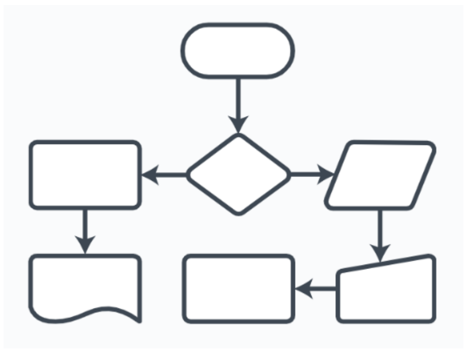
Gekko model blueprint

Thomas Thomsen, 27/11 2020



In this blueprint, some issues regarding model and equation definitions, syntax, capabilities, etc. are discussed.

In the first part of the blueprint, models and equations are considered in a more abstract way. The second part looks at datastructures generally (and syntax for those), whereas the third part considers more specific questions related to the Gekko implementation. The last part tries to sum up.

These Gekko model enhancement suggestions primarily have to do with how models and equations are defined and combined, and not how models are solved with a solver.

The suggestions will probably be refined and at some point implemented, probably step by step over time. Because of this projected stepwise implementation, it is important to have a reasonably clear idea of how it should look like, before such work begins. In the end, it is the hope that the modelling capabilities and the syntax dealing with these will feel unified and consistent with other parts of Gekko.

1. Models and equations in general

1.1 Abstract models

A model can be thought of as a set of equations, for instance as follows:

$\mathrm{F}_1(x_1, x_2, \dots, x_n) = 0$

$\mathrm{F}_2(x_1, x_2, \dots, x_n) = 0$

...

$\mathrm{F}_n(x_1, x_2, \dots, x_n) = 0$

This could illustrate the solution corresponding to a single time period. Time, t, could be added into each variable, transforming the first equation into $\mathrm{F}_1\left(x_{1,t}, x_{2,t}, \dots, x_{n,t}\right) = 0$, but lags and leads could obviously be present, too, transforming it into something like $\mathrm{F}_1\left(x_{1,t-1}, x_{1,t}, x_{1,t+1}, x_{2,t-1} \ \dots, x_{n,t+1}\right) = 0$, assuming here that the maximal lag and lead is 1 period. In timeseries-oriented software, such an equation is often represented with the notation $\mathrm{F}_1(x_1[-1], x_1, x_1[+1], x_2[-1] \ \dots, x_n[+1]) = 0$.

Economic model equations are often similar in different dimensions, not least the time dimension. For instance, the following equation:

$$0.1\ \cdot x_1[-1] +\ 0.8\ \cdot x_1 +\ 0.1\ \cdot x_1[+1]\ - 100 = 0$$

is typically unrolled over time, for instance corresponding to the following for $t = 2020$:

$$0.1\ \cdot x_1[2019] +\ 0.8\ \cdot x_1[2020] +\ 0.1\ \cdot x_1[2021]\ - 100 = 0$$

This unrolling can be repeated for $t = 2021, t = 2022$, and so on – which is called (time) *stacking*.

Besides time, t, equations are also often similar over other dimensions, for instance age groups, sectors, countries, etc. For instance, we might consider the following equations:

$$0.1\ \cdot x_1[-1] +\ 0.8\ \cdot x_1 +\ 0.1\ \cdot x_1[+1]\ - 100 = 0$$

$$0.1\ \cdot x_2[-1] +\ 0.8\ \cdot x_2 +\ 0.1\ \cdot x_2[+1]\ - 100 = 0$$

$$0.1\ \cdot x_3[-1] +\ 0.8\ \cdot x_3 +\ 0.1\ \cdot x_3[+1]\ - 100 = 0$$

These equations are obviously similar, and one interpretation could be that x_i represents x in different sectors, or that x_i represents x in different age groups. A way of representing this could be the following:

$$0.1\ \cdot x[\#i][-1] +\ 0.8\ \cdot x[\#i] + 0.1\ \cdot x[\#i][+1]\ - 100 = 0$$

where the list $\#i = 1, 2, 3$. The use of $'\#'$ is is Gekko notation, and in Gekko x would be a 1-dimensional Gekko-array-series. Gams has a similar representation (this is no coincidence):

$$0.1\ \cdot x[i, t-1]\ +\ 0.8\ \cdot x[i, t]\ \ + 0.1\ \cdot x[i, t+1]\ - 100 = 0$$

where $i = 1, 2, 3$, and x is a 2-dimensional GAMS-variable. In GAMS, i is called a *set*, or more specifically: an 'uncontrolled' set (we will get back to the distinction between 'controlled' and 'uncontrolled' sets). Another way of avoiding repetitions is by using lists/sets in sums. Consider this equation:

$$xs = \text{sum}(\#i,\ x[\#i])$$

With list $\#i = 1, 2, 3$, this would be equivalent to $xs = x[1] + x[2] + x[3]$. This could represent, say, age groups, so that xs is the sum of 1-, 2-, and 3-year olds (note here that lags and leads in Gekko must always start with either - or +, so there will be no confusion). If, instead, the list had comprised these elements, $\#i = a, b, c$, the sum would be equivalent to $xs = x[a] + x[b] + x[c]$, where a, b, c could be, for instance, sector names.[1]

In an expression like $\text{sum}(\#i,\ x[\#i])$, the list $\#i$ in $x[\#i]$ is said to be 'controlled', because it is in a sense absorbed into the sum() function, not producing three equations to unroll. In

[1] Written in the most formal/strict way, the sums should be written as $x['1'] + x['2'] + x['3']$, or $x['a'] + x['b'] + x['c']$, highlighting the fact that array-series dimensions in Gekko are represented by strings, not numbers. The more compact version without quotes works in Gekko (but not in GAMS).

contrast, in an equation like $0.1 \cdot x[\#i][-1] + 0.8 \cdot x[\#i] + 0.1 \cdot x[\#i][+1] - 100 = 0$, the list $\#i$ is ‘uncontrolled’, implying that it will generate (in this case) three implicit unrolled equations. If there were two uncontrolled lists, $\#i$ and $\#j$, the implicitly generated equations would be the product of the number of elements of the two list.

Sometimes there are exceptions to general patterns, and in Gekko (and GAMS) such exceptions are handled via $-conditionals. For instance, an equation like this:

$$xs = \mathrm{sum}(\#i,\ x[\#i]\ \$\ (\#i \text{ in } \#i0))$$

If $\#i = a, b, c$, and $\#i0 = a, c$, the equation will amount to $xs = x[a] + x[c]$. These $-conditionals can contain any logical condition, for instance $\mathrm{sum}(\#i,\ x[\#i]\ \$\ (x[\#i] > 0))$ would sum all positive $x[\#i]$ and ignore the rest. There are some technicalities regarding $-conditionals, for instance whether they switch off implicit equations, or whether they implicitly assign contributions to 0 in sums. In any case, $-conditionals are a very practical way to handle exceptions to general rules, instead of using loops and if-statements.

In addition to $-conditionals, there is the question of whether equations should be stated with a “dependable” variable on the left-hand side. For instance:

$$0.1 \cdot x[\#i][-1] + 0.8 \cdot x[\#i] + 0.1 \cdot x[\#i][+1] - 100 = 0$$

could instead be written as:

$$x[\#i] = \frac{100 - 0.1 \cdot x[\#i][-1] - 0.1 \cdot x[\#i][+1]}{0.8}$$

In a GAMS-type model, a ‘dependent’ variable is not necessarily defined in its ‘own’ unique equation like above. In such models, the user needs to denote a list of endogenous (dependent) variables, since these are not necessarily given as the set of left-hand side variables.

The current modelling facilities in Gekko demands that each endogenous variable is defined in one and only one equation. This is pretty easy to circumvent, though. Consider this GAMS-like equation:

$$0.1 \cdot x[\#i][-1] + 0.8 \cdot x[\#i] + 0.1 \cdot x[\#i][+1] - 100 = 0$$

This can be written equivalently as follows:

$$x[\#i] = x[\#i] + (0.1 \cdot x[\#i][-1] + 0.8 \cdot x[\#i] + 0.1 \cdot x[\#i][+1] - 100)$$

This has the same solution, obviously, and the Newton solver algorithm in Gekko does not care about these superfluous $x[\#i]$. But still it is more aesthetically pleasing to also support GAMS-like equations, where the left-hand and right-hand sides are free to chose.

1.2 Name-composition

Instead of using array-series, Gekko and other software packages also supports name-composition. For instance, consider the array-series $x[a]$, that is, a one-dimensional array-series with an element $'a'$. Instead of an array-series, you may use a name like xa. In that case, you may still use looping and sums, for instance (this work for series-statements in Gekko 3.0):

$$x\{\#i\} = y\{\#i\} + z\{\#i\}$$

$$xs = \text{sum}(\#i,\ x\{\#i\})$$

If $\#i = a, b$, these equations are equivalent to $xa = ya + za,\ xb = yb + zb$ and $xs = xa + xb$. Compare this to the array-series variants, $x[a] = y[a] + z[a],\ x[b] = y[b] + z[b]$ and $xs = x[a] + x[b]$.

Why use array-series at all? There are several answers to that question:

- Easier to decipher
- No name-collisions. Consider the two array-series $x[ab]$ versus $xa[b]$, or alternatively $y[a, bc]$ versus $y[ab, c]$. Naive translation to non-array-series names would produce xab and $yabc$, respectively, forcing the use of for instance underscore as separator (x_ab versus xa_b, or y_a_bc versus y_ab_c).
- Array series dimensions can be designated with allowed values for the elements of each dimension (domain restrictions, also known from GAMS), eliminating a source of errors.
- Array-series provide more structure and book-keeping, more of an "object" feel. For instance `PRT x;` could print all sub-series contained inside the array-series `x`, or `PRT x[b..d, *];` could print out parts of `x`. Also, arithmetics can be supported, for instance `PRT x1*x2;`, where `x1` and `x2` are array-series, and where the corresponding elements are multiplied before being printed.

1.3 Submodels

As shown in the first section, a model can be thought of as a list of n equations, with n endogenous variables (these may be stacked over time internally by the solving algorithm, if the model contains leaded variables).

$y = c + i + g + x$ (E1)

$c = 0.6 \cdot y$ (E2)

$i = 0.2 \cdot y + i0$ (E3)

$g = g0 + jg$ (E4)

$iy = i/y$ (E5)

A model like the above model is “normalized” in the sense that endogenous variables are put on the left-hand sides, and these equations are thought of as “designating” the variable. Still, the user can swap endogenous and exogenous variables, for instance endogenizing jg and exogenizing y. This would amount to a goals-means problem, finding the jg adjustment of g that makes y attain some specific value.

Models may be split into prologue, simultaneous block, and epilogue. For instance, in this model, the prologue consists of E4 (government spending) the simultaneous block of E1, E2, E3 (production, consumption, and investments), and the epilogue of E5 (investments share). In Gekko, this sorting of equations is done automatically, reducing simulation time, because the prologue can be run independently as a first step (as a recursive pre-model), and the epilogue can be run independently as a last step (as a recursive post-model).

For large models, identifying all prologue and epilogue equations (and also ordering these correctly, so they only need to be run one time each) is error-prone, and it is therefore best if the software can do this automatically.

Regarding the partitioning of a model into model blocks, the distinction of whether an equation belongs to the prologue, the simultaneous block, or the epilogue can sometimes be useful for the model user (to reason about causes and effects), but model-wise it is typically more interesting to divide a model into blocks that belong logically together, for instance consumption, investments, the housing sector, foreign trade, prices and wages, etc.[2]

For instance, we could divide the above simple model into households (c), firms (i), government sector (g), and bookkeeping identities (y and iy). If the household sector is big enough, it could be sub-divided into for instance consumption, housing, wealth, etc.[3]

[2] Using goals-means (swapping of endogenous and exogenous variables) alters the partitioning of model equations into prologue, simultaneous, and epilogue, so this partitioning in pretty conditional in any case.
[3] There is the question of whether an equation ought to be able to belong to different categories, for instance the S (savings) could belong to both households and bookkeeping identities. This is probably too complicated to operate with in practice.

2. Nested data structures in Gekko, and more generally

2.1 Gekko databanks and maps

When designing advanced modelling capabilities, including being able to deal with both (sub)models and equations as objects, there is a striking similarity between these capabilities and the existing databank and map capabilities of Gekko 3.0. We will therefore describe these first.

In Gekko 3.0, databanks and maps store objects ("variables") of any kind. Or actually, you cannot store a databank inside another databank, or a databank inside a map, but a map can be thought of as a mini-databank, just with some syntactical differences.[4] An example:

```
#m1 = (%v = 123, x = 234);
```

This defines a map that stores the two variables `%v` and `x` (a scalar value and a timeseries). To print these, you can use dot, for instance:

```
prt #m1.%v;
prt #m1.x;
```

A databank operates in the following way:

```
open <edit> m1;
m1:%v = 123;
m1:x = 234;
prt m1:%v;
prt m1:x;
```

In some cases, the `m:` designator could be omitted when using databanks, but we are not addressing the details of databank logic in this blueprint. Note instead that when printing, the only difference is that in the map case, we use for instance `#m.%v`, whereas in the databank case we use `m:%v`.

Now, a scalar value like `%v` can live inside a map. Maps can live inside databanks, and (child) maps can live inside (parent) maps, so the map capabilities seem similar to how we would like to be able to operate on models, submodels, equations, etc.

Maps in Gekko 3.0 are still a bit basic syntactically, at present not allowing a lot of functionality compared to databanks. Regarding databanks, we can for instance do the following:

[4] In the Gekko source code, both databanks and maps share a common interface, so the internals of these components is identical.

- Clear (remove all variables)
- Copy variables between databanks
- Rename/move variables
- Delete variables
- Index (search for variables using wildcards and ranges)
- Import variables from data files
- Export variables to data files

At the moment, these functionalities are lacking regarding maps, and it should obviously also be possible to for instance copy variables between databanks and maps. Basically this is mostly a question of adjusting for instance this syntax:

```
copy m1:x*k to m2:*_m;
```

into this syntax.

```
copy #m1.x*k to #m2.*_m;
```

In the first statement, the databank m is searched for variables starting with `x` and ending with `k`. These are copied to the databank `m2`, all with suffix `_m` appended. In the second statement, syntactically we simply prefix a `#`, and replace the colon `:` with a dot `.`.

Here, though, we may ask which databank `#m1` and `#m2` belong to? If not stated, it is the first-position databank (often the Work databank), but otherwise we could indicate this with for instance `copy b1:#m1.x*k to b2:#m2.*_m;`. There is a similar difference regarding sub-maps too, namely the fact that maps may contain maps. In that case, the syntax `#m1.#m2.#m3.%v` can be used to fetch a variable inside nested maps. Some tailor-made capabilities could perhaps be practical, for instance harvesting all variables inside all sub-maps of a given map (this would be similar to the existing flatten() function that deals with nested lists).[5]

But apart from that, it seems the databank syntax capabilities can just be more or less reused regarding maps. In addition to the above-mentioned capabilities, there are also quite a lot of helper methods regarding databanks available in Gekko 3.0. For instance addBank(), getBank(), removeBank(), setBank(), etc. These functions break up a variable name like `m:x` into the bank part (`m`), and the name part (`x`) and perform operations on these (lists of such descriptions can also be operated on in bulk). Therefore, given a list of such variable names, it is easy to for instance add a bank name (if not present), set a bank name (even if present), remove bank names, etc.

[5] For instance, `#m.flatten()` could flatten a model, that is, remove any submodel structure, so that the model only contains raw equations. We could also envision functions like `#m.removemodel('#trade')` to remove a particular (sub)submodel.

Something similar could be done regarding maps, making it possible to handle lists of variable names like `#m1.%v1`, `#m1.%v2`, `#m2.%v1` in a similar way (breaking the description up into for instance `#m1` and `%v1`), with new functions addMap(), getMap(), removeMap(), setMap() etc.

2.2 Tree structure versus pure code, generally

Another thing to mention before moving on is the conceptual difference between designating a tree structure and using pure code. We can take the so-called WPF as an example: it is the .NET technology used for the Gekko graphical interface. A graphical interface is essentially a tree structure of graphical objects, for instance on top the main window, inside it a visual layout (for instance a grid-layout), inside the layout a text box, inside the text fox attributes like the text itself, font, color, scroll settings, etc.

This graphical tree structure in WPF can be constructed with a XML-like syntax (called XAML), or alternatively with pure code (C#).

An example (a bit simplified from a "real" XAML tree structure):

Tree structure (XAML)

```
<Window Orientation = "Vertical">
    <Grid>
        <TextBox Font = "Courier">Hello 1</TextBox>
    </Grid>
</Window>
```

Pure code (C#)

```
var window1 = new Window();
window1.Orientation = "Vertical";
var grid1 = new Grid();
var textBox1 = new TextBox();
textBox1.Text = "Hello 1";
textBox1.Font = "Courier";
grid1.Add(textBox1);
window1.Add(grid1);
```

The tree structure of the graphical object is much more visible in the first example, than in the equivalent pure code. Also, moving sub-objects around is easy with a tree structure, for instance copying the textbox part into some other parts of the tree structure. This is more tedious to do in pure code. In the pure code, we could take lines 4-6 and copy them somewhere else, but then we would need to manually add the textbox to the object it is supposed to belong to (if this is forgotten, the textbox will not show up at all).

On the flip side, some things are just easier to do in pure code. Say you wanted 5 textboxes to show up one after another, the first with text "Hello 1", the second with "Hello 2", and so on. This would be really easy in pure code (with a loop in C#), but much less easy in a tree structure (XAML).[6]

The point of all this is to say that Gekko .frm files (with sub-models, sub-sub-models, etc.) could be interpreted as a kind of tree structure, whereas the construction of models in .gcm command files could be interpreted as a kind of pure code. The former is good for keeping things structured and manageable, whereas the latter is good for more complicated tasks, for instance auto-generating some of the equations from text strings and certain rules, or using estimated parameters via OLS, etc.

We will return to the question of tree structure vs. pure code.

3. Suggestions

Sections 3.1-3.3 looks at models and equations from a pure code perspective, looking at the assembly of models and equations in Gekko command files (.gcm). Section 3.4 takes a tree structure perspektive, looking at model files (.frm).

3.1 Submodel considerations

As mentioned, there are a lot of similarities between operating with models and submodels, compared to operating on maps and submaps. Gekko restricts the use of variables without special starting symbols to timeseries, for instance the timeseries `x`, in contrast to the scalar `%x`, or the collection `#x`.

It is always debatable if we should allow for instance model or equation objects to be without these special symbols, but since Gekko is a timeseries-oriented software package, there is some sense to restricting symbol-free use to timeseries. It should be remembered that symbol-free names also covers banknames and filenames, in addition to words like `ignore` in an option like `prt <missing = ignore> ...;`. So symbol-free words are already hard at work in Gekko, and it would probably be more confusing than helpful to allow symbol-free names for models and equations.[7]

[6] XAML also allows more dynamic ”programming” (dynamic transformation of the object tree), but then the syntax typically get hairy, and the cleanness of XAML loses some of its advantage. There are also hybrid possibilities, for instance where a listbox is defined and placed in the XAML tree structure, but where its elements (for instance rows of strings) are populated via pure code, typically in another file.
[7] For this reason, and in contrast to AREMOS, Gekko matrices are stated like #m, not m.

If we think of a model object as `#m`, and an equation object as `#e`, we can operate on these syntax-wise just as if they were, for instance, maps. We could therefore envision statements like the following:

```
copy #m1.#e*k to #m2.*_m;
```

This time, we are taking the model `#m1`, asking for the equations in this model that start with `#e` and end with `k`. These are copied to the model `#m2`, each with suffix `_m` added. If the models resided in different databanks, we could for instance have the following:

```
copy b1:#m1.#e*k to b2:#m2.*_m;
```

In principle, we could use a very databank-like syntax for models, allowing them to be symbol-free, and even allowing the use of colon instead of dot to fetch submodels or equations. In that case, we could have statements like `copy b1:m1:e*k to b2:m2:*_m;`, containing less boilerplate syntactical 'noise'. But the drawback is that users may confuse databanks and models, and the use of the databank colon and databank operations is already heavily used in existing Gekko command files. Also, there would be the question about why symbol-free colon indexing is allowed for models, and not for maps? A model seems much more similar conceptually to a map, than to a databank, and therefore it makes sense that maps and models look similar syntactically.

Another advantage of sticking to the use of #-symbol and dot for models and submodels is that a lot of existing syntax and functionality can be reused/adjusted, and some of the new model capabilities would transfer directly into similar map capabilities (for instance copying, deleting, indexing, etc.).

Consistency of the Gekko syntax is important, so that the user does not get the feeling that certain parts of the software has certain syntax conventions, but it also means that skills are transferable. If a user has previously used maps to store variables, and has successfully copied, renamed and indexed variables inside such maps, doing similar operations on models and submodels would be easy.

The same goes for list operations. Gekko lists are often lists of strings (or 2D-like nested lists of lists representing for instance rows and columns of spreadsheet cells), and the inbuilt existing list functions could be reused to deal with aspects of models and equations, too.

For instance, it is easy to obtain two lists of timeseries names from two different banks, and the Gekko functions except(), intersect(), union(), and unique() represent operations from set theory used on lists. Similarly, one could obtain two lists of equation names (or left-hand side series names) from two models and use such functions to identify differences and similarities between the two models (this would be similar to the existing COMPARE command for Gekko 3.0 databanks).

A list of equation names would look like `(‘#e1’, ‘#e2’, ‘#e3’)`, that is, starting with the #-symbol. If the user prefers to operate on such names without #-symbol, there are Gekko functions that can easily remove or add this symbol to a list of strings. Example:

```
#e1 = (1, 2);
#e2 = (2, 3);
#m1 = #e1, #e2;  //two strings (‘#e1’, ‘#e2’)
#m2 = #m1.replaceinside('#', '');
p #m1;
p #m2;
p {#m1};
p #{#m2};

// --------- output ---------

#m1
'#e1', '#e2'

#m2
'e1', 'e2'

{#m1}
(1, 2), (2, 3)

#{#m2}
(1, 2), (2, 3)
```

Here, the lists `#e1` and `#e2` represent two equations, and we could imagine that the equations of a given model are given as the `#m1` list (a list of strings). The `#m2` list corresponds to the list of strings `(‘e1’, ‘e2’)`, which the user may prefer to operate on, when comparing with other models etc.[8] When using `#m2`, the user would have to prepend the #-symbol for printing, etc.

3.2 Equations

In GAMS, equations are stated in two steps: declaration and definition. For instance:

```
EQUATION E_qI[i, t] ‘Total quantity’;
E_qI[i, t] $ (tx0[t]) ..  pI[i, t-1] * qI[i, t]
                      =E= sum(s, pI_s[i, s, t-1] * qI_s[i, s, t]);
```

In the first step, the equation `E_qI` is defined `to` run over the sets `i` and `t`. Here, `i` could be the sectors `a`, `b`, and `c`, and `t` could be a time period like 2020, 2021, 2022. This would yield 3

[8] In the current Gekko 3.0, it is not possible to write such a list as `#m1 = #e1, #e2`, being short-hand for `#m1 = ('#e1', '#e2')`, because this simpler syntax is deemed confusing for the users (even though it is logical enough). This syntactical restriction will be lifted, see more on naked lists here.

x 3 GAMS equations when unrolled (GAMS always unrolls over time, too). Note that a description is present in the declaration, too.

In the definition, the sets `i` and `t` are repeated, and there is a $-restriction, too. In this case, it is the restriction that `t` belongs to the set `tx0`, which could for instance be defined as 2020 and 2021. Hence, in reality, only 3 x 2 = 6 implicit equations are unrolled.

Gekko 3.0 already allows equivalent syntax for timeseries statements in command files, in this case it would be:

```
qI[#i] = sum(#s, pI_s[#i, #s][-1] * qI_s[#i, #s]) / pI[#i][-1];
```

- Note that the time index `t` has disappeared, since time is implicitly given in Gekko (and lags are given as for instance `x[-1]`). Gekko also supports $-conditionals (restrictions on sets/lists), but in this case, the restriction on `t` would have to be "translated" into a restriction on the time period over which the equation is generated.
- Note also that `pI[#i][-1]` has been transferred to the right-hand side, and `#i` is used instead of `i`.
- A final thing to note is that in Gekko it is generally not necessary to define explicitly which sets/lists an equation or an expression runs over. The set `#s` is "controlled" via the sum() function, which leaves only one "uncontrolled" set, namely `#i`. If `#i = a, b, c`, in Gekko the statement will run over these three elements, where the first unrolled equation will be `qI[a] = sum(#s, pI_s[a, #s][-1] * qI_s[a, #s]) / pI[a][-1];`.

As mentioned, at the moment, this GAMS-like syntax is only supported regarding series statements in command files (.gcm). But we could envision an EQU command residing in Gekko command files, where these EQU's (together with for instance OLS output) could be used to form a model. EQU statements could look like the following:

```
EQU  #ey   dlog(y)  =  0.5 * dlog(x)  +  0.5 * dlog(x[-1]);
```

or

```
EQU  #eqI   qI[#i] = sum(#s, pI_s[#i, #s][-1] * qI_s[#i, #s]) / pI[#i][-1];
```

These statements could mimic the capabilities of FRML statements in .frm files. In that case, the general rule could be that the equation name is the name of the left-hand side variable, with prefix `#` (in this case: `#y`). Many economic models require that each equation has a "dependent" variable, and that the set of these dependent variables has no dublets (but rather identifies the set of endogenous variables). Since timeseries are not allowed to use `#` as first character, prefixing this symbol on the left-hand side variables creates an unique set of equation names. In other software packages, it is often seen that the name resembles the left-hand side variable anyway, using for instance `E_y` for the above equation. Compared to such conventions, a default name like `#y` does not feel as an inferior name-convention.

For multidimensional equations in command files (EQU) and equations in model files (FRML), there is the question of whether the “uncontrolled” sets/lists should be indicated. For instance like this:

```
EQU[#i]  #eqI   qI[#i] = sum(#s, pI_s[#i, #s][-1] * qI_s[#i, #s]) / pI[#i][-1];
```

In this case, only `#i` and not `#s` is uncontrolled, but the machine can easily detect this, too (this auto-detection is already done regarding array-series statements). So it is more a question of whether `EQU[#i]` helps the user to see that the equations are unrolled over `#i` and not any other sets (in this example, the left-hand side variable `qI[#i]` indicates it too, but the equation could just as well be in more implicit GAMS-like form, with a long expression = 0.

In GAMS-like equations, the equation name can be used to indicate the “dependent” variable (granted that this makes sense), if this variable is not easily identifiable from the left-hand side expression (or if it is, for instance, buried somewhere on the right-hand side expression).

Formula codes could be included too, for instance:

```
EQU <code = _GJRD>  #ey   dlog(y)  =  0.5 * dlog(x)  +  0.5 * dlog(x[-1]);
```

In practice, setting the equation name like this would probably be more convenient than requiring a name field like the following:

```
EQU <name = ‘#ey’ code = _GJRD>  #ey   dlog(y)  =  0.5 * dlog(x)  +  0.5 *
dlog(x[-1]);
```

It would perhaps be wise to require that the equation name starts with `#`. For long files of equations the `#` symbol makes it clearer where the name is. A description could also be added, but perhaps this is better done inside the `<>` field, for instance:

```
EQU <code = _gjrd label = ‘Total quantity’>
                           #ey   dlog(y)  =  0.5 * dlog(x)  +  0.5 * dlog(x[-1]);
```

In practice, such labels will probably be handled elsewhere than in an EQU definition. It should be possible to ‘glue’ them on afterwards, like `#ey.%label = ‘Total quantity’;`.

An EQU statement in a command file would not run any series expressions in itself, and the variables do not need to exist when the EQU is defined. In this sense, an EQU is a bit like a user-defined function.

We could envision a model object being constructed from three equations:

```
#simple = model(#eq1, #eq2, #eq3);
```

Here `#m` would be a model object, constructed from three equation objects. Instead it could be constructed from a list of equation names, for instance:

```
#eqs = #eq1, #eq2, #eq3;  //three strings (‘#eq1’, ‘#eq2’, ‘#eq3’)
#simple = model({#eqs});
```

Note the `{}`-parenteses. Without them, the model() function would be fed with a list of strings, where it expects to be fed with a list of equation objects.

This could be developed, implementing other helper functions to deal with models and equations. It should also be possible to construct a model from a submodel + equations, and nested submodels should be possible.[9]

There should be functions available to obtain the names of submodels and equations belonging to a given model, and inversely, there should be functions available to indicate the ‘parent’ model of a given submodel or equation.

Constructing a model with internal submodel structure:

```
equ <code = _i>  y = c + i + g + x;
equ <code = _s>  c = 0.6 * y;
equ <code = _s>  i = 0.2 * y + i0;
equ <code = _d>  g = g0 + jg;
equ <code = _i>  s = y – c;
#identities = model(#y, #s);
#stochastic = model(#c, #i);
#simple = model(#identities, #stochastic, #g);
```

If no equation name is given, the first equation becomes #y, but else the name can be stated:

```
equ <code = _i>  #y  y = c + i + g + x;
```

We can envision inbuilt function that deal with models and equations, for instance:

```
replaceeq(#simple.#c, #simple2.#c);
```

This replaces the equation `#simple.#c` with the equation `#simple2.#c`. Similar functions could handle other aspects of the handling of models, submodels and equations.

Because Gekko 3.0 supports so-called Uniform Function Call Syntax (UFCS), the statement can alternatively be written as this:

```
#simple.#c.replaceeq(#simple2.#c);
```

Such helper functions could be added to Gekko when needed.

[9] Gekko should remember the order in which the equations and submodels are added.

3.3 Lists of equations

Lists of equations are expected to be prevalent in user command files, because such lists can be used to construct models and submodels. Similarly, the user will often construct lists of timeseries, for all kinds of uses. In the latter case, there are two possibilities regarding such lists, (1) create a list of the series objects proper, or (2) create a list of series *names*. In many cases, the latter possibility is easier to manage, for instance such a list of *names* can be used to compare timeseries in two databanks, where the same name is looked up in both these databanks. Also, one can use the large number of functions that deal with such (lists of) names, making it easy to add/remove the frequency part of names, add/remove databank part of a name, etc. (and in addition, the user can use normal string manipulation functions to 'wrangle' the series names at will). When using such a list of names, the user must use {}-parentheses when referring to the object corresponding to the name, for instance `PRT {#m};` to print out the timeseries corresponding to the elements of `#m`, rather than their raw names (as strings). This allows for easy syntax like `PRT b1:{#m}, b2:{#m};` for printing out the series of the list `#m` (a list of strings representing series names) from two different databanks. It *is* possible to store series objects themselves inside a list, but in practice it is often more convenient to operate on the *names* of the series.

A list of equations could look like the following (using a 'naked' list, that is, a list definition without parentheses):

```
#eq1 = #e1, #e2;
```

where `#eq1` will be equal to the list of strings `('#e1', '#e2')`. So this would be a list of *names*, not of equation objects. To create a list of equation objects, the user could use `(#e1, #e2)` but in many cases it is better to operate on lists of names.

Lists of such names can be combined, for instance:

```
#eq1 = #e1, #e2;  //list of strings
#eq2 = #e3, #e4;  //list of strings
#eqs = {#eq1}, {#eq2}; //list of strings
```

Note that the {}-parentheses tell Gekko to fetch the *contents* of `#e1` and `#e2`, so that `#eqs` becomes a list of the four strings `('#e1', '#e2', '#e3', '#e4')`. If these `{}`-parenteses were omitted, `#eqs` would become a list of the two strings `('#eq1', '#eq2')` instead.[10]

[10] The following non-naked list statement `#eqs = (#eq21, #eqs2);` would yield the same.

3.4 Model files

Model files are a practical way of storing equations in one place, representing a tree structure of model, submodels, and equations. Storing the information like this also makes it less likely that model parts are accidentally changed, and hash codes (digital signatures) can keep track of such integrity. In addition, model equations and data generation is not mixed into one confusing mess.

The mirror image of EQU statements in .gcm files is FRML statements in .frm files. In .frm files, the user cannot operate on model and equation objects in the same completely free manner, but must instead use descriptive syntax like the following:

```
model #simple;
  model #identities;
    frml  _I  Y = C + I + G + X;
    frml  _I  S = Y – C;
  endmodel #identities;
  // -----------------------------------
  model #stochastic;
    frml  _S  C = 0.6 * Y;
    frml  _S  I = 0.2 * Y + I0;
  endmodel #stochastic;
  // -----------------------------------
  frml  _D  G = G0 + JG;
endmodel #simple;
```

Here, the model consists of two submodels, and one “rogue” equation (`G`). The statements are indented for readability here, but would typically not be indented in a real model file. Therefore, and because submodel sections may span many lines, it is perhaps wise to have an ending tag with a name, rather than just `endmodel;`. Using the `model` tag for both the ‘parent’ and ‘child’ model (rather than using for instance the tags `model` and `submodel`) makes it easier to ‘promote’ a submodel into a ‘parent’ model or vice versa, just by cutting and pasting. No name needs to be provided regarding the uppermost model, which could be simply enclosed inside `model; ... ; endmodel;` tags, or the tags could be completely omitted. In that case, the model object uses the filename, for instance transforming the file simple.frm into the model object `#simple`.

The MODEL command looks like the following, when fetching a file:

```
MODEL simple.frm;
```

or just

```
MODEL simple;
```

which will read the simple.frm file, and the model variable name would become `#simple`, unless a different name is stated in the first `model` tag inside the the file. Per default, the model object will be put into the Global databank, so that it will survive for instance READ and CLEAR statements. Equation objects are not produced from a model file and put into the Global databank, unless the users ask Gekko to do this (this is to avoid clutter).

To simulate the model, we would have to do the following:

```
SIM #simple;
```

If only one model is present in the Global databank, it should be possible to simply write

```
SIM;
```

where Gekko will look for an unique model in the Global databank.

When loading the model, it will be possible to fetch (sub)models from different files and combine them. For instance:

```
MODEL #m = m1 <removemodel = #stochastic>,  m2 <setmodel = #stochastic>;
```

This will combine the files m1.frm and m2.frm. From m1.frm, the submodel `#stochastic` is removed, and from m2.frm only the submodel `#stochastic` is fetched. In essence, this is the same as replacing the `#stochastic` submodel, apart from the ordering of equations, where `#stochastic` is now at the end of the combined model. This ordering is only of relevance when using the Gauss-Seidel solving algorithm (not the Newton algorithm). Perhaps syntax like the following could do a replace while preserving equation order:

```
MODEL #m = m1, replacemodel #stochastic from m2;
```

It should be possible to use commas like the following:

```
MODEL #m = m1, m2, m3;
```

combining the three files m1.frm, m2.frm, and m3.frm into one model.

If a sub-submodel needs to be added/removed/replaced, it could be done as follows (here the submodel names also differ):

```
MODEL #m = m1, replacemodel #trade.#imports with #tradenew.#imports from m2;
```

Note that `m1` and `m2` are filenames, and these can therefore be long with paths etc. When loaded under for instance the model name `#m1`, references can be more compact with dot notation, like `#m1.#trade.#imports`.

After reading the model file, the 'parent' model would be stored under the given name (for instance `#m`) in the Global databank, and for instance, the `#imports` sub-submodel shown above could be called with `disp #m.#trade.#imports`, which prints out info on that sub-submodel (number of variables and other info).

Replacing/removing/adding individual equations should also be possible, for instance:

```
MODEL #m = m1 <replaceequ #gdp from m2>;
```

3.5 Correspondence between .gcm models and .frm models,

I will be possible to build a model either in pure code (.gcm command file), or with a tree structure (.frm model file).

I will probably be practical to be able to turn a model (understood as a Gekko model object) into a tree structure (.frm model file). In this way, more complicated parts of a model can be generated in pure code, and also estimated with OLS. When the model is ready, it would be practical to be able to make a "snapshot" of this object, in the form of a model (.frm) file.

4. More

- User-defined functions. It would be practical to be able to define and write for instance `ces(x1, x2, kappa, delta, sigma)` in model files and equations, where ces() is a user-defined function that represents the CES production function. Such functions should perhaps only be able to use values as input, and return values as output.
- How to store parameter values (like `%sigma1` etc.) not written explicitly as fixed values in the equations? Should they be present in the .frm file, or could they be loaded dynamically from a databank?
- Functionality regarding implicit "static models", finding or defining long-run equilibrium values ("structural levels") where dynamics are ignored.
- Signing. How is this done with models, submodels, etc.? How to sign models originating from command files (pure code)?
- More support for seasonality in for instance quarterly models?
- Metadata in models, like lists of damped equations etc.
- AFTER, AFTER2 support? Or can this be done with submodels or lists of equations?
- More work on special kinds of equations like T-, Y-, P-type equations, and the possibility of running gcm files before or after a simulation.

5. Conclusions

Preliminary conclusions:

All this needs to simmer for a while, but thinking about it, a model with submodels, sub-submodels, etc., is structurally a lot like a Gekko map, and also very similar to a Gekko databank. Gekko uses the symbol `#` for all "collections", whereas `%` is reserved for scalars, and symbol-free for timeseries. Therefore, it makes a lot of sense that models are referenced to with `#`, for instance `#m1`. Similarly, equations should use `#`, too, for instance `#e1`. Then `#m1.#e1` is equation 1 from model 1, and `b1:#m1.#e1` is the same, but this time `#m1` is found in the `b1` databank. Models should typically be put in the Global databank, so that they survive READ, CLEAR etc.

Searching, indexing, copying, deleting etc. should be relatively easy to implement, because models and equations are just new objects alongside the other Gekko objects. Therefore, most of the existing syntax and features regarding this (dealing with objects residing in databanks), can be reused.

All this should be developed in conjunction with maps, so that indexing a submodel for its submodels and equations is similar to indexing a map for its sub-maps and other objects. The syntax ought to be the same, and the underlying code ought to be shared. This makes the implementation less resource-intensive, and provides a unified experience for the users, so that they feel that they are doing the same kind of operations (with similar syntax) whether they are operating on databanks, on maps, or on models/equations.

It is probably wise to offer two ways of writing a model, (a) as a tree structure (.frm file with submodels), or (b) in more dynamic pure code (in a .gcm file). In the latter case, OLS can also be used to obtain parameter values. Sometimes, parts of a model can be as .frm, and other parts more dynamically constructed in pure code. Perhaps it would be convenient to be able to take a "snapshot" of a given model object and write it out as a .frm file (for instance for deployment of a model for its end-users).

Some syntax suggestions regarding .frm files are given in section 3.4, but the main idea is to use designators like `model` `model #identities; ... ; ... ; endmodel #identities;` The use of `#` in the names is a nod to the naming of the objects after being loaded by Gekko. Both models and submodels use `model` tag, because the distinction is not very important (a submodel *is* a model in its own right).

When loading frm files with the MODEL command, it should be possible to combine parts of (submodels of) frm files, using a syntax tailored to that end. This syntax is under constructions, but some ideas are presented in section 3.4.