. Only works for lists of strings. See also except() and intersect().</p> <p>Returns: list</p>	<code>#y = #x1.union(#x2); //or: union(#x1, #x2) #y = #x1 #x2;</code>
unique(x1)	<p>Retains only those elements of list x1 that are unique (list of strings only).</p> <p>Returns: list</p>	<code>#y = #x1.unique(); //or: unique(#x1)</code>

upper(x)	Returns string elements in the list as upper-case. Returns: list	<pre>#y = #x1.upper(); //or: upper(#x1)</pre>
vals(x)	Tries to convert each element of the list x to a value Returns: list	<pre>#y = vals(#x);</pre>

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	<pre>%name = addBank('x!q', 'b2'); Result: 'b2:x!q'</pre>
addFreq(x, freq)	If x does not have a freq, a freq is added. The input x may be string or list. Returns: string or list	<pre>%name = addFreq('x', 'q'); Result: 'x!q'</pre>
getBank(x)	Returns the bank part of x. The input x may be series, string or list. Returns: string or list	<pre>%bank = getBank('b2:x!q'); Result: 'b2'</pre>
getFreq(x)	Returns the freq part of x. The input x may be series, string or list. Returns: string or list	<pre>%bank = getFreq('b2:x!q'); Result: 'q'</pre>
getFullName(bank, name, freq)	Returns the full name corresponding to the input, where bank, name and freq are strings. Returns: string	<pre>%name = getFullName('b2', 'x', 'q'); Result: 'b2:x!q'</pre>
getFullName(bank,	Returns the full name corresponding to the input, where bank, name and freq	<pre>%name = getFullName('b2', 'x', 'q', ('a', 'b')); Result: 'b2:x!q[a,b]'</pre>

name, freq, index)	are strings, and index is a list (of strings) Returns: string	
getIndex(x)	Returns the index part of x. The input x may be string or list. Returns: list	<pre>#index = getIndex('b2:x!q[a, b]'); Result: ('a', 'b') If the input is a list, the output will be a list of lists.</pre>
getName(x)	Returns the name part of x. The input x may be series, string or list. Returns: string or list	<pre>%name = getName('b2:x!q'); Result: 'x'</pre>
getNameAnd Freq(x)	Returns the name part of x. The input x may be series, string or list. Returns: string or list	<pre>%name = getNameAndFreq('b2:x!q'); Result: 'x!q'</pre>
removeBank (x)	Removes any bank in x. The input x may be string or list. Returns: string or list	<pre>%name = removeBank('b2:x!q'); Result: 'x!q'</pre>
removeBank (x, bank)	Removes any banks in x with the indicated bankname. The input x may be string or list. Returns: string or list	<pre>%name = removeBank('b2:x!q', 'b2'); Result: 'x!q' %name = removeBank('b3:x!q', 'b2'); Result: 'b2:x!q'</pre>
removeFreq(x)	Removes any freq in x. The input x may be string or list. Returns: string or list	<pre>%name = removeFreq('b2:x!q'); Result: 'b2:x'</pre>
removeFreq(x, freq)	Removes any freq in x with the indicated freqname. The input x may be string or list. Returns: string or list	<pre>%name = removeFreq('b2:x!q', 'q'); Result: 'b2:x' %name = removeFreq('b3:x!q', 'm'); Result: 'b2:x!q'</pre>
removeIndex (x)	Removes any index in x. The input x may be string or list. Returns: string or list	<pre>%name = removeIndex('b2:x!q[a, b]'); Result: 'b2:x!q'</pre>

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	<pre>%name = replaceBank('b2:x!q', 'b2', 'b3'); Result: 'b3:x!q' %name = replaceBank('b2:x!q', 'b3', 'b4'); Result: 'b2:x!q'</pre>
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	<pre>%name = replaceFreq('b2:x!q', 'q', 'm'); Result: 'b3:x!m' %name = replaceFreq('b2:x!q', 'm', 'a'); Result: 'b3:x!q'</pre>
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	<pre>%name = setBank('b3:x!q', 'b2'); Result: 'b2:x!q'</pre>
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	<pre>%name = setFreq('b2:x!q', 'm'); Result: 'b2:x!m'</pre>
setName(x, name)	The indicated name is set. The input x may be string or list. Returns: string or list	<pre>%name = setName('b2:x!q', 'y'); Result: 'b2:y!m'</pre>
setNamePrefix(x, p)	The name of x has prefix p added. The input x may be string or list. Returns: string or list	<pre>%name = setNamePrefix('b2:x!q', 'a'); Result: 'b2:ax!m'</pre>
setNameSuffix(x, s)	The name of x has suffix s added. The input x may be string or list. Returns: string or list	<pre>%name = setNameSuffix('b2:x!q', 'b'); Result: 'b2:xb!m'</pre>

Timeseries functions

Function name	Description	Examples
---------------	-------------	----------

getdomains(x)	Returns a list of strings containing the domains for each dimension of the array-series x. Returns an empty list if there are no domains given. Returns: list	<pre>#d = getdomains(x); //or: #d = x.getdomains();</pre>
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 calculation). Time period may be indicated with t1-t2. Returns: series	<pre>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</pre>
laspchain(plist, qlist, t) laspchain(<t1 t2>, plist, qlist, t)	Laspeyres chain index. Calculates a map containing two series <i>p</i> and <i>q</i> (price and quantity) from a list of prices and a corresponding list of quantities, setting <i>p</i> = 1 in period <i>t</i> (<i>t</i> is a date). See also the bottom of the LIST help page. A period can be indicated in the <t1 t2> field. Returns: map	<pre>#p = p1, p2; //prices #q = q1, q2; //quantities #m = laspchain(#p, #x, 2000); // #m is a map PRT #m.p, #m.q; Or in one command (only the quantity printed): PRT laspchain(('p1', 'p2'), ('q1', 'q2'), 2000).q; #m = laspchain(<1980 2020>, #p, #x, 2000); //with period</pre>
laspfixed(plist, qlist, t) laspfixed(<t1 t2>, plist, qlist, t)	Laspeyres fixed-price index. As laspchain(), but with fixed prices. Returns: map	<pre>//See example regarding laspchain()</pre>
percentile(x, %v)	Computes the %v percentile for the series x, for the global time period. Any missing values within that sample are ignored.	<pre>%z = percentile(y, 0.25); %z = percentile(y, 0.50);</pre>

	<p>Setting %v = 0.5 results in the median. Returns: val</p>	
replace(x, v1, v2)	<p>For the series x, the function replaces the value v1 with the value v2, over the given sample.</p> <p>See also iif() and the \$-conditional.</p> <p>Returns: series</p>	<pre>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: <2000 2004> z = x.replace(m(), 0); //result is (0, 1, 0, 3, 0)</pre>
rotate(x, d)	<p>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 subseries for each age group 0 to 100, that is, pop['0'], pop['1'], ... , pop['100']. These 101 subseries are all defined over a time period, say 2020-2050. Then profile = rotate(pop, 1) will be a new array-series containing subseries for each time period 2020 to 2050, that is, profile['2020'], profile['2021'], ... , profile['2050']. These 31 subseries 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.</p> <p>Returns: series</p>	<pre>//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];</pre>

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. Returns: series	<pre> 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] = ... </pre>
setdomains(x, d)	Sets a list of strings (d) containing the domains for each dimension. Returns: nothing	<pre> #d = ('#b', '#x'); setdomains(x, #d); //or: x.setdomains(#d); </pre>
timeless(freq, v)	<p>Constructs a timeless series of the given frequency, to value v. You can skip some of these options, see examples.</p> <p>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.</p> <p>Returns: series</p>	<pre> 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] = ... </pre>

Time, databank and timeseries info

Function name	Description	Examples
bankfilename(s) bankfilename(s, p)	Return the filename of the s databank, where s is a string. Can include path. Returns: string	<pre> %s1 = bankfilename('work'); %s1 = bankfilename('work', 'fullpath'); //with path </pre>

bankname(s)	Returns the name of the bank. Input can be the string 'first' or 'ref', or a val designating the number in the databank list. Returns: string	<pre>%b1 = bankname('first'); %b0 = bankname('ref'); %b2 = bankname(2);</pre>
banktime(s)	Return the time stamp of the s databank, where s is a string. Returns: string	<pre>%s1 = banktime('work');</pre>
currentDateTime()	Returns current date and time. Returns: string	<pre>%s = currentDateTime(); //Returns: '15-09-2014 12:34:58' (for instance). TELL currentDateTime();</pre>
currentDate()	Returns current date. Returns: string	<pre>%s = currentDate(); //Returns: '15-09-2014' (for instance).</pre>
currentDay()	Returns the current day. [New in 3.0.5]. Returns: val	<pre>%v = currentDay();</pre>
currentFolder()	Returns the current working folder (cf. <code>OPTION folder working = ...</code>). [New in 3.0.6]. Returns string.	<pre>%s = currentFolder();</pre>
currentFreq()	Returns the current frequency, for instance 'a', 'q', 'm' or 'u'. Returns: string	<pre>%s = currentFreq(); //Returns: 'a' (depending upon frequency setting, cf. option freq).</pre>
currentHour()	Returns the current hour. [New in 3.0.5]. Returns: val	<pre>%v = currentHour();</pre>
currentMinute()	Returns the current minute. [New in 3.0.5]. Returns: val	<pre>%v = currentMinute();</pre>
currentMonth()	Returns the current month. [New in 3.0.5]. Returns: val	<pre>%v = currentMonth();</pre>

currentPerStart()	Returns the start of the global time period. Returns: date	<pre>%s = currentPerStart(); //Returns: 2012q1 (for instance).</pre>
currentPerEnd()	Returns the end of the global time period. Returns: date	<pre>%s = currentPerStart(); //Returns: 2015q4 (for instance).</pre>
currentSecond()	Returns the current second. [New in 3.0.5]. Returns: val	<pre>%v = currentSecond();</pre>
currentTime()	Returns current time. Returns: string	<pre>%s = currentTime(); //Returns: '12:34:58' (for instance).</pre>
currentYear()	Returns the current year. [New in 3.0.5]. Returns: val	<pre>%v = currentYear();</pre>
exist(x)	Returns 1 if the variable x exists, else 0. The variable name must be a string. 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 <code>option databank search</code> setting (that is, in <code>sim-mode</code> it will only look in the first-position databank, if a databank name is not provided). Returns: val	<pre>%v = exist('gdp'); %v = exist('db2:gdp');</pre>
filteredperiods(d1, d2)	Returns the number of filtered periods between d1 and d2 (these are dates). Returns: val	<pre>%v = filteredperiods(%d1, %d2);</pre>
fromSeries(x, type)	Accesses meta-information from the timeseries. Type can be <ul style="list-style-type: none"> 'name' 	<pre>%s = fromSeries(ref:gdp, 'label'); //Returns 'Gross domestic product' (for instance). %d = fromSeries(gdp,</pre>

	<ul style="list-style-type: none"> • 'bank' • 'freq' • 'label' • 'source' • 'units' • 'stamp' • 'perStart' or 'perEnd' (may include missings) • 'dataStart' or 'dataEnd' (period with actual data) <p>NOTE: The x argument must be a series (without quotes).</p> <p>Returns: string or date</p>	<pre>'perStart');</pre> <pre>//Returns 1980q1 (for instance). Same logic regarding 'perEnd' argument.</pre> <pre>%s = fromSeries(gdp, 'freq');</pre> <pre>//Returns 'q' (for instance).</pre>
gekkoVersion()	<p>Returns the Gekko version number (xx.yy.zz).</p> <p>Returns: string</p>	<pre>%s = gekkoVersion();</pre> <pre>//Returns: '3.0.1' (for instance).</pre>
getEndoExo()	<p>Returns a list with names of those variables that start with 'endo_' or 'exo_'. This is used with GAMS models, when fixing equations.</p> <p>Returns: list of strings.</p>	<pre>#m = getEndoExo();</pre>
isOpen(x)	<p>Returns 1 if the databank with the name x (a string) is open, and 0 otherwise.</p> <p>Returns: val</p>	<pre>%v = isopen('mybank');</pre>
time() time(<t1 t2>)	<p>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.</p> <p>Returns: series</p>	<pre>option freq q;</pre> <pre>time 2010q1 2011q4;</pre> <pre>p time();</pre> <pre>p time(<2010q3 2011q3); //truncated</pre> <pre>//the first one prints</pre> <pre>2010.125, 2010.375, 2010.625,</pre> <pre>2010.875, ...</pre>

Matrix functions:

Function name	Description	Examples
avgc(x)	Average over cols. Returns: matrix	<code>#m2 = avgc (#m1);</code>
avgr(x)	Average over rows Returns: matrix	<code>#m2 = avgr (#m1);</code>
chol(x) chol(x, type)	Cholesky decomposition of matrix x. Accepts type (string), either 'upper' or 'lower'. Returns: matrix	<code>#m2 = chol (#m1, 'upper');</code>
cols(x)	Returns the number of columns of x Returns: val	<code>%v = cols (#m);</code>
det(x)	Determinant of a matrix. Returns: val	<code>%v = det (#m);</code>
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	<code>#m2 = diag (#m1);</code>
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	<code>#x = divide (#x1, #x2);</code>

<code>i(n)</code>	Returns a $n \times n$ identity matrix. Returns: matrix	<code>#m = i(10);</code>
<code>inv(x)</code>	Inverse of matrix x Returns: matrix	<code>#m2 = inv(#m1);</code>
<code>maxc(x)</code>	Max over cols Returns: matrix	<code>#m2 = maxc(#m1);</code>
<code>maxr(x)</code>	Max over rows Returns: matrix	<code>#m2 = maxr(#m1);</code>
<code>minc(x)</code>	Min over cols Returns: matrix	<code>#m2 = minc(#m1);</code>
<code>minr(x)</code>	Min over rows Returns: matrix	<code>#m2 = minr(#m1);</code>
<code>m(r, c)</code> or <code>miss(r, c)</code>	Returns a $n \times k$ matrix filled with missing values. Cf. also <code>m()</code> function for values. Returns: matrix	<code>#m = m(5, 10);</code>
<code>multiply(x1, x2)</code>	Element by element multiplication of the two matrices. If <code>x2</code> is a row vector, each <code>x1</code> column will be multiplied with the corresponding value from the row vector. And if <code>x2</code> is a column vector, each <code>x1</code> row will be multiplied with the corresponding value from the column vector. Returns: matrix	<code>#x = multiply(#x1, #x2);</code>
<code>ones(n, k)</code>	Returns a $n \times k$ matrix filled with 1's Returns: matrix	<code>#m = ones(5, 10);</code>
<code>pack(v1, v2, ...)</code> <code>pack(<t1 t2>, v1, v2, ...)</code>	Using period <code>t1-t2</code> , the timeseries <code>v1, v2, ...</code> are packed into a $n \times k$ matrix, where n is the number of observations and k is the	<code>#m = pack(<2020 2030>, x, y, z);</code> Returns: a 11×3 matrix <code>#m</code> 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	<code>%v = rows (#m);</code>
sumc(x)	Sum over cols Returns: matrix	<code>#m2 = sumc (#m1);</code>
sumr(x)	Sum over rows Returns: matrix	<code>#m2 = sumr (#m1);</code>
t(x)	Returns the transpose of a matrix. Returns: matrix	<code>#m2 = t (#m1);</code>
trace(x)	Returns the trace of a matrix. Returns: val	<code>%v = trace (#m);</code>
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	<code>//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</code>
zeros(n, k)	Returns a n x k matrix filled with 0's. Zeroes() can be used as alias. Returns: matrix	<code>#m = zeros (5, 10);</code>

Miscellaneous functions:

Function name	Description	Examples
m() or miss()	Returns a missing value. Useful in some series or matrix expressions. Cf. also the m(r, c) function for matrices. Returns: value	<pre><2020 2020> y = m(); #m = [1, 2; m(), 4];</pre>
map()	Returns an empty map.	<pre>#m = map();</pre>
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). [New in 3.0.6]. Returns: null variable	<pre>#m = ((1, 2, 3), (4, null(), 6)); PRT #m; PRT #m[2, 2].type(); // 'null'</pre>
readFile(x)	Reads the file x (string) as a string. See also writeFile(). Returns: string	<pre>%s = readFile('rawdata.txt');</pre>
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	<pre>#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'</pre>
writeFile(x, s)	Writes the string s to the file x (string). A newline can be indicated with '\n'. See also readFile().	<pre>writeFile('rawdata.txt', '170 121 387');</pre>

Complete alphabetical list of in-built functions:

- `abs()`
- `addbank()`
- `addfreq()`
- `append()`
- `avg()`
- `avgc()`
- `avgr()`
- `avgt()`
- `bankfilename()`
- `bankname()`
- `chol()`
- `cols()`
- `concat()`
- `contains()`
- `count()`
- `currentdate()`
- `currentdatetime()`
- `currentfreq()`
- `currentperend()`
- `currentperstart()`
- `currenttime()`
- `data()`
- `date()`
- `dates()`
- `det()`
- `diag()`
- `dif()`
- `diff()`
- `diffy()`
- `dify()`
- `divide()`
- `dlog()`
- `dlogy()`
- `endswith()`
- `except()`
- `exist()`
- `exp()`
- `extend()`
- `filteredperiods()`
- `flatten()`
- `format()`
- `fromseries()`
- `gekkoversion()`
- `getbank()`
- `getdomains()`
- `getendoexo()`
- `getfreq()`

- getfullname()
- getindex()
- getmonth()
- getname()
- getnameandfreq()
- getquarter()
- getsubper()
- getyear()
- hpfilter()
- i()
- iif()
- index()
- intersect()
- inv()
- ismiss()
- isopen()
- lag()
- laspchain()
- laspfixed()
- len()
- length()
- list()
- log()
- lower()
- m()
- map()
- maxc()
- maxr()
- minc()
- minr()
- miss()
- movavg()
- movsum()
- multiply()
- null()
- ones()
- pack()
- pch()
- pchy()
- percentile()
- pop()
- pow()
- power()
- preextend()
- prefix()
- prepend()
- readfile()
- remove()
- removebank()
- removefreq()
- removeindex()

- `replace()`
- `replacebank()`
- `replacefreq()`
- `replaceinside()`
- `rnorm()`
- `rotate()`
- `round()`
- `rows()`
- `rseed()`
- `runif()`
- `seq()`
- `series()`
- `setbank()`
- `setdomains()`
- `setfreq()`
- `setname()`
- `setnameprefix()`
- `setnamesuffix()`
- `sort()`
- `sqrt()`
- `startswith()`
- `string()`
- `strings()`
- `strip()`
- `stripend()`
- `stripstart()`
- `substring()`
- `suffix()`
- `sum()`
- `sumc()`
- `sumr()`
- `sumt()`
- `t()`
- `time()`
- `timeless()`
- `trace()`
- `type()`
- `union()`
- `unique()`
- `unpack()`
- `upper()`
- `val()`
- `vals()`
- `writefile()`
- `zeroes()`
- `zeros()`

Part V

5 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](#)

5.1 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 t_1 to t_2 , this simulation is repeated again and again (where the values of any lead variable are simply taken from the databank), until the lead variable(s) converge (that is, do not move from iteration to iteration). The particular way each of the simulations from t_1 to t_2 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 lead 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 y_0 ($= y[2000]$) and y_5 ($= y[2005]$) 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 $100 \times 100 = 10.000$ 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:

$$y_{\text{new}} = F(y_{\text{old}})$$

So y_{old} is a $n \times 1$ 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 $y_{\text{new}} = y_{\text{old}}$, 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 $n \times n$ 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 lead 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 ($y_{\text{new}} - y_{\text{old}}$) into a new guess regarding y . In order to do this, the 4×4 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:

$$\begin{aligned} y_1 &= 0.1*y_0 + 0.2*y_1 + 0.3*y_2 + 100 \\ y_2 &= 0.1*y_1 + 0.2*y_2 + 0.3*y_3 + 100 \\ y_3 &= 0.1*y_2 + 0.2*y_3 + 0.3*y_4 + 100 \\ y_4 &= 0.1*y_3 + 0.2*y_4 + 0.3*y_4 + 100 \end{aligned}$$

Note the last equation, where y_4 is used instead of y_5 . 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:

$$\begin{aligned} y_1 &= 0.1*y_0 + 0.2*y_1 + 0.3*y_2 + 100 \\ y_2 &= 0.1*y_1 + 0.2*y_2 + 0.3*y_3 + 100 \\ y_3 &= 0.1*y_2 + 0.2*y_3 + 0.3*y_4 + 100 \\ y_4 &= 0.1*y_3 + 0.2*y_4 + 0.3*y_4*y_4/y_3 + 100 \end{aligned}$$

The last equation corresponds to $y_5/y_4 = y_4/y_3$, 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 y_4 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 $(k*n) \times (k*n)$ instead of $n \times n$, 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 lead 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 commands:

```
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 its 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.

Part VI

6 Guided tours

In this part of the documentation, some guided tours are provided. A guided tour showcases some of Gekko's capabilities, in a manageable step-by-step manner. The tours are intended for potential or actual Gekko users who wish to get a quick look around, without having to read lots of help pages etc.

- [Modeling guided tour](#)

Some planned tours:

- Data handling guided tour (planned)
- Array-series primer (planned)
- GAMS modeling interface (planned)

Regarding the tours, the reader has two options: (a) download Gekko and the provided files, and run the examples step by step along with the guide, or (b) just read the guide without installing/using Gekko, to get a quick idea of the software.

6.1 Guided tour: modeling

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

1. [Installation and download](#)
2. [Graphical interface etc.](#)
3. [Historical simulation](#)
4. [Multiplier analysis \(shocks\)](#)
5. [Add-factors etc.](#)
6. [Goal-search etc.](#)
7. [Forward-looking](#)

6.1.1 1. Installation and download

Gekko is open-source, and free of charge. Gekko is intended for the simulation of large-scale econometric (or energy) models, and more generally for handling of time-series data. This guide will focus on the simulation capabilities, that is, sim-mode (in contrast to data-mode).

Installation is normally quite simple, if using a Windows system (Windows XP service pack 3, Windows Vista, Windows 7, 8 or 10). First, since this guide is intended for Gekko 3.0, please go to the [download page](#) and download and install Gekko 3.0 (right-click and download the installer file, `InstallerForGekko.msi`). After the file is downloaded, you can double-click it and start the installation. Please see the troubleshooting page if problems arise during installation. (If comfortable with computers, you may alternatively download a zip-file for manual installation).

After the installation is finished, you should be able to find and start Gekko under the Windows start menu.

When Gekko starts for the first time, it will typically set the working folder to your desktop folder. This is not particularly convenient, so please create a new working folder somewhere else (in Windows Explorer), and point Gekko to that location (File – > Set working folder...).

The example data is a model and databank file. 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..."

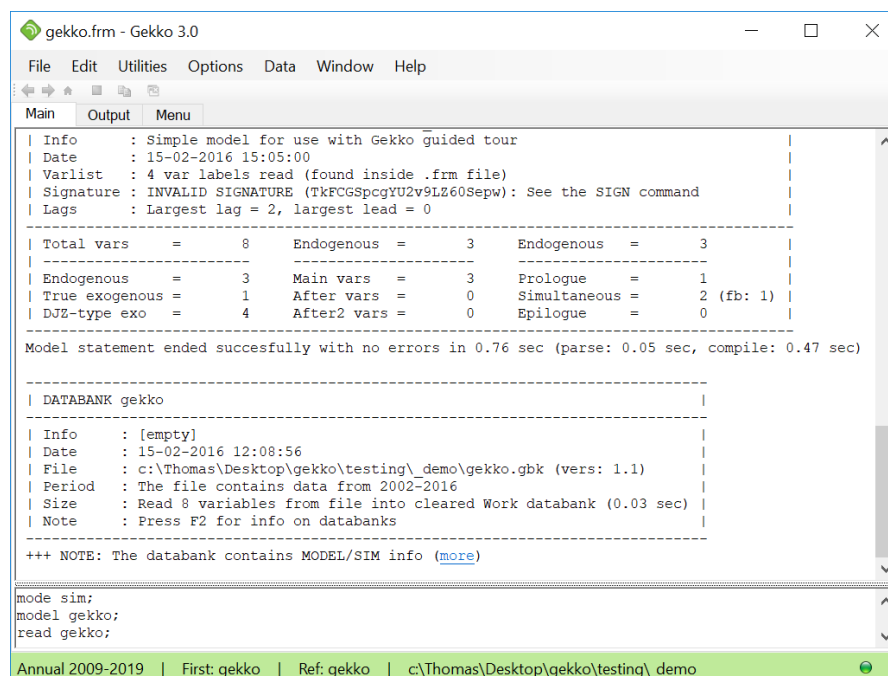
- Example model and data files, [demo.zip](#)

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.

6.1.2 2. Graphical interface etc.

This section describes basics of the graphical user interface (GUI), in addition to some concepts and conventions used in Gekko.

The GUI looks as follows:



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 command is also available by means of typing "help [commandname]" at the command 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.

In addition to the GUI, some other basic concepts used in Gekko should be noted. If you prefer, you may skip the rest of this section and jump directly to the hands-on examples in section 3.

Some basic Gekko concepts

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.

Timeseries: A timeseries resides in a databank. It may have frequency annual, quarterly, monthly 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 a databank is read (the READ command), 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 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 commands to do such comparisons, for instance [MULPRT](#), [DECOMP](#) etc.).

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](#) command. This command clears the Ref databank, and copies the Work databank into it (in memory). You may alternatively read a file directly into the Ref databank, if you use

READ<ref>. CLONE is typically used just after simulating ([SIM](#)) a baseline/reference scenario.

There is a cleanup-command: [RESTART](#). This command 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-commands in this file will be run, so `gekko.ini` can be used to contain options and other commands (for instance a [MODEL](#) command) that the user wishes to "survive" a RESTART statement.). You may use [CLS](#) ("clear screen") to clear the output window.

In general, when doing simulations (in so-called sim-mode) you will have to [CREATE](#) a timeseries before you update its values/observations with the [SERIES](#) command (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 command 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).

Note that commands 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](#) command.

Note also that the so-called operators are used in many places. 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) command 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 available, for instance `abs` or `pch` (try for instance `PRT<abs>` or `PRT<pch>`).

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](#) commands. Regarding equation syntax, you may consult the latter part of the [MODEL](#) help file, if you need more information on this.

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

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, you can use `{%x}` instead of `%x`, 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'`.

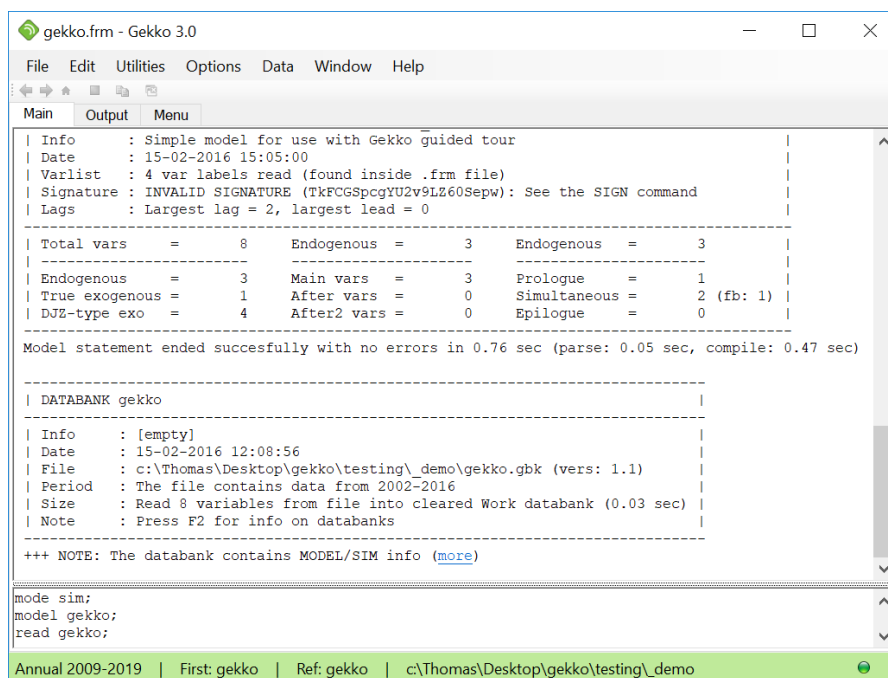
Commands must end with `;`, but in the Gekko main window, if you hit `[Enter]` while being at the end of a line, Gekko will add the `;` automatically.

6.1.3 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.



```

gekko.frm - Gekko 3.0
File Edit Utilities Options Data Window Help
Main Output Menu
| 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 (TkFCGSpGcYU2v9LZ60Sepw): 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 command prompt (at the bottom):

```
RESTART;
MODE sim;
TIME 2004 2016;
```

If you copy-paste these commands 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 commands 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;
```

Main Output Menu						
	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 commands load gekko.frm and read the gekko.gbk databank file. Note the use of the `<` and `>` brackets denoting local options. In all relevant commands, you can state a time period inside such brackets, and the given time period will only be used in that command. 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](#) command 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](#) commands 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;
```

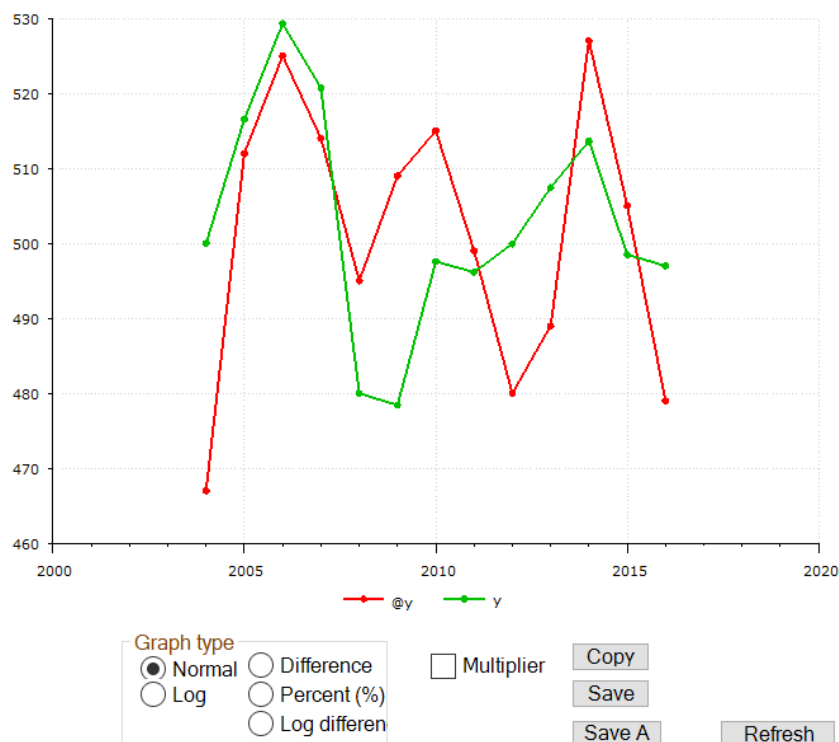
The symbol [@](#) indicates that the value is taken from the Ref databank. Alternatively, try:

```
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

In general, for commands 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 command:

```
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 example 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 commands 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](#) command. This command 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` command. Type:

```
DECOMP y;
```

		Time-change		Multiplier		Options	
		Raw	Decomp	Raw	Decomp		
Levels	<i>n</i>	<input checked="" type="radio"/>	<input type="radio"/>	Levels	<i>n</i>	<input type="radio"/>	<input type="radio"/>
Abs. time-change	<i>d</i>	<input type="radio"/>	<input type="radio"/>	Abs. multiplier	<i>m</i>	<input type="radio"/>	<input type="radio"/>
Growth rate	<i>p</i>	<input type="radio"/>	<input type="radio"/>	Relative multiplier	<i>q</i>	<input type="radio"/>	<input type="radio"/>
Chng. growth rate	<i>dp</i>	<input type="radio"/>	<input type="radio"/>	Mul. growth rate	<i>mp</i>	<input type="radio"/>	<input type="radio"/>

	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;

```

6.1.4 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 commands load gekko.frm and reads the gekko.tsdx 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 command:

```
g %= 0;
SIM;
PRT {#vars};
```

The $\% =$ operator in the first (SERIES) command 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 \forall means 'verbose'):

```
MULPRT <v> y;
```

As it is seen, there are missing values in the Ref databank. The [CLONE](#) command 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:

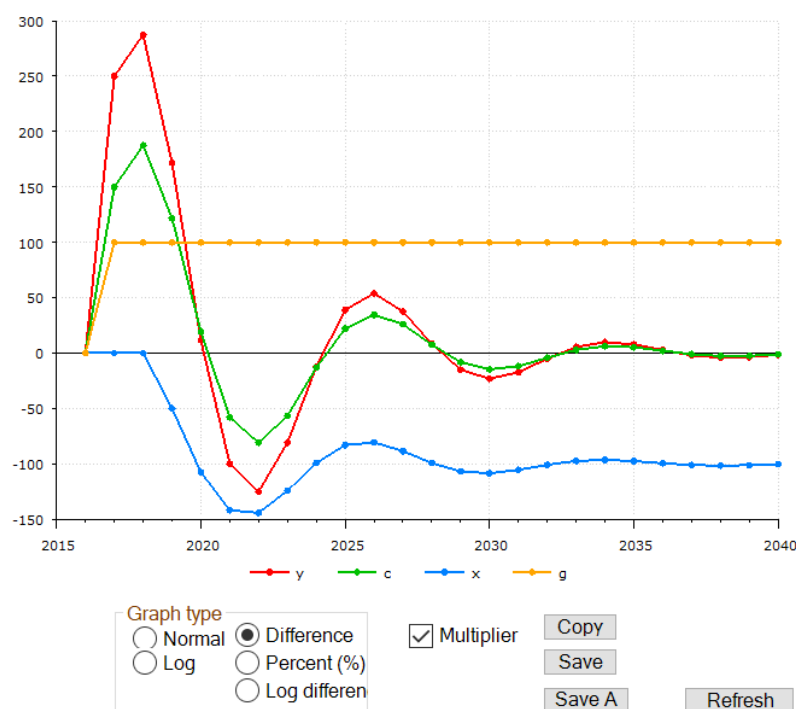
```
SIM;
```

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

```
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.



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

The effect is augmented a little bit more the following year, because of the lagged endogenous term in the \bar{c} -equation. From 2019 and on, however, the net exports begin to decline (the effect propagates via the term $\bar{y}(-2)$ in the \bar{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 \bar{x} pulls \bar{y} back downwards, and as it is seen, \bar{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, \bar{y} and \bar{c} are unchanged, whereas \bar{g} has been augmented by 100 and \bar{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 \bar{g} is quite large in percentages, whereas \bar{y} and \bar{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 commands 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		Multiplier	
		Raw	Decomp	Raw	Decomp
Levels	n	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Abs. time-change	d	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>
Growth rate	p	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Chng. growth rate	dp	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

	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;

```

6.1.5 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 commands, 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
```

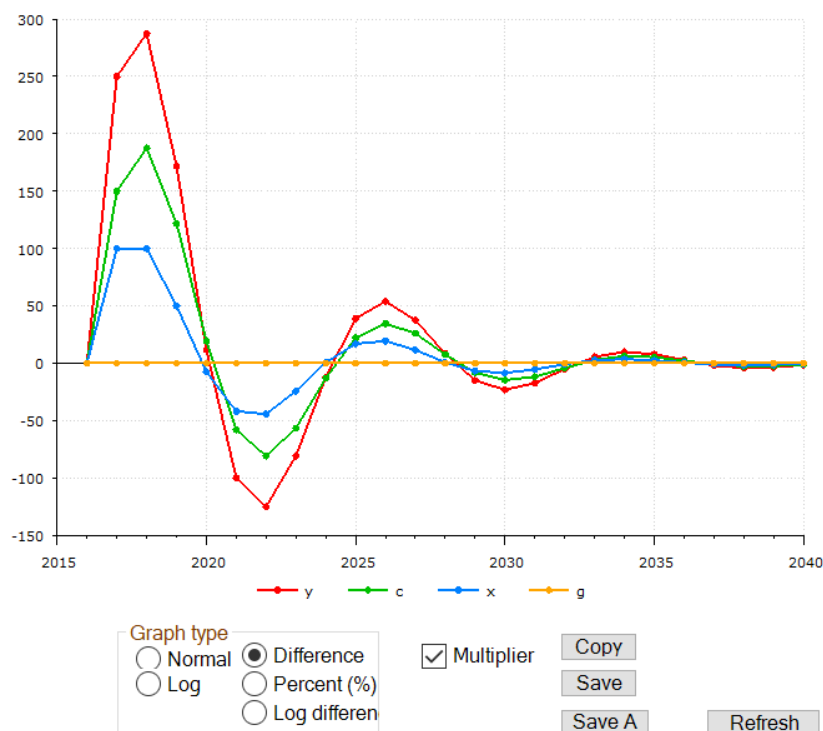
Note the add-factor `JDx`, and note also that the left-side `dif()`-function has also been resolved. This latter equation is the one that Gekko actually simulates, and these "detailed" equations can be shown by means of the [DISP](#) command:

```
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 $\text{dif}(x)$ 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 possibilities 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: D_c , J_c , and Z_c . The "full" c -equation is this: $c = (0.6*y + 0.1*c[-1] + J_c) * (1-D_c) + D_c*Z_c$. The variables D_c and Z_c can be used to exogenize an equation. If D_c is set to 1, the equation will reduce to $c = Z_c$, so that Z_c 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;

```

6.1.6 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 commands, 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 \bar{y} with 250 units for a 7-year period relative to the baseline scenario, by means of changing \bar{g} ? This experiment is done quite easily, via the [EXO](#) and [ENDO](#) commands:

```

EXO y;
ENDO g;

```

These commands state that \bar{y} should be redefined as an exogenous (goal) variable, whereas \bar{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 \bar{y} :

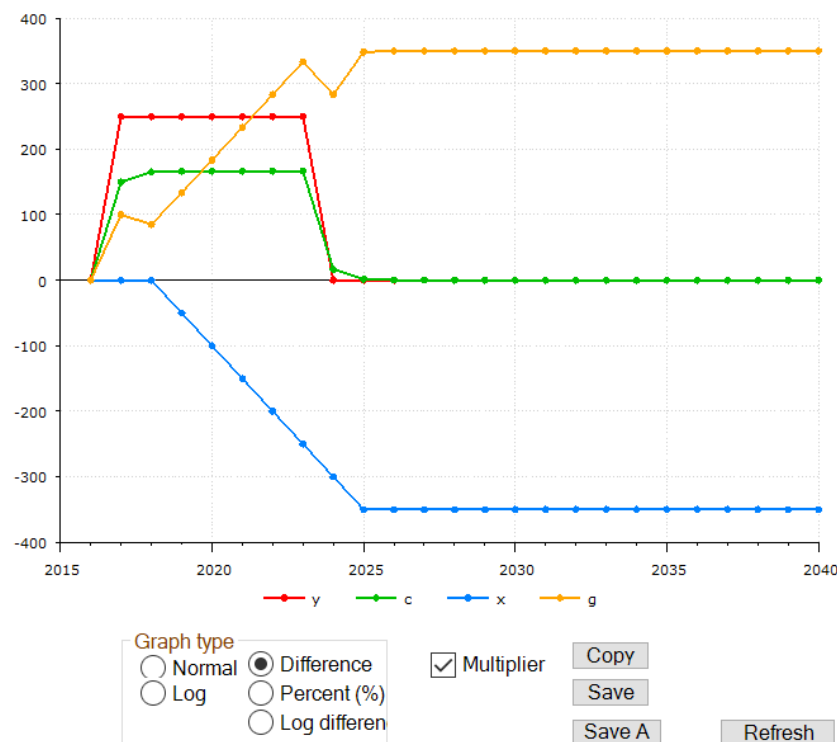
```
y <2017 2023> += 250;
```

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 command. Plot the simulated variables (multiplier):

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



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 (\bar{x}) get squeezed out, and the effect on \bar{x} is negative from 2019 on (implying a likely trade deficit, depending upon the price of imports and exports). When the \bar{y} -effect ends after 2023, \bar{g} stays at a permanently higher level (350 units higher). The mirror-image of this is that net exports has diminished by the same amount (-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](#) command 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.

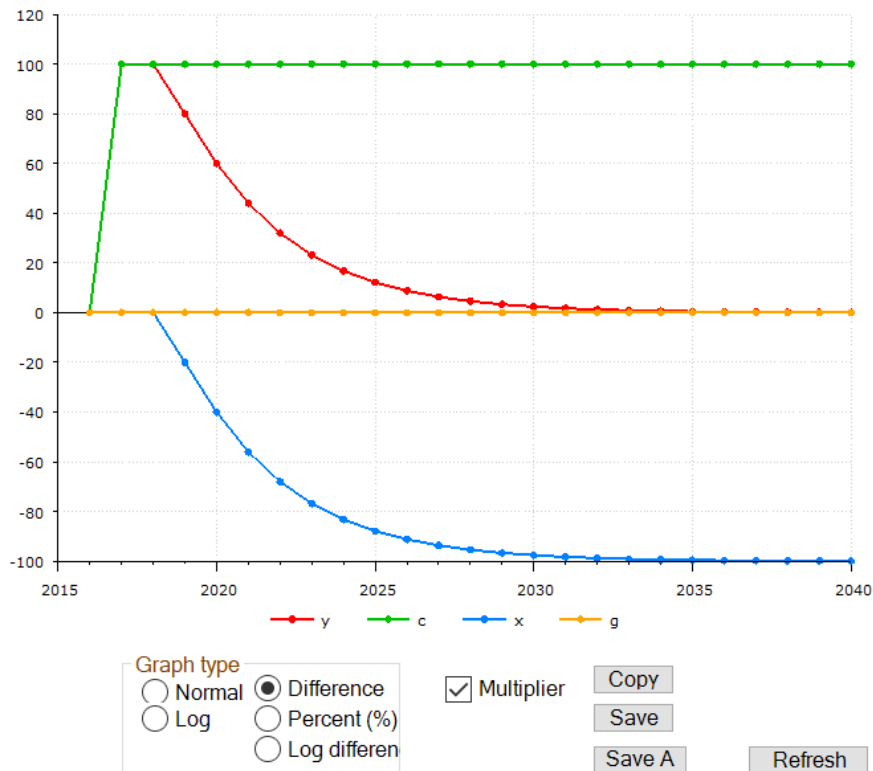
The private consumption function **c** has code **_GJ_D**, implying that it has an additive add-factor (the **J** part of the code), and exogenization dummy (the last **D** in the code). So the "full" **c**-equation really is $c = (0.6 * y + 0.1 * c[-1] + Jc) * (1 - Dc) + Dc * Zc$. The **J** part of 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 commands 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 commands:

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



As you can see, \bar{c} is now fixed at a value 100 higher than in the baseline databank. In the long run this crowds out net exports \bar{x} , so that GDP (\bar{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 \bar{c} in a convenient fashion. If, after the above simulation, \bar{Dc} is set to 0 again, and the model simulated, one would expect \bar{c} to return to its previous values. This is not the case however: the value of \bar{c} would stay at +100, even though the dummy has been set back. The explanation is that the add-factor \bar{Jc} is calculated in a way so that the \bar{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](#) commands are used. The Newton algorithm is more powerful than 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 algorithm, 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 $z = 1.1 \cdot z - 0.2 \cdot h$. This model has the solution $z = 2 \cdot h$, 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;
```

6.1.7 7. Forward-looking

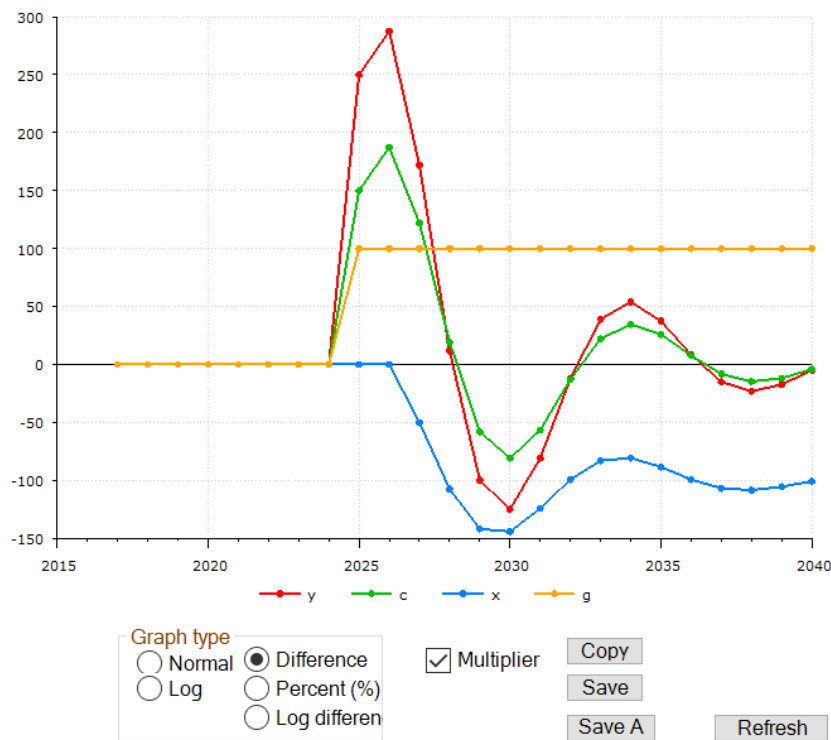
Gekko can handle forward-looking models, that is, models where one or more of the endogenous variables contain leads. There are different solvers regarding this, see under [SIM](#), and also under [OPTION](#) solve forward We will first try to perform a multiplier analysis without leaded variables, similar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



This is the same effects as seen in section 4, were in this case, \bar{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 \bar{y} is augmented by 1, \bar{c} will augment by 0.6 in the same year. We will now create a new model with a different equation regarding consumption, \bar{c} .

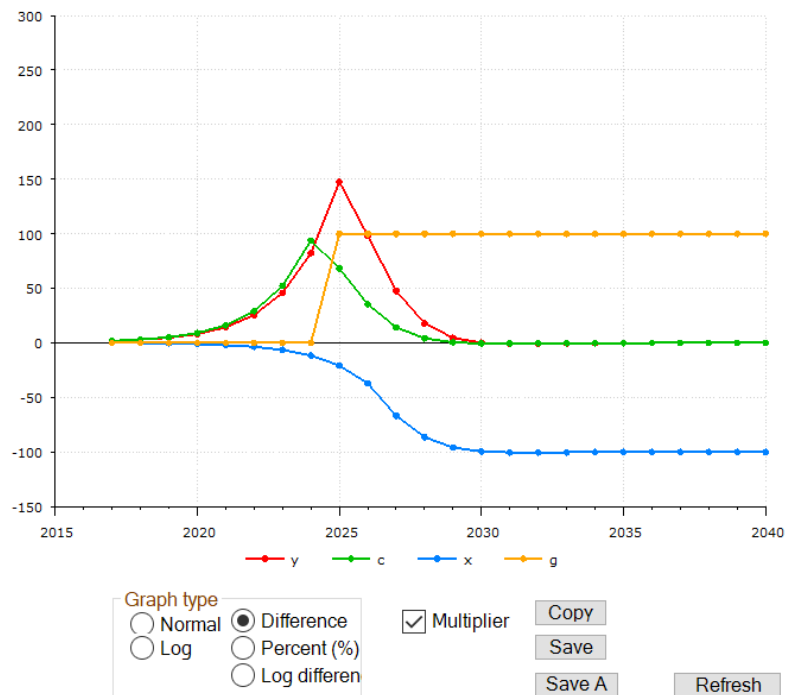
- In the file system, make a copy of gekko.frm and call it gekko2.frm
- In gekko2.frm, in the `c` equation, change `0.6*y` into `0.6*y[+1]`.

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};
```



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 \bar{y} decreases from 2025 to 2026, and hence \bar{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 $\text{dif}(x) = -0.2 * (y[-2] - 500)$ is stationary). Hence, we are setting the terminal condition manually for this problem.

Part VII

7 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)

7.1 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 data revision 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 <code>OPEN<prim></code> , local time settings like <code><2010 2020></code> , 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	<p>Related to program structure, the basic Gekko syntax (abstracting from concrete command names) is also basically similar to AREMOS syntax, albeit with some differences.</p> <ul style="list-style-type: none"> • Lists of items are separated by commas (','), for instance <code>"PRINT x1, x2 x3;"</code> • All lines end with semicolon (';') • Indexing uses square brackets ('[' and ']'), including lags. For instance <code>gdp[-1]</code>, <code>gdp[2020]</code>, <code>#m[2, 3]</code>. Ranges are <code>'..'</code> instead of <code>'-'</code> in Gekko: <code>#m[2..3, 1..5]</code>. • Lists use '#' prefix, for instance <code>#m</code>. Contrary to AREMOS, Gekko matrices also use '#' prefix to distinguish them from timeseries.

		<ul style="list-style-type: none"> • 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	<p>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.</p> <p>Gekko does not have a support bank, but it is intended to put procedures/functions into a general 'support' folder instead.</p> <p>Gekko includes READ/WRITE for dealing with databanks. This is often practical. IMPORT/EXPORT is also supported.</p> <p>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.</p>

		<p>Advanced structures and documents are not supported, but there is a DOC command to handle label and source for timeseries.</p> <p>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).</p>
Speed	ok	Gekko runs models and command files much quicker than AREMOS. For command files, AREMOS uses interpretation on a line-by-line basis, whereas Gekko compiles the commands into machine code.
Timeseries and frequencies	partial	<p>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).</p> <p>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.</p> <p>COLLAPSE, SMOOTH, SPLICE, TRUNCATE and ANALYZE are supported. Seasonal adjustment (X12A) is provided.</p> <p>Transformation series are not supported.</p>
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	<p>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 '-'. </p> <p>Pack/unpack regarding timeseries is provided.</p> <p>A matrix editor is not provided, but copy/pasting a matrix to Excel is easy (just use the copy-button in the Gekko interface).</p>

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 rather 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	different	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 global option settings in the same way as AREMOS. The syntax is OPTION rather than SET, however.
Functions/procedures	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 <i>one</i> component, and there is <i>one</i> main window. Gekko can be called silently in batch mode if needed.
Editor	no	Gekko does not contain an inbuilt editor for command files or procedures, but the EDIT command calls Notepad. External editors like Kedit or Notepad work fine with the RUN command. There is a XEDIT command 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 command 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 command 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 whether 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 commands/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 command). 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 command 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 command 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 command 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 command, 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.

7.2 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 commands (like in Gekko or AREMOS or other packages). The EViews user can run a whole file 'batch', or a block of commands can be marked and executed by right-clicking and choosing in the menu. The only way to run a single command 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 commands (run from the command window), or from batch/command 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'`", instead 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 command 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 a 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](#) command 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 to 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 probably not something that could be hidden/wrapped inside a function. Correspondingly, reference 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 originates from TSP, since EViews originates 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 commands 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 command files into executable code, but seems to use a so-called interpreter instead. This hampers the execution speed of EViews command 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.

Part VIII

8 Appendix

The following pages contain different appendices.

- [AREMOS translator details](#)
- [Gekko 1.8 translator details](#)

8.1 AREMOS translator details

The AREMOS translator (cf. the [TRANSLATE](#) command) tries to translate AREMOS command files into corresponding Gekko command 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 command 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 ... (this is Gekko 2.0 syntax). 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 apparently 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 command 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-hand side 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;.

8.2 Gekko 1.8 translator details

The Gekko 1.8 to 2.0 translator (cf. the [TRANSLATE](#) command) tries to translate Gekko 1.8 command files into corresponding Gekko 2.0 command files.

The following commands are translated:

- ADD ==> RUN.
- CLEAR ==> RESTART.
- CLEAR<all> ==> RESET.
- CPLOT ==> CLIP.
- CLOSEALL ==> RESTART.
- CLOSEBANKS ==> RESTART.
- DIFPRT ==> COMPARE.
- DISPLAY ==> TELL.
- EFTER ==> SIM<after>
- GENR ==> SERIES. Ending '\$' changed to ';'.
- GMULPRT ==> MULPRT<v>.
- HDG has quotes added.
- LIST: The old "LIST + #m a b c;" is changed into "LIST m = a, b, c;".
- MULBK ==> CLONE.
- MULBK bank ==> READ<ref>bank
- NDIFPRT ==> COMPARE<abs>.
- OPTION. Some options have been renamed.
- PPLOT ==> PLOT.
- RES ==> SIM<res>.
- SIM ==> SIM<fix>, if there has been ENDO/EXO statements.
- SKIP ... ==> /* ... */. Only skips until the next semicolon.
- STAMP ==> TELL currentDate();
- STRING ==> NAME. Most often, a Gekko 1.8 string corresponds to a Gekko 2.0 name.
- TRIMVARS ==> DELETE<nonmodel>.
- UDVALG ==> DECOMP.
- UPD ==> SERIES, also handles '\$'-operator.
- VERS ==> TELL gekkoVersion();
- WPLOT ==> SHEET.
- filename ==> RUN filename

In addition:

- '()' is translated into '//'
- Lags/leads with soft parentheses are translated into brackets, for instance fY(-1) ==> fY[-1]. The same goes for years in soft parantheses: fY(2020) ==> fY[2020].
- Lines with '&' joins are handled (Gekko 2.0 uses ';' to end such blocks).
- Commas are set: for instance "PRT x1 x2 x3;" is changed into "PRT x1, x2, x3;".
- All commands are decorated with an ending ';' if it is missing.
- Time settings like "PRT 2010 2020 fY;" are changed into "TIME 2010 2020; PRT fY;". Gekko 2.0 does not allow such (global) time settings outside <>-brackets.
- The translator changes '#' to '%' for scalars. The translator tries to keep track of scalars and lists, so that lists keep their '#'-identifier.

-
- The UPD command with line breaks is handled specially (line breaks without '&' joins were allowed for UPD)
 - Any '.cmd' is changed into '.gcm', for instance "RUN f.cmd;" ==> "RUN f.gcm;"
 - Some options are renamed.

8.3 Gekko 2.0 translator details

The Gekko 2.0 to 3.0 translator (cf. the [TRANSLATE](#) command) tries to translate Gekko 2.0/2.2/2.4 command files into corresponding Gekko 3.0 command files.

The following commands 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 a note about time periods and <all>
- (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: prefix --> .prefix(), suffix --> .suffix(), trim --> .unique(), sort --> sort(), strip --> replaceinside().
- MATRIX: '||' --> ';'.
- VAL, STRING, DATE have % added on left-hand side and command removed.
- LIST and MATRIX have # added on left-hand side and command removed.
- SERIES has command removed.
- SERIES: Operator '^' --> '^=', '%' --> '%=', '+' --> '+=', '*' --> '*=', '#' --> '#='.
- SERIES #m = ... --> {#m} = ...
- SERIES y = 1, 2, 3 --> y = (1, 2, 3);
- SERIES y = 100 rep * --> y = 100.

In addition:

- Index [0] --> length(), for instance #m[0] --> #m.length().
- Use of 'strip' in list has a syntax explanation
- A string like 'ab %s cd' is transformed into 'ab {%s} cd' (string interpolation).
- piece() --> substring()
- search() --> index()
- strip() --> use replace()
- trim() --> strip()
- difference() --> except()
- hpfilter() --> changed order
- fromseries() --> changed order
- avgt() --> changed order
- sumt() --> changed order
- An option field like SERIES <2020 2030> y = 100; is moved to the right of y, so it becomes y <2020 2030> = 100;.

8.4 Assignments

Assignments are of the form `y = x;`, that is, assigning `x` to `y`. There may be a type indicator, like `SERIES y = x;`, stating that `y` should become a series type.

Note in general that when assigning variables, the left-hand side is always assigned to a *copy* of the right-hand side.

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 includes 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.

8.5 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 kids, measured at the beginning of August each year, cf. the following data table:

Education	Age	Year	Number
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 data observations. The columns apart from the number count can be thought of as dimensions, in this case education (kindergarden or school), age, and year. It is noted that there are no data rows corresponding to 4-year old schoolchildren, and the last data row regarding 5-year old schoolchildren in 2012 seems to be missing.

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 subseries/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 subseries `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-subseries `x[s, 4]`, so that particular subseries 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 subseries `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 include an explicit row with the value 0 to indicate that, instead of just skipping the row:

Education	Age	Year	Number
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 resources. 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]` as 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" subseries 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 command, 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 explain 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 subseries as 0's does the trick:

```
OPTION series array calc missing = zero;
PRT <n> sum((#e, #a), x[#e, #a]);
```

	<code>sum((#e, #a</code>
	<code>), x[#e, #a])</code>
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];
```

	<code>x[k, 4]</code>	<code>x[s, 4]</code>	<code>x[k, 5]</code>	<code>x[s, 5]</code>
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];
```

	<code>x[k, 4]</code>	<code>x[s, 4]</code>	<code>x[k, 5]</code>	<code>x[s, 5]</code>
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 subseries `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];
```

	<code>x[k, 4]</code>	<code>x[k, 5]</code>	<code>x[s, 5]</code>
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]);
```

	<code>sum((#e, #a), x[#e, #a])</code>
2011	124.0000
2012	102.0000

The first option handles the missing subseries `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 subseries 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 columns, and the missing subseries 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 command:

```
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>.

Index

- . -

.NET 42

- A -

ACCEPT 81, 273

ANALYZE 83, 85

Appendix 499

AREMOS 393, 487, 500

- B -

Basic concepts 19, 26, 30, 36

- C -

C# 42

CHECKOFF 87

CLEAR 89

Clear screen 95

CLIP 91

Clipboard 91

CLONE 92

CLOSE 93

CLS 95

Cmd prompt 373

COLLAPSE 96

Combine timeseries 363

Command file 327

Command overview 69, 74, 77

Commands in Gekko 67

Commands in Gekko - reading guide 68

COMPARE 98

Compare Gekko software 486

Conditional 134, 135, 170

COPY 102

Correlation 83, 85

COUNT 107

CREATE 110

- D -

Data 93, 142, 174, 251, 316, 403

Databank 89, 93, 142, 167, 174, 251, 316, 320, 403

Databank copy 92

Databanks 211, 397

DATE 112

DECOMP 117

Decompose 117

DELETE 121

Demo 44

DISP 123

Display equation 123

DOC 126

Documentation 126

Dos prompt 373

DOWNLOAD 128

- E -

EDIT 132

Editor 132, 409

END 134, 135

ENDO 136, 396

Environment 186

Equations 234, 354

EViews 494

Examples 44

Excel 346

Execute 327

EXIT 140

EXO 141, 396

EXPORT 142, 174, 403

External file 93, 142, 174, 251, 316, 403

- F -

F1 41, 168

File 274

Filenames 39

Files 39

FINDMISSINGDATA 147

FOR 150, 326

Forward-looking model 449

Frequency 96, 187, 406
Function 156, 288, 326, 413, 414
Function keys 40
Functions 413, 414

- G -

Gekko 1.8 393, 502, 504, 505, 506
Gekko software 42
Gekko.ini 186, 257, 323
Global time 387
Goals and means 136, 141
GOTO 162, 165, 209, 384
Graph 277
Graphics 277
Guided tour 44

- H -

HDG 167
HELP 41, 168
Hlp 41, 168

- I -

i 48
IF 170, 326
IMPORT 174
in-built 156
Index 181, 314
INI 17, 186, 327
Interface 40
INTERPOLATE 187
Introduction to Gekko 7
ITERSHOW 189

- L -

Leads 449
Linear algebra 216
LIST 191
LOCK 211, 397
Loop 134, 135, 150

- M -

map 212, 331, 401
maps 156
MATRIX 216
MEM 226
MENU 228, 375
Meta-information 126
Missing value 147, 360
MODE 74, 77, 231
Mode: data 231
Mode: sim 231
Mode: universal 231
MODEL 234, 354, 356
MULPRT 241, 295

- N -

NAME 366
New features 9, 14
Notepad 132

- O -

OLS 14, 247
Online databank 128
On-line help 41
OPEN 93, 251
Open source 42
OPTION 257
Options 257
Ordinary least squares 247

- P -

Path 39
PAUSE 81, 273
Period 112
PIPE 274
PLOT 277
Print 123, 241, 295
PROCEDURE 156, 288
Program 156, 288, 327
PRT 241, 295

- R -

R 307, 309, 310
 R_EXPORT 307
 R_FILE 309
 R_RUN 310
 READ 316
 REBASE 314
 Rebase timeseries 314
 Redirect 274
 RENAME 320
 RESET 322
 RESTART 323
 RETURN 326
 RUN 327

- S -

Scalar 366
 Scalar variable 112, 226, 366, 398
 Script 327
 Search 181
 Seasonal correction 406
 SERIES 331, 401
 Set period 387
 Setup of Gekko 17
 SHEET 346
 SIGN 354
 SIM 87, 136, 141, 189, 234, 356, 448, 449
 Simulation 87, 136, 141, 189, 356, 396, 448, 449
 SMOOTH 360
 Solve 356, 448, 449
 SPLICE 363
 Spreadsheet 346
 Start-up 186
 STOP 365
 Stop Gekko 140, 326, 365
 STRING 366, 385
 Syntax basics 47
 Syntax rules 48
 SYS 373
 System shell 373

- T -

TABLE 228, 375
 TARGET 384
 TELL 375, 385
 TIME 387, 390
 TIMEFILTER 390
 Timeseries 331, 395, 401
 Timeseries functionality 83, 85, 96, 147, 187, 314, 331, 360, 363, 395, 401
 Translate 393, 500, 502, 504, 505, 506
 TRUNCATE 395

- U -

UNFIX 396
 UNLOCK 211, 397
 User defined function 156, 288
 User defined procedure 156, 288
 User input 81

- V -

VAL 398
 Value 398

- W -

Wait 273
 Wildcard 181, 191, 212
 WRITE 142, 316, 403

- X -

X12A 406
 XEDIT 409
 XmlNotepad 409