

Sparse Model Matrices for Multidimensional Hierarchical Aggregation

by Øyvind Langsrud

Abstract Multidimensional hierarchical sum aggregations can be formulated as matrix multiplications involving dummy matrices which can be referred to as model matrices. In contrast to standard model matrices, all categories of all variables must be included. For this purpose, the R package *SSBtools* includes functionality to create model matrices in two alternative ways, by model formulas or by so-called hierarchies. The latter means a coding of hierarchical relationships, and this can be done in several ways. Tree-shaped hierarchies are not required. The internal standard in the package is a parent-child coding. Functionality to find hierarchies automatically from the data is also included. The model matrix functionality is applied in several R packages for statistical disclosure control. This enables general implementation of methods and a flexible user interface. This paper describes the model matrix and hierarchy functions in *SSBtools*, as well as the methods and functions behind it.

1 Introduction

In general, a vector of sum aggregates (z) can be calculated from a data vector (y) through a dummy matrix (x) by

$$z = x^T y \quad (1)$$

The matrix x can be referred to as a model matrix. In package *SSBtools* (Langsrud and Lupp 2023b) there are several tools for creating such model matrices and for computing aggregates via such matrices. This article focuses on these tools, while there are additional tools included in the package that are not addressed herein. Note that the package gathers functions that are used by other packages for specific purposes.

For some applications it is important to have access to the model matrix. In other applications, the interface or the computational efficiency is what is needed. For efficiency, the aggregates may sometimes be computed by

$$z = x_1^T y x_2 \quad (2)$$

where y is a matrix of input data that is appropriately reorganized into multiple columns and where x_1 and x_2 are two dummy matrices.

An important part of the model matrix framework within *SSBtools* is the handling of hierarchical relationships. That is, the categorical variables in the input data are hierarchically related. Or, alternatively, some codings of hierarchical relationships, such as parent-child, are supplied as separate input. We refer to these codings as hierarchies and these are not limited to being tree-shaped. Hierarchies are used both in the process of constructing the model matrix and also to organize the categorical variables in the aggregated output data.

Model matrices are usually associated with regression models which can be expressed by model formulas and fitted by the `lm` function or by other more advanced functions. In this article, we will also use model formulas to specify model matrices. For this purpose, only the right-hand side of a model formula is needed. A formula consisting of the variables a and b including the interaction ($a:b$) can be written as $\sim a + b + a:b$, or equivalently as $\sim a * b$. By using model formulas, we can take advantage of the asterisk sign and other possibilities to write comprehensive models in compact ways (see `?formula` in an interactive R console). The intercept term, which is included by default, can be removed by subtracting a one ($\sim a * b - 1$). In the model matrix, the intercept term is a column of ones. In our context, we would rather refer to this model term as the overall total.

Most elements of the model matrices considered in this paper will be zero. When using regular dense matrices, as created by the base-R function `matrix`, an unnecessary amount of memory is used to store all the zeros. Instead, we use sparse matrices defined in the *Matrix* package (Bates, Maechler, and Jagan 2023). Then the matrices are stored in a compressed format, while ordinary matrix operations can still be performed. In many cases, the matrices will be so large that applications which do not utilize sparse matrices will be impossible in practice. Please note that the functions discussed in this article do not create any new classes. The primary outputs of these functions include sparse matrices

according to [Matrix](#), data frames, or a combination of both in the form of lists.

The rest of this paper is structured as follows. First, we describe the function, `ModelMatrix`, for generating model matrices. The following section takes a closer look at some underlying computations. Then, a section is devoted to a more advanced function for two-way computation. Within this function, equation (2) is applied. The next section looks at aggregation beyond summation, such as median calculation. Then, there is a section where various alternatives for creating model matrices and aggregates are compared in terms of both time and memory usage. In the penultimate section, we consider applications in other packages. Finally, a short conclusion is given.

The theory in this paper is described using examples and R code.

2 The function `ModelMatrix`

The base-R function `model.matrix` is a workhorse function for constructing model or design matrices used in regression modeling. To create sparse matrices, the corresponding function `sparse.model.matrix` in the [Matrix](#) package (Bates, Maechler, and Jagan 2023) can be used. When all variables involved are categorical, the model matrices are dummy matrices of zeroes and ones. The function `ModelMatrix` in package [SSBtools](#) is an alternative function with a slightly different purpose. The aim is to represent the transformation from input to output when producing various types of aggregated data. As described below, the model matrix can be constructed with both a formula interface and a hierarchy interface.

2.1 Model matrix from formula

In the examples below, `d` is a data frame.

```
> d
```

```
      age      geo      eu value
1 young   Spain    EU   66.9
2 young  Iceland nonEU   1.8
3 young Portugal  EU   11.6
4 old     Spain    EU  120.3
5 old  Iceland nonEU   1.5
6 old  Portugal  EU   20.2
```

A model matrix can be specified with a formula.

```
> ModelMatrix(d, formula = ~age + geo)
```

```
      Total-Total old-Total young-Total Total-Iceland Total-Portugal Total-Spain
[1,]           1         .           1             .             .           1
[2,]           1         .           1             1             .           .
[3,]           1         .           1             .             1           .
[4,]           1         1           .             .             .           1
[5,]           1         1           .             1             .           .
[6,]           1         1           .             .             1           .
```

In contrast to standard model matrices, all categories are included. All variables are considered categorical. The column names correspond to rows in a data frame, called `crossTable`, which can be requested as output. Such a model matrix can be used to compute aggregates according to (1), for example by:

```
> t(ModelMatrix(d, formula = ~age + geo)) %*% d$value
```

```
      [,1]
Total-Total  222.3
old-Total    142.0
young-Total   80.3
Total-Iceland   3.3
Total-Portugal  31.8
Total-Spain   187.2
```

Exactly the same aggregates can be computed more directly using a related [SSBtools](#) function: `FormulaSums(d, formula = value~age + geo)`. With this function, the aggregates are computed somewhat faster when the data is large. Internally, each model term triggers a call to the function `aggregate`. Correspondingly, `ModelMatrix` calls the `Matrix` function `fac2sparse` repeatedly. Note that the formula interface within `aggregate` is different from our approach. The formula `~age + geo` interpreted by `aggregate` corresponds to `~age:geo - 1` interpreted by `ModelMatrix` or `FormulaSums`. So the formula interface in `aggregate` is less flexible. For example, `aggregate(value ~age + geo, d, sum)` and `aggregate(value ~age * geo, d, sum)` give the same results.

To illustrate hierarchical variables, we proceed with a more complicated formula:

```
> ModelMatrix(d, formula = ~age*eu + geo, crossTable = TRUE)

$modelMatrix

[[ suppressing 12 column names 'Total-Total', 'old-Total', 'young-Total' ... ]]

[1,] 1 . 1 1 . . . 1 . . 1 .
[2,] 1 . 1 . 1 1 . . . . 1
[3,] 1 . 1 1 . . 1 . . . 1 .
[4,] 1 1 . 1 . . . 1 1 . . .
[5,] 1 1 . . 1 1 . . . 1 . .
[6,] 1 1 . 1 . . 1 . 1 . . .

$crossTable
      age      geo
1 Total    Total
2  old    Total
3 young    Total
4 Total      EU
5 Total   nonEU
6 Total  Iceland
7 Total Portugal
8 Total   Spain
9  old      EU
10 old   nonEU
11 young   EU
12 young  nonEU
```

The column names suppressed from the `$modelMatrix` output can be seen in `$crossTable`. Here there are two variables in `crossTable` even though there are three in the formula. This is because it has automatically been discovered that `geo` and `eu` are hierarchically related. Combining hierarchical variables in this way is often useful and this is common within official statistics. Automatic detection and handling of hierarchical relationships can be turned off with the `avoidHierarchical` parameter. The calculations made in the background are related to the automatic generation of hierarchies discussed below.

Note that `ModelMatrix` does not define any new classes. The dummy matrix will have a sparse matrix class according to [Matrix](#). Though, an attribute called `startCol` has been added to make locating individual model terms easy and a related function, `FormulaSelection`, is available in [SSBtools](#).

2.2 Model matrix from hierarchies

From the data we can generate hierarchies coded as trees:

```
> dimLists <- FindDimLists(d[c("age", "geo", "eu")])
> dimLists

$age
  levels codes
1      @ Total
2      @@  old
3      @@ young
```

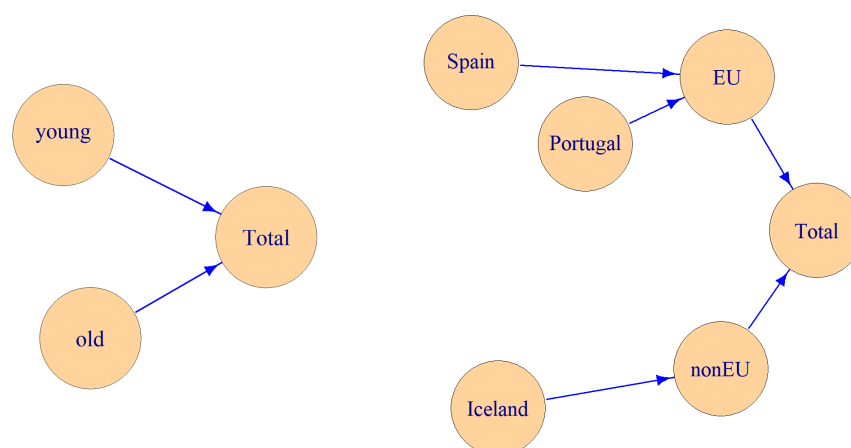


Figure 1: The hierarchies in the examples.

```

$geo
  levels  codes
1      @   Total
2     @@    EU
3    @@@ Portugal
4     @@@   Spain
5     @@   nonEU
6    @@@  Iceland

```

The two hierarchies are illustrated in Figure 1. This way of coding hierarchies follows the standard used in the R package [sdcTable](#) (Meindl 2023), which is a tool for statistical disclosure control (SDC) (Hundepool et al. 2012). A model matrix can be constructed from the data and the hierarchies.

```

> ModelMatrix(d[-6, -3], hierarchies = dimLists, inputInOutput = c(TRUE, FALSE),
+             crossTable = TRUE)

```

```

$modelMatrix

```

```

[[ suppressing 9 column names 'Total:Total', 'Total:EU', 'Total:nonEU' ... ]]

```

```

[1,] 1 1 . . . 1 1 .
[2,] 1 . 1 . . . 1 . 1
[3,] 1 1 . . . . 1 1 .
[4,] 1 1 . 1 1 . . . .
[5,] 1 . 1 1 . 1 . . .

```

```

$crossTable

```

```

  age  geo
1 Total Total
2 Total  EU
3 Total nonEU
4  old Total
5  old  EU
6  old nonEU
7 young Total
8 young  EU
9 young nonEU

```

By default, with hierarchy input, all possible combinations are made. Here, parameter `inputInOutput` is used to prevent columns being constructed from input codes for `geo` (country names). The `eu` column

(3rd) was removed from the input data to illustrate that this column is not being used. Instead, the information about EU and nonEU is now given in the hierarchy. The last row (6th) was also removed to illustrate that complete data is not required. Either way, the rows in the output correspond to the rows in the input.

An alternative coding of the above hierarchies follow the standard used in the SDC program called τ -Argus (Wolf et al. 2014). Conversion to and from this standard is possible:

```
> hrc <- DimList2Hrc(dimLists)
> hrc

$age
[1] "old"    "young"

$geo
[1] "EU"      "@Portugal" "@Spain"    "nonEU"    "@Iceland"
```

However, the standard we use within the framework of ModelMatrix is more general and tree-shaped hierarchies are not required. By the function AutoHierarchies, hierarchies coded in several ways are converted to our internal standard. As shown below, it is also possible to specify total codes.

```
> hi <- AutoHierarchies(hrc, total = c("All", "Europe"))
> hi

$age
  mapsFrom mapsTo sign level
1      old    All      1      1
2    young    All      1      1

$geo
  mapsFrom mapsTo sign level
1      EU Europe      1      2
2 Portugal    EU      1      1
3   Spain    EU      1      1
4 nonEU Europe      1      2
5 Iceland nonEU      1      1
```

The variable names mapsFrom and mapsTo were chosen because the starting point was to reprogram an application implemented in the Validation and Transformation Language (VTL) described in Airo et al. (2015). The level variable indicates the order in which the codes must be calculated. This variable can be computed automatically and is not needed in input. Sign, as coded in the sign column, can also be negative (-1), which means subtraction instead of addition. The hierarchies can be very general (but not circular) and it is possible that the final model matrix consists of values beyond zeros and ones. A dummy matrix can be ensured by unionComplement = TRUE. Then, in accordance with VTL, sign is interpreted as union or complement instead of addition or subtraction.

Another possible coding that can be input to AutoHierarchies is formulas. This should not be confused with model formulas in the section above. Conversion to formulas from the internal standard is possible.

```
> Hierarchies2Formulas(hi)

$age
[1] "All = old + young"

$geo
[1] "Europe = EU + nonEU"    "EU = Portugal + Spain" "nonEU = Iceland"
```

By using the parameter select, the columns in the model matrix, i.e. crossTable, can be specified in advance. This can be much more efficient than making the selection afterwards.

```
> ModelMatrix(d, hierarchies = hi, select = data.frame(
+   age = c("young", "young", "All", "All"),
+   geo = c("EU", "nonEU", "nonEU", "Europe")))
```

```

      young:EU young:nonEU All:nonEU All:Europe
[1,]      1      .      .      1
[2,]      .      1      1      1
[3,]      1      .      .      1
[4,]      .      .      .      1
[5,]      .      .      1      1
[6,]      .      .      .      1

```

With few rows in input, many columns in the model matrix will only consist of zeros. As illustrated below, such columns can be omitted by the parameter, `removeEmpty`.

```
> ModelMatrix(d[c(1, 4), ], hierarchies = hi, removeEmpty = TRUE)
```

```

      All:Europe All:EU All:Spain old:Europe old:EU old:Spain young:Europe
[1,]      1      1      1      .      .      .      1
[2,]      1      1      1      1      1      1      .
      young:EU young:Spain
[1,]      1      1
[2,]      .      .

```

Due to `removeEmpty`, nine columns were omitted ("All", "old" and "young" crossed with "nonEU", "Iceland" and "Portugal"). In large real data sets, the effect of `removeEmpty` can be enormous.

Two special possibilities are that the hierarchies can be encoded as a string or as an empty string. A string is considered a total code. An empty string means that the variable is considered a pure categorical variable without a hierarchy.

```
> ModelMatrix(d[c(1,2,4), ], hierarchies = list(age = "", geo = "Europe"))
```

```

      old:Europe old:Iceland old:Spain young:Europe young:Iceland young:Spain
[1,]      .      .      .      1      .      1
[2,]      .      .      .      1      1      .
[3,]      1      .      1      .      .      .

```

In this case only two countries are present in the input data. If `removeEmpty` had been `TRUE`, the `old:Iceland` column would have been omitted.

Note that both `ModelMatrix` parameters, `hierarchies` and `formula`, can be used simultaneously. How the hierarchies are to be crossed is then defined by the formula.

3 Underlying computations

3.1 Hierarchies automatically from the data

To find hierarchies automatically from the data, a kind of correlation matrix is first computed by the function `FactorLevCorr`. The function was named with factor variables and their levels in mind, but any variable type is possible. To illustrate a little more complexity, we add `isSpain` as a logical variable.

```

> d$isSpain <- d$geo == "Spain"
> fCorr <- FactorLevCorr(d[c("age", "geo", "eu", "isSpain")])
> fCorr

```

```

      age geo  eu isSpain
age      2  0  0.0    0.0
geo      0  3 -1.0   -1.0
eu       0  1  2.0    0.5
isSpain  0  1 -0.5    2.0

```

The diagonal elements in this matrix are used to store the number of unique values of each variable (n_i). To calculate our correlations, we first find the number of unique combinations (m_{ij}) of each pair of variables. For this purpose, the function, `unique`, is repeatedly called with two-column data frames as input. The absolute values of off-diagonal elements are 0 when $m_{ij} = n_i * n_j$, 1 when $m_{ij} = \max(n_i, n_j)$

and otherwise computed as $(n_i * n_j - m_{ij}) / (n_i * n_j - \max(n_i, n_j))$. So 0 means that all possible level combinations exist in the data and 1 means that the two variables are hierarchically related. In our example the correlation between eu and isSpain isn't zero since the combination (nonEU, TRUE) is missing in the data. The signs of off-diagonal elements are set to be positive when $n_i < n_j$ and negative when $n_i > n_j$. In cases where $n_i = n_j$, elements will be positive above the diagonal and negative below. Values other than 0, -1 and 1 could be useful for detecting errors in the data. For example, values close to one may be suspicious.

In our application, to generate hierarchies, we only need to look for ones. A one means that two hierarchically related variables are found. More generally, one can find trees by connecting relations when the variables are ordered according to the diagonal elements. The function, `HierarchicalGroups`, does this repeatedly using a recursive algorithm so that all possible trees are found. Below we run this function with `eachName = TRUE` so that names are included instead of indices.

```
> HierarchicalGroups(fCorr = fCorr, eachName = TRUE)
```

```
$age
[1] "age"
```

```
$geo
[1] "geo" "eu"
```

```
$geo
[1] "geo"      "isSpain"
```

Here two groups were named geo. This means also that `FindDimLists` will result in two hierarchies named geo. Each hierarchy is achieved by looking at sorted unique rows. For example, this hierarchy,

```
> FindDimLists(d[c("age", "geo", "eu", "isSpain")])[[3]]
```

```
levels codes
1      @   Total
2     @@  FALSE
3    @@@ Iceland
4    @@@ Portugal
5     @@   TRUE
6    @@@   Spain
```

was constructed from this data frame:

```
> SSBtools::SortRows(unique(d[c("isSpain", "geo")]))
```

```
isSpain geo
2 FALSE Iceland
3 FALSE Portugal
1  TRUE   Spain
```

The top level (@) is always the total code. The remaining levels correspond to the columns in the data as defined by the hierarchical groups. Here this means "isSpain" (@@) and "geo" (@@@).

Note that the corresponding function `FindHierarchies` produces only a single geo hierarchy.

```
> FindHierarchies(d[c("age", "geo", "eu", "isSpain")])$geo
```

```
mapsFrom mapsTo sign level
1 Iceland nonEU    1      1
2 Iceland Total    1      1
3 Iceland FALSE    1      1
4 Portugal  EU      1      1
5 Portugal Total    1      1
6 Portugal FALSE    1      1
7  Spain    EU      1      1
8  Spain    Total    1      1
9  Spain    TRUE     1      1
```

FindHierarchies wraps FindDimLists and AutoHierarchies into a single function. By default, when multiple hierarchies exist for the same variable, AutoHierarchies combines them by generating a flattened hierarchy with a single level. The output codes are the parents (mapsTo) and all childs (mapsFrom) are input codes. Parent-child relationships corresponding to multilevel tree structures are not directly available, but the information needed to create model matrices is available. In fact, the hierarchies are combined via dummy matrices described below.

3.2 From hierarchies to dummy matrices

To produce model matrices, the hierarchies are converted to dummy matrices and the parameter inputInOut is used. Here we convert the hierarchies, hi, in the previous section:

```
> hiM <- DummyHierarchies(hi, inputInOut = c(TRUE, FALSE))
> hiM
```

```
$age
      old young
old      1    .
young    .    1
All      1    1

$geo
      Iceland Portugal Spain
EU      .            1     1
nonEU    1            .     .
Europe   1            1     1
```

The first two rows of hiM\$age are a perturbed identity matrix since inputInOut = TRUE for this variable. The first two rows of hiM\$geo can be found directly from the level 1 rows of the hierarchy, hi\$geo. The nonzero elements are taken from the sign variable and the corresponding rows and columns can be read from the variables mapsFrom and mapsTo, respectively. A similar matrix can be constructed from level 2. In our example this matrix is:

```
> geo2 <- Matrix(1, 1, 2, dimnames = list("Europe", c("EU", "nonEU")))
> geo2
```

```
      EU nonEU
Europe 1     1
```

The last row of hiM\$geo can now be found by the matrix multiplication:

```
> geo2 %*% hiM$geo[1:2, ]
```

```
      Iceland Portugal Spain
Europe      1          1     1
```

In general, the dummy matrix is constructed successively by adding one level at a time. The new rows are found as a matrix constructed from the new level multiplied by the preliminary dummy matrix. The parameter unionComplement mentioned above is used within this process.

The next step is to extend these matrices to match the data and this can be achieved by the function DataDummyHierarchies:

```
> hiD <- DataDummyHierarchies(d, hiM, colNamesFromData = TRUE)
> hiD
```

```
$age
      young young young old old old
old      .    .    .    1    1    1
young    1    1    1    .    .    .
All      1    1    1    1    1    1

$geo
```


	Spain	Iceland	Portugal	Spain	Iceland	Portugal
EU	1	.	1	1	.	1
nonEU	.	1	.	.	1	.
Europe	1	1	1	1	1	1

For readability `colNamesFromData` was here set to `TRUE`. This also means that the relationship to `hiM` is direct. Now a transposed model matrix can be achieved by the function `Khatr iRao` (column-wise Kronecker product) within the `Matrix` package.

```
> Matrix::Khatr iRao(hiD$geo, hiD$age, make.dimnames = TRUE)
```

	Spain	Iceland	Portugal	Spain	Iceland	Portugal
old:EU	.	.	.	1	.	1
young:EU	1	.	1	.	.	.
All:EU	1	.	1	1	.	1
old:nonEU	1	.
young:nonEU	.	1
All:nonEU	.	1	.	.	1	.
old:Europe	.	.	.	1	1	1
young:Europe	1	1	1	.	.	.
All:Europe	1	1	1	1	1	1

Now the column names are misleading since only names from the first input matrix are used. However, the row names can be useful. With more than two variables, `Khatr iRao` is run several times by adding one variable at a time.

We now recall how a model matrix with pre-selected columns was created in the previous section using the `select` parameter to `ModelMatrix`. When we look at dummy matrices here, this means pre-selected rows. To produce such a matrix with pre-selected rows, one possibility is to first produce separate matrices for each variable.

```
> hiD$age[c("young", "young", "All", "All"), ]
```

	young	young	young	old	old	old
young	1	1	1	.	.	.
young	1	1	1	.	.	.
All	1	1	1	1	1	1
All	1	1	1	1	1	1

```
> hiD$geo[c("EU", "nonEU", "nonEU", "Europe"), ]
```

	Spain	Iceland	Portugal	Spain	Iceland	Portugal
EU	1	.	1	1	.	1
nonEU	.	1	.	.	1	.
nonEU	.	1	.	.	1	.
Europe	1	1	1	1	1	1

The transposed model matrix is then obtained by multiplying these matrices together. This is a fast way to obtain the model matrix. A possible problem is, however, that the matrices to be multiplied are not as sparse as the final matrix. Therefore `Khatr iRao` combined with reducing the matrices as much as possible first, can still be better. The choice between the two methods is controlled by a parameter, called `selectionByMultiplicationLimit`. The functionality of the parameter `removeEmpty` mentioned above, is implemented by `Khatr iRao` combined with reducing the matrices.

4 Two-way computation

4.1 The function `HierarchyCompute`

We recall that sum aggregates of a data vector (y) can be calculated by equation (1) where the model matrix (x) can be produced by the function `ModelMatrix`. As mentioned above, with formula input an alternative function is `FormulaSums`. Then the aggregates are calculated without x being created. In the cases of hierarchies, a corresponding function is `HierarchyCompute`. Although this function

makes aggregates via one or two dummy matrices, a model matrix that matches the input data is not necessarily made. A special case of `HierarchyCompute` is to make such a model matrix and this matrix can also be requested as output. Actually, `ModelMatrix` with hierarchy input make use of `HierarchyCompute`. The `HierarchyCompute` function does not take many hierarchy types as input, so `AutoHierarchies` may have to be run first.

In general, aggregates are computed by `HierarchyCompute` according to equation (2). But the original version of the function, without the `colVar` parameter, is limited to equation (1). However, y may be a multi-column matrix containing a reorganized version of the input data vector. Then the columns represent a single categorical variable with no hierarchy. To tell `HierarchyCompute` to treat a categorical variable in this way, this categorical variable is encoded as the string "colFactor" in the hierarchy input. Other categorical variables are encoded as "rowFactor". The function `AutoHierarchies` re-encodes an empty string to "rowFactor".

To illustrate this function, we add an extra variable, year, to the data. Although, it is only a single year.

```
> HierarchyCompute(data = cbind(d, year = 2022),
+   hierarchies = list(age = "colFactor", geo = hi$geo, year = "rowFactor"),
+   valueVar = "value")
```

```
   age   geo year value
1  old    EU 2022 140.5
2  old nonEU 2022   1.5
3  old Europe 2022 142.0
4 young   EU 2022  78.5
5 young nonEU 2022   1.8
6 young Europe 2022  80.3
```

Except for column and row order, the output will be the same if "rowFactor" and "colFactor" are swapped or if both age and year are set to "rowFactor". If output = "matrixComponents" is included in the function call, the internal differences can be seen. Then, in our example, output become:

```
> hc <- HierarchyCompute(data = cbind(d, year = 2022),
+   hierarchies = list(age = "colFactor", geo = hi$geo, year = "rowFactor"),
+   valueVar = "value", output = "matrixComponents")
> hc
```

```
$dataDummyHierarchy
```

```
EU:2022      . 1 1
nonEU:2022   1 . .
Europe:2022  1 1 1
```

```
$valueMatrix
```

```
      old young
[1,]   1.5   1.8
[2,]  20.2  11.6
[3,] 120.3  66.9
```

```
$fromCrossCode
```

```
      geo year
1 Iceland 2022
2 Portugal 2022
3   Spain 2022
```

```
$toCrossCode
```

```
      geo year
1     EU 2022
2 nonEU 2022
3 Europe 2022
```

The matrix, `valueMatrix` is a reorganized version of the value variable in the input data. If a variable is selected as "colFactor", there is one column for each level of that variable. Even without such a variable, there may be fewer rows in the `valueMatrix` than in the input data. This is because

rows where the values are zero can be removed and because duplicated code combinations in the input can be aggregated. This is controlled by parameters "reduceData" and "handleDuplicated". The data frame fromCrossCode contains variables that characterize the rows of valueMatrix. The matrix dataDummyHierarchy is a transposed model matrix that matches valueMatrix. This means that the aggregate output values can be calculated as dataDummyHierarchy %*% valueMatrix:

```
> hc$dataDummyHierarchy %*% hc$valueMatrix
```

```
      old young
EU:2022  140.5  78.5
nonEU:2022  1.5   1.8
Europe:2022 142.0  80.3
```

The data frame toCrossCode contains variables that characterize the rows of this matrix. Along with column names, this is the information needed to reorganize the aggregate result into a regular output data frame.

4.2 HierarchyCompute with colVar

When the parameter colVar is applied, two (transposed) model matrices are made, one for rows and one for columns. Parameter colVar splits the hierarchy variables in two groups and this variable overrides the difference between "rowFactor" and "colFactor". Actually, there will be two runs of HierarchyCompute. With output = "matrixComponents", output from the two runs are returned as a list with elements hcRow and hcCol. Below we illustrate this by including the age hierarchy with inputInOutput = TRUE (FALSE is default).

```
> hc2 <- HierarchyCompute(data = cbind(d, year = 2022),
+   hierarchies = list(age = hi$age, geo = hi$geo, year = "rowFactor"),
+   colVar = "age", inputInOutput = c(TRUE, FALSE),
+   valueVar = "value", output = "matrixComponents")
```

For rows, we obtain the same matrices as earlier. The aggregates in the previous subsection can now be computed as:

```
> hc2$hcRow$dataDummyHierarchy %*% hc2$hcRow$valueMatrix
```

```
      1  2
EU:2022  140.5  78.5
nonEU:2022  1.5   1.8
Europe:2022 142.0  80.3
```

The model matrix for columns can be seen as:

```
> t(hc2$hcCol$dataDummyHierarchy)
```

```
      old young All
[1,]  1     .   1
[2,]  .     1   1
```

Finally, the aggregate output values can be calculated as:

```
> hc2$hcRow$dataDummyHierarchy %*% hc2$hcRow$valueMatrix %*% t(hc2$hcCol$dataDummyHierarchy)
```

```
      old young All
EU:2022  140.5  78.5 219.0
nonEU:2022  1.5   1.8   3.3
Europe:2022 142.0  80.3 222.3
```

Taking into account that the dummy matrices are transposed model matrices, equation (2) is applied. With ordinary output, these values are organized into a data frame and we do not see what is going on internally.

```
> HierarchyCompute(data = cbind(d, year = 2022),
+   hierarchies = list(age = hi$age, geo = hi$geo, year = "rowFactor"),
+   colVar = "age", inputInOutput = c(TRUE, FALSE),
+   valueVar = "value")

   age   geo year value
1  old    EU 2022 140.5
2  old nonEU 2022   1.5
3  old Europe 2022 142.0
4 young    EU 2022  78.5
5 young nonEU 2022   1.8
6 young Europe 2022  80.3
7  All     EU 2022 219.0
8  All nonEU 2022   3.3
9  All Europe 2022 222.3
```

We obtain the same results without specifying `colVar`. One reason to use `colVar` is to do the computations more efficiently. Another reason may be that the internal matrices may be of interest. Note that variable combinations for output can be specified by parameters `rowSelect`, `colSelect` and `select`. Then `colVar` also matters.

5 Aggregation beyond summation

The most recent functions in the **SSBtools** package are about general aggregation beyond summation. With respect to future application packages, this is written in a different coding style (lower snake case). The function `model_aggregate` combines the model specification capabilities of `ModelMatrix` with aggregation with general functions. Below is an example:

```
> model_aggregate(d, formula = ~age + eu,
+   fun_vars = c(max = "value", median = "value", length = "geo"))

   age   eu value_max value_median geo_length
1 Total Total    120.3      15.90         6
2  old Total    120.3      20.20         3
3 young Total     66.9      11.60         3
4 Total   EU    120.3      43.55         4
5 Total nonEU     1.8       1.65         2
```

The underlying computations make use of a model matrix. By default (`pre_aggregate = TRUE`), the number of rows in the input data frame is reduced before `ModelMatrix` is called. In practice, this pre-aggregation step is done by running `aggregate` with `FUN = function(x){x}` and `simplify = FALSE`. The individual observations to be aggregated are then included as lists. In this example, the reduced input data is:

```
   age   eu      value      geo
1  old    EU 120.3, 20.2 Spain, Portugal
2 young    EU  66.9, 11.6 Spain, Portugal
3  old nonEU    1.5      Iceland
4 young nonEU    1.8      Iceland
```

The corresponding model matrix is:

```
      Total-Total old-Total young-Total Total-EU Total-nonEU
[1,]          1          1          .          1          .
[2,]          1          .          1          1          .
[3,]          1          1          .          .          1
[4,]          1          .          1          .          1
```

The function `model_aggregate` has many possibilities for specifying the aggregation. Functions of several data variables are possible. Output variable names can be passed as input, even for functions with multiple outputs. This is possible since `model_aggregate` calls `dummy_aggregate` (model matrix is input) which calls `aggregate_multiple_fun` which is an advanced wrapper to `aggregate`. The implementation uses indexes to rows, which made it possible to call `aggregate` only once. Since matrix multiplication is a fast way to do sum aggregation, this is also included as a possibility in `model_aggregate` (parameter `sum_vars`).

6 Comparisons with comments

To facilitate comparisons, we utilize hierarchical example data with 10^i rows obtained by crossing i dimensions. For each dimension there is a child variable with 10 categories and a parent variable with 3 categories (2, 3 and 5 categories aggregated). The child and parent variables are named alphabetically by small and capital letters, respectively. Below is an illustration of a `HierarchyCompute` call when $i = 4$, showing eight rows in both the input and output.

```
> d <- SSBtoolsData("power10to4")
> d[c(1:4, 3715:3716, 9999:10000), ]
```

	a	A	b	B	c	C	d	D
1	a1	A100	b1	B100	c1	C100	d1	D100
2	a2	A100	b1	B100	c1	C100	d1	D100
3	a3	A200	b1	B100	c1	C100	d1	D100
4	a4	A200	b1	B100	c1	C100	d1	D100
3715	a5	A200	b2	B100	c8	C300	d4	D200
3716	a6	A300	b2	B100	c8	C300	d4	D200
9999	a9	A300	b10	B300	c10	C300	d10	D300
10000	a10	A300	b10	B300	c10	C300	d10	D300

```
> hi <- FindHierarchies(d)
> d$y <- 1:nrow(d)
> output <- HierarchyCompute(d, hi, "y", inputInOutput = TRUE)
> output[c(1, 5473:5475, 37852:37854, 38416), ]
```

	a	b	c	d	y
1	a1	b1	c1	d1	1
5473	A300	B300	Total	d10	2382000
5474	Total	B300	Total	d10	4762750
5475	a1	Total	Total	d10	949600
37852	a9	b10	C200	Total	146970
37853	A100	b10	C200	Total	293490
37854	A200	b10	C200	Total	440460
38416	Total	Total	Total	Total	50005000

Here an integer y is added to the input data. Either way, the variable becomes numeric in the output. In the comparisons below, the input class is also numeric. We use `inputInOutput = TRUE`, which is the default in `ModelMatrix`. With 10 child categories, 3 parent categories, and the inclusion of a total code, this results in 14^i rows in the output.

Table 1 displays the CPU times for different function calls. The measurements were made by `system.time` (first element). The median values from five runs are shown. Table 2 displays the memory usage as peak RAM, measured using the [peakRAM](#) package (Quinn 2017). The median values from five runs are shown. These runs were conducted separately from the CPU time measurements. Table 2 also presents object sizes obtained using the `object.size` function. The source code is shown in Table 3. These comparisons were made with R version 4.2.2 running on Linux. The server had about 500 GiB of RAM, and it appeared that less than 10% of it were occupied by other users.

Since some functions are based on hierarchy input that may need to be generated from data, `FindHierarchies` are also included in the tables. Some functions produce aggregate results (`HierarchyCompute`, `FormulaSums` and `model_aggregate`), while others produce a model matrix. However, the time required for calculating sum aggregates through matrix multiplication is almost negligible. When $i = 6$, `crossprod(x, d$y)` takes four seconds. Here x refers to the sparse model matrix.

Generation of the model matrix from hierarchies by `ModelMatrix` is done via `HierarchyCompute`. The transposed model matrix is outputted before the ordinary results are produced. The main reason why `ModelMatrix` still takes longer, when $i \geq 4$, is a parameter setting that ensures that the column order is more similar to a usual model matrix (`reorder = TRUE`). Some time is also spent on matrix transposition. Even if `ModelMatrix` is run with `crossTable = FALSE`, which is the default, this information is still internally generated in order to create column names.

If the desired output is sum aggregates, there is no doubt that `HierarchyCompute` with `colVar` is the most efficient option. In this case, equation (2) is utilized, and instead of creating one large model matrix, two smaller matrices are generated. For example, when $i = 5$, dimensions a and b are used for columns, while c , d , and e are used for rows.

Table 1: User CPU time in seconds based on hierarchical input data with 10^i rows, resulting in 14^i output aggregates. Thus, the size of the model matrix is $10^i \times 14^i$. The source code is given in Table 3.

function call	$i = 2$	$i = 3$	$i = 4$	$i = 5$	$i = 6$
FindHierarchies	0.010	0.032	0.392	8	131
HierarchyCompute	0.017	0.027	0.188	5	147
HierarchyCompute with colVar	0.018	0.028	0.086	1	22
ModelMatrix from hierarchies	0.014	0.026	0.229	7	218
ModelMatrix from formula	0.016	0.083	1.294	60	2922
sparse.model.matrix	0.026	0.173	11.703	1597	
model.matrix	0.004	0.024	3.238		
FormulaSums	0.021	0.110	1.417	43	1041
median by model_aggregate	0.030	0.208	3.353	74	1403

Table 2: Peak RAM in MiB for the calculations described in Tables 1 and 3. For $i \geq 4$, the size of the output object is provided within parentheses, denoted in MiB as well.

function call	$i = 2$	$i = 3$	$i = 4$	$i = 5$	$i = 6$
FindHierarchies	0.9	1.9	12 (0.010)	57 (0.012)	610 (0.015)
HierarchyCompute	3.0	3.8	69 (1.5)	1074 (25)	30792 (402)
HierarchyCompute with colVar	1.2	2.1	20 (1.5)	201 (25)	1273 (402)
ModelMatrix from hierarchies	1.0	4.3	71 (12.2)	1079 (324)	32246 (9003)
ModelMatrix from formula	0.7	16.8	71 (12.2)	919 (324)	27809 (9003)
sparse.model.matrix	2.4	40.0	536 (13.1)	61654 (333)	
model.matrix	0.6	42.1	5864 (2935)		
FormulaSums	1.0	18.2	431 (3.1)	6884 (48)	1252 (689)
median by model_aggregate	0.9	15.2	230 (1.5)	2257 (25)	63435 (402)

Table 3: The source code for the calculations underlying Tables 1 and 2. With i defined, the code can be run directly from top to bottom. Be careful not to run FindHierarchies(d) with y included in d .

function call, etc.	source code
Input data	<code>d <- SSBtoolsData(paste0("power10to", i))</code>
FindHierarchies	<code>hi <- FindHierarchies(d)</code>
Adding numeric variable	<code>d\$y <- as.numeric(1:nrow(d))</code>
HierarchyCompute	<code>HierarchyCompute(d, hi, "y", inputInOutput = TRUE)</code>
HierarchyCompute with colVar	<code>HierarchyCompute(d, hi, "y", inputInOutput = TRUE, colVar = letters[1:(floor(i/2))])</code>
ModelMatrix from hierarchies	<code>ModelMatrix(d, hierarchies = hi)</code>
Formula	<code>f <- as.formula(strtrim("y ~ (a+A)*(b+B)*(c+C)*(d+D)*(e+E)*(f+F)", 3 + 6*i))</code>
ModelMatrix from formula	<code>ModelMatrix(d, formula = f)</code>
sparse.model.matrix	<code>ModelMatrix(d, formula = f, viaOrdinary = TRUE)</code>
model.matrix	<code>ModelMatrix(d, formula = f, viaOrdinary = TRUE, sparse = FALSE)</code>
FormulaSums	<code>FormulaSums(d, formula = f)</code>
median by model_aggregate	<code>model_aggregate(d, hierarchies = hi, fun_vars = c(median = "y"))</code>

Even though the functions handle hierarchical relationships, the formula used in the input must be written appropriately in order to achieve the desired output. To generate all combinations, the input formula to `ModelMatrix` at $i = 4$ would be $\sim(a+A)*(b+B)*(c+C)*(d+D)$. In Table 3, a response term is included in the formula, but it is only necessary for the `FormulaSums` function. The output from `FormulaSums` is a one-column matrix containing aggregates. However, the object size is larger compared to the data frame output of `HierarchyCompute`. This is because row names are a less efficient method of storing the information. It is interesting to note that `FormulaSums` uses more memory at $i = 5$ than $i = 6$. This is probably because the aggregate function uses a different method for large data.

The initial implementation of `ModelMatrix` utilized the functions `sparse.model.matrix` and `model.matrix`. This option still remains available by setting `viaOrdinary = TRUE`. Since these functions omit a factor level, the input variables are encoded as factors and an empty factor level is added. In this way, the desired matrix is achieved, here a $10^i \times 14^i$ matrix. With small data sets ($i \leq 3$), `model.matrix` is the fastest method, but with $i \geq 5$, there was not enough available memory. With $i = 5$ and $i = 6$, the size of the output object would be approximately 400 GiB and 55 TiB, respectively. The results show, perhaps surprisingly, that `sparse.model.matrix` consumes significant time and memory. It was not feasible to run it at $i = 6$. Profiling indicates that this is primarily due to interaction calculations. In contrast, in `ModelMatrix`, which only handles categorical variables, this process is simplified by combining multiple variables from the input before calling `fac2sparse`. The hierarchy interface to `ModelMatrix`, however, is significantly faster than the formula interface. This is because a few calls to the `KhatriRao` function are much faster than building the matrix with `cbind` based on numerous `fac2sparse` calls. The object size of the output matrix is slightly larger when using `sparse.model.matrix` compared to the regular `ModelMatrix`. This is primarily because the output from `sparse.model.matrix` includes row names as well.

Here, `model_aggregate` is used to calculate median values instead of the sums calculated by `HierarchyCompute`. Output is similar, but the rows are sorted differently. Internally, `ModelMatrix` is called, and the formula interface is also possible. Instead of matrix multiplication, the median function is now called 14^i times. This is more resource-intensive, but still feasible when $i = 6$.

7 Applications in other R packages

The first of the functions demonstrated above to be used in a specific purpose package was `FindDimLists`. Package `easySdcTable` (Langsrud 2022) provides a simplifying interface to the statistical disclosure control (SDC) package `sdcTable` (Meindl 2023). This is about protecting frequency tables by the suppression method. Automatic hierarchies by `FindDimLists` is an important part of the simplification.

The function `HierarchyCompute` was originally made for Statistics Norway's modernized calculations of `municipal accounts`, which involve complicated hierarchies. The original intention was to do the computations by an implementation of VTL (Airo et al. 2015), but this approach proved to be too inefficient and R was chosen instead. The calculations are implemented in the package `Kostr` (Lillegård et al. 2023) that is made for internal use in Statistics Norway. The two complicated hierarchies involved are illustrated on Figure 2. This figure as well as Figure 1 is made using package `igraph` (Csardi and Nepusz 2006). Plotting could be achieved by converting the hierarchies produced by `AutoHierarchies` into `igraph` objects using the `graph_from_data_frame` function.

The hierarchy interface part of `ModelMatrix` was created as a spin-off from `HierarchyCompute`. The first application was in the SDC package `SmallCountRounding` (Langsrud and Heldal 2022), which protects frequency tables data by the perturbation method described by Langsrud and Heldal (2018). The earliest versions of this package were limited to formula input and the package then contained the first version of `ModelMatrix`. Two other SDC packages also use `ModelMatrix`. The packages `SSBcellKey` (Lupp and Langsrud 2023) and `GaussSuppression` (Langsrud and Lupp 2023a) can be used to protect both frequency and magnitude tables by, respectively, perturbation (Thompson, Broadfoot, and Elazar 2013) and suppression (Hundepool et al. 2012).

All the SDC packages mentioned above are about protection of tabular data. When `ModelMatrix` is used, the model matrix itself is important to the algorithm. The model matrix is therefore necessary for more than aggregating the data by a matrix multiplication. In the package `SSBcellKey`, however, use of the model matrix is limited to running the `SSBtools` function `DummyApply` which is a precursor to the `dummy_aggregate` function mentioned above.

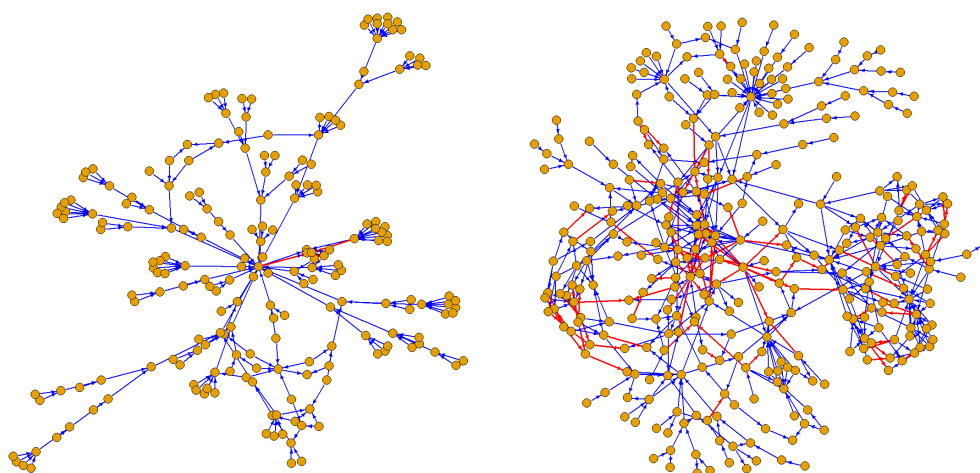


Figure 2: The function (left) and art (right) hierarchy involved in Statistics Norway's municipal accounts calculations. Red arrow means negative sign.

8 Conclusion

The **SSBtools** package provides functionality to handle hierarchical relationships and to generate corresponding model matrices. This is useful in contexts where multidimensional aggregation and related computations are to be done. This is applied by several R packages for statistical disclosure control, a field that is important for official statistics.

Acknowledgements

I would like to thank my colleagues Daniel Lupp and Johan Fosen at Statistics Norway, as well as an anonymous reviewer, for their valuable comments that led to improvements.

References

- Airo, Sami, Foteini Andrikopoulou, David Barraclough, Luigi Bellomarini, Marc Bouffard, Maurizio Capaccioli, Vincenzo Vecchio, et al. 2015. *Validation & Transformation Language, Part 2 Library of Operators, Version 1.0*. https://sdmx.org/wp-content/uploads/VTL1_2015_part2_operators_final.pdf.
- Bates, Douglas, Martin Maechler, and Mikael Jagan. 2023. *Matrix: Sparse and Dense Matrix Classes and Methods*. <https://CRAN.R-project.org/package=Matrix>.
- Csardi, Gabor, and Tamas Nepusz. 2006. "The Igraph Software Package for Complex Network Research." *InterJournal Complex Systems*: 1695. <https://igraph.org>.
- Hundepool, Anco, Josep Domingo-Ferrer, Luisa Franconi, Sarah Giessing, Eric Schulte Nordholt, Keith Spicer, and Peter-Paul de Wolf. 2012. *Statistical Disclosure Control*. John Wiley & Sons, Ltd. <https://doi.org/10.1002/9781118348239.ch1>.
- Langsrud, Øyvind. 2022. *easySdcTable: Easy Interface to the Statistical Disclosure Control Package 'sdcTable' Extended with Own Implementation of 'GaussSuppression'*. <https://CRAN.R-project.org/package=easySdcTable>.
- Langsrud, Øyvind, and Johan Heldal. 2018. "An Algorithm for Small Count Rounding of Tabular Data." *Privacy in statistical databases, Valencia, Spain*.
- . 2022. *SmallCountRounding: Small Count Rounding of Tabular Data*. <https://CRAN.R-project.org/package=SmallCountRounding>.
- Langsrud, Øyvind, and Daniel Lupp. 2023a. *GaussSuppression: Tabular Data Suppression Using Gaussian Elimination*. <https://CRAN.R-project.org/package=GaussSuppression>.
- . 2023b. *SSBtools: Statistics Norway's Miscellaneous Tools*. <https://CRAN.R-project.org/package=SSBtools>.
- Lillegård, Magnar, Anna-Karin Mevik, Johan Heldal, and Øyvind Langsrud. 2023. *Kostrat: Functions for Kostra*. <https://github.com/statisticsnorway/Kostrat>.

- Lupp, Daniel, and Øyvind Langsrud. 2023. *SSBcellKey: Cell-Key Method for Tabular Data*. <https://github.com/statisticsnorway/SSBcellKey>.
- Meindl, Bernhard. 2023. *sdcTable: Methods for Statistical Disclosure Control in Tabular Data*. <https://CRAN.R-project.org/package=sdcTable>.
- Quinn, Thomas. 2017. *peakRAM: Monitor the Total and Peak RAM Used by an Expression or Function*. <https://CRAN.R-project.org/package=peakRAM>.
- Thompson, G., S. Broadfoot, and D. Elazar. 2013. "Methodology for the automatic confidentialisation of statistical outputs from remote servers at the Australian Bureau of Statistics." Joint UNECE/Eurostat Work Session on Statistical Data.
- Wolf, Peter-Paul de, Anco Hundepool, Sarah Giessing, Juan-José Salazar, and Jordi Castro. 2014. "tau-ARGUS user's manual, version 4.1." Statistics Netherlands. <https://github.com/sdcTools/tauargus>.

Øyvind Langsrud
Statistics Norway
P.O. Box 8131 Dep.,
0033 Oslo, Norway
<https://github.com/olangsrud>
ORCID: 0000-0002-1380-4396
Oyvind.Langsrud@ssb.no