Array operations in the gRbase package

Søren Højsgaard

gRbase version 2.0.2 as of 2024-05-29

Contents


```
## Warning:  package 'gRbase' was built under R version 4.4.0
## Error:  package or namespace load failed for 'gRbase' in dyn.load(file, DLLpath = DLLpath,
...):
```

```
##  unable to load shared object '/home/sorenh/R/x86_64-pc-linux-gnu-library/4.3/gRbase/libs/gRbase.so':
##   libRblas.so:  cannot open shared object file:  No such file or directory
```

1 Introduction

This note describes some operations on arrays in R. These operations have been implemented to facilitate implementation of graphical models and Bayesian networks in R.

2 Arrays/tables in R

The documentation of R states the following about arrays:

> *An array in R can have one, two or more dimensions. It is simply a vector which is stored with additional attributes giving the dimensions (attribute "dim") and optionally names for those dimensions (attribute "dimnames"). A two-dimensional array is the same thing as a matrix. One-dimensional arrays often look like vectors, but may be handled differently by some functions.*

2.1 Cross classified data - contingency tables

Arrays appear for example in connection with cross classified data. The array `hec` below is an excerpt of the `HairEyeColor` array in R:

```
hec <- c(32, 53, 11, 50, 10, 25, 36, 66, 9, 34, 5, 29)
dim(hec) <- c(2, 3, 2)
dimnames(hec) <- list(Hair = c("Black", "Brown"),
                      Eye = c("Brown", "Blue", "Hazel"),
                      Sex = c("Male", "Female"))
hec
## , , Sex = Male
##
##        Eye
## Hair    Brown Blue Hazel
##   Black    32   11    10
##   Brown    53   50    25
##
## , , Sex = Female
##
##        Eye
## Hair    Brown Blue Hazel
##   Black    36    9     5
##   Brown    66   34    29
```

Above, `hec` is an array because it has a `dim` attribute. Moreover, `hec` also has a `dimnames` attribute naming the levels of each dimension. Notice that each dimension is given a name.

Printing arrays can take up a lot of space. Alternative views on an array can be obtained with `ftable()` or by converting the array to a dataframe with `as.data.frame.table()`. We shall do so in the following.

```r
##flat <- function(x) {ftable(x, row.vars=1)}
flat <- function(x, n=4) {as.data.frame.table(x) %>% head(n)}
hec %>% flat

## Error in hec %>% flat:  could not find function "%>%"
```

An array with named dimensions is in this package called a *named array*. The functionality described below relies heavily on arrays having named dimensions. A check for an object being a named array is provided by `is.named.array()`[gRbase]

```r
is.named.array(hec)

## Error in is.named.array(hec):  could not find function "is.named.array"
```

2.2 Defining arrays

Another way is to use `tabNew()`[gRbase] from **gRbase**. This function is flexible wrt the input; for example:

```r
dn <- list(Hair=c("Black", "Brown"), Eye=~Brown:Blue:Hazel, Sex=~Male:Female)
counts <- c(32, 53, 11, 50, 10, 25, 36, 66, 9, 34, 5, 29)
z3 <- tabNew(~Hair:Eye:Sex, levels=dn, value=counts)

## Error in tabNew(~Hair:Eye:Sex, levels = dn, value = counts):  could not find function "tabNew"

z4 <- tabNew(c("Hair", "Eye", "Sex"), levels=dn, values=counts)

## Error in tabNew(c("Hair", "Eye", "Sex"), levels = dn, values = counts):  could not find
function "tabNew"
```

Notice that the levels list (`dn` above) when used in `tabNew()`[gRbase] is allowed to contain superfluous elements. Default `dimnames` are generated with

```r
z5 <- tabNew(~Hair:Eye:Sex, levels=c(2, 3, 2), values = counts)

## Error in tabNew(~Hair:Eye:Sex, levels = c(2, 3, 2), values = counts):  could not find function
"tabNew"

dimnames(z5) %>% str

## Error in dimnames(z5) %>% str:  could not find function "%>%"
```

Using `tabNew`[gRbase], arrays can be normalized to sum to one in two ways: 1) Normalization can be over the first variable for *each* configuration of all other variables and 2) over all configurations. For example:

```r
z6 <- tabNew(~Hair:Eye:Sex, levels=c(2, 3, 2), values=counts, normalize="first")

## Error in tabNew(~Hair:Eye:Sex, levels = c(2, 3, 2), values = counts, normalize = "first"):
could not find function "tabNew"

z6 %>% flat

## Error in z6 %>% flat:  could not find function "%>%"
```

3 Operations on arrays

In the following we shall denote the dimnames (or variables) of the array `hec` by H, E and S and we let (h, e, s) denote a configuration of these variables. The contingency table above shall be denoted by T_{HES} and we shall refer to the (h, e, s)-entry of T_{HES} as $T_{HES}(h, e, s)$.

3.1 Normalizing an array

Normalize an array with `tabNormalize()`[gRbase] Entries of an array can be normalized to sum to one in two ways: 1) Normalization can be over the first variable for *each* configuration of all other variables and 2) over all configurations. For example:

```
tabNormalize(z5, "first") %>% flat

## Error in tabNormalize(z5, "first") %>% flat:  could not find function "%>%"
```

3.2 Subsetting an array – slicing

We can subset arrays (this will also be called “slicing”) in different ways. Notice that the result is not necessarily an array. Slicing can be done using standard `R` code or using `tabSlice`[gRbase]. The virtue of `tabSlice`[gRbase] comes from the flexibility when specifying the slice:

The following leads from the original $2 \times 3 \times 2$ array to a 2×2 array by cutting away the `Sex=Male` and `Eye=Brown` slice of the array:

```
tabSlice(hec, slice=list(Eye=c("Blue", "Hazel"), Sex="Female"))

## Error in tabSlice(hec, slice = list(Eye = c("Blue", "Hazel"), Sex = "Female")):  could not
find function "tabSlice"

## Notice: levels can be written as numerics
## tabSlice(hec, slice=list(Eye=2:3, Sex="Female"))
```

We may also regard the result above as a $2 \times 2 \times 1$ array:

```
tabSlice(hec, slice=list(Eye=c("Blue", "Hazel"), Sex="Female"), drop=FALSE)

## Error in tabSlice(hec, slice = list(Eye = c("Blue", "Hazel"), Sex = "Female"), :  could
not find function "tabSlice"
```

If slicing leads to a one dimensional array, the output will by default not be an array but a vector (without a dim attribute). However, the result can be forced to be a 1-dimensional array:

```
## A vector:
t1 <- tabSlice(hec, slice=list(Hair=1, Sex="Female")); t1

## Error in tabSlice(hec, slice = list(Hair = 1, Sex = "Female")):  could not find function
"tabSlice"

## Error in eval(expr, envir, enclos):  object 't1' not found

## A 1-dimensional array:
t2 <- tabSlice(hec, slice=list(Hair=1, Sex="Female"), as.array=TRUE); t2
```

```
## Error in tabSlice(hec, slice = list(Hair = 1, Sex = "Female"), as.array = TRUE): could not
find function "tabSlice"

## Error in eval(expr, envir, enclos):  object 't2' not found

## A higher dimensional array (in which some dimensions only have one level)
t3 <- tabSlice(hec, slice=list(Hair=1, Sex="Female"), drop=FALSE); t3

## Error in tabSlice(hec, slice = list(Hair = 1, Sex = "Female"), drop = FALSE): could not
find function "tabSlice"

## Error in eval(expr, envir, enclos):  object 't3' not found
```

The difference between the last two forms can be clarified:

```
t2 %>% flat

## Error in t2 %>% flat:  could not find function "%>%"

t3 %>% flat

## Error in t3 %>% flat:  could not find function "%>%"
```

3.3 Collapsing and inflating arrays

Collapsing: The HE–marginal array T_{HE} of T_{HES} is the array with values

$$T_{HE}(h,e) = \sum_s T_{HES}(h,e,s)$$

Inflating: The "opposite" operation is to extend an array. For example, we can extend T_{HE} to have a third dimension, e.g. `Sex`. That is

$$\tilde{T}_{SHE}(s,h,e) = T_{HE}(h,e)$$

so $\tilde{T}_{SHE}(s,h,e)$ is constant as a function of s.

With `gRbase` we can collapse arrays with[1]:

```
he <- tabMarg(hec, c("Hair", "Eye"))

## Error in tabMarg(hec, c("Hair", "Eye")):  could not find function "tabMarg"

he

## Error in eval(expr, envir, enclos):  object 'he' not found
```

```
## Alternatives
tabMarg(hec, ~Hair:Eye)

## Error in tabMarg(hec, ~Hair:Eye):  could not find function "tabMarg"

tabMarg(hec, c(1, 2))
```

[1] FIXME: Should allow for abbreviations in formula and character vector specifications.

```
## Error in tabMarg(hec, c(1, 2)):  could not find function "tabMarg"

hec %a_% ~Hair:Eye

## Error in hec %a_% ~Hair:Eye:  could not find function "%a_%"
```

Notice that collapsing is a projection in the sense that applying the operation again does not change anything:

```
he1 <- tabMarg(hec, c("Hair", "Eye"))

## Error in tabMarg(hec, c("Hair", "Eye")):  could not find function "tabMarg"

he2 <- tabMarg(he1, c("Hair", "Eye"))

## Error in tabMarg(he1, c("Hair", "Eye")):  could not find function "tabMarg"

tabEqual(he1, he2)

## Error in tabEqual(he1, he2):  could not find function "tabEqual"
```

Expand an array by adding additional dimensions with `tabExpand()`[gRbase]:

```
extra.dim <- list(Sex=c("Male", "Female"))
tabExpand(he, extra.dim)

## Error in tabExpand(he, extra.dim):  could not find function "tabExpand"
```

```
## Alternatives
he %a^% extra.dim

## Error in he %a^% extra.dim:  could not find function "%a^%"
```

Notice that expanding and collapsing brings us back to where we started:

```
(he %a^% extra.dim) %a_% c("Hair", "Eye")

## Error in (he %a^% extra.dim) %a_% c("Hair", "Eye"):  could not find function "%a_%"
```

3.4 Permuting an array

A reorganization of the table can be made with `tabPerm`[gRbase] (similar to `aperm()`), but `tabPerm`[gRbase] allows for a formula and for variable abbreviation:

```
tabPerm(hec, ~Eye:Sex:Hair) %>% flat

## Error in tabPerm(hec, ~Eye:Sex:Hair) %>% flat:  could not find function "%>%"
```

Alternative forms (the first two also works for `aperm`):

```
tabPerm(hec, c("Eye", "Sex", "Hair"))

## Error in tabPerm(hec, c("Eye", "Sex", "Hair")):  could not find function "tabPerm"

tabPerm(hec, c(2, 3, 1))

## Error in tabPerm(hec, c(2, 3, 1)):  could not find function "tabPerm"

tabPerm(hec, ~Ey:Se:Ha)

## Error in tabPerm(hec, ~Ey:Se:Ha):  could not find function "tabPerm"

tabPerm(hec, c("Ey", "Se", "Ha"))

## Error in tabPerm(hec, c("Ey", "Se", "Ha")):  could not find function "tabPerm"
```

3.5 Equality

Two arrays are defined to be identical 1) if they have the same dimnames and 2) if, possibly after a permutation, all values are identical (up to a small numerical difference):

```
hec2 <- tabPerm(hec, 3:1)

## Error in tabPerm(hec, 3:1):  could not find function "tabPerm"

tabEqual(hec, hec2)

## Error in tabEqual(hec, hec2):  could not find function "tabEqual"
```

```
## Alternative
hec %a==% hec2

## Error in hec %a==% hec2:  could not find function "%a==%"
```

3.6 Aligning

We can align one array according to the ordering of another:

```
hec2 <- tabPerm(hec, 3:1)

## Error in tabPerm(hec, 3:1):  could not find function "tabPerm"

tabAlign(hec2, hec)

## Error in tabAlign(hec2, hec):  could not find function "tabAlign"
```

```
## Alternative:
tabAlign(hec2, dimnames(hec))

## Error in tabAlign(hec2, dimnames(hec)):  could not find function "tabAlign"
```

3.7 Multiplication, addition etc: $+$, $-$, $*$, $/$

The product of two arrays T_{HE} and T_{HS} is defined to be the array $\tilde{T}_{HES}$ with entries

$$\tilde{T}_{HES}(h, e, s) = T_{HE}(h, e) + T_{HS}(h, s)$$

The sum, difference and quotient is defined similarly: This is done with `tabProd()`[gRbase], `tabAdd()`[gRbase], `tabDiff()`[gRbase] and `tabDiv()`[gRbase]:

```
hs <- tabMarg(hec, ~Hair:Eye)

## Error in tabMarg(hec, ~Hair:Eye):  could not find function "tabMarg"

tabMult(he, hs)

## Error in tabMult(he, hs):  could not find function "tabMult"
```

Available operations:

```
tabAdd(he, hs)

## Error in tabAdd(he, hs):  could not find function "tabAdd"

tabSubt(he, hs)

## Error in tabSubt(he, hs):  could not find function "tabSubt"

tabMult(he, hs)

## Error in tabMult(he, hs):  could not find function "tabMult"

tabDiv(he, hs)

## Error in tabDiv(he, hs):  could not find function "tabDiv"

tabDiv0(he, hs) ## Convention 0/0 = 0

## Error in tabDiv0(he, hs):  could not find function "tabDiv0"
```

Shortcuts:

```
## Alternative
he %a+% hs

## Error in he %a+% hs:  could not find function "%a+%"

he %a-% hs

## Error in he %a-% hs:  could not find function "%a-%"

he %a*% hs

## Error in he %a*% hs:  could not find function "%a*%"
```

```
he %a/% hs

## Error in he %a/% hs:  could not find function "%a/%"

he %a/0% hs ## Convention 0/0 = 0

## Error in he %a/0% hs:  could not find function "%a/0%"
```

Multiplication and addition of (a list of) multiple arrays is accomplished with `tabProd()`[gRbase] and `tabSum()`[gRbase] (much like `prod()`[gRbase] and `sum()`[gRbase]):

```
es <- tabMarg(hec, ~Eye:Sex)

## Error in tabMarg(hec, ~Eye:Sex):  could not find function "tabMarg"

tabSum(he, hs, es)

## Error in tabSum(he, hs, es):  could not find function "tabSum"

## tabSum(list(he, hs, es))
```

3.8 An array as a probability density

If an array consists of non–negative numbers then it may be regarded as an (unnormalized) discrete multivariate density. With this view, the following examples should be self explanatory:

```
tabDist(hec, marg=~Hair:Eye)

## Error in tabDist(hec, marg = ~Hair:Eye):  could not find function "tabDist"

tabDist(hec, cond=~Sex)

## Error in tabDist(hec, cond = ~Sex):  could not find function "tabDist"

tabDist(hec, marg=~Hair, cond=~Sex)

## Error in tabDist(hec, marg = ~Hair, cond = ~Sex):  could not find function "tabDist"
```

3.9 Miscellaneous

Multiply values in a slice by some number and all other values by another number:

```
tabSliceMult(es, list(Sex="Female"), val=10, comp=0)

## Error in tabSliceMult(es, list(Sex = "Female"), val = 10, comp = 0):  could not find function
"tabSliceMult"
```

4 Examples

4.1 A Bayesian network

A classical example of a Bayesian network is the "sprinkler example", see e.g. `http://en.wikipedia.org/wiki/Bayesian_network`:

> *Suppose that there are two events which could cause grass to be wet: either the sprinkler is on or it is raining. Also, suppose that the rain has a direct effect on the use of the sprinkler (namely that when it rains, the sprinkler is usually not turned on). Then the situation can be modeled with a Bayesian network.*

We specify conditional probabilities $p(r)$, $p(s|r)$ and $p(w|s,r)$ as follows (notice that the vertical conditioning bar (|) is replaced by the horizontal underscore:

```
yn <- c("y","n")
lev <- list(rain=yn, sprinkler=yn, wet=yn)
r <- tabNew(~rain, levels=lev, values=c(.2, .8))

## Error in tabNew(~rain, levels = lev, values = c(0.2, 0.8)):  could not find function "tabNew"

s_r <- tabNew(~sprinkler:rain, levels = lev, values = c(.01, .99, .4, .6))

## Error in tabNew(~sprinkler:rain, levels = lev, values = c(0.01, 0.99, :  could not find
function "tabNew"

w_sr <- tabNew( ~wet:sprinkler:rain, levels=lev,
             values=c(.99, .01, .8, .2, .9, .1, 0, 1))

## Error in tabNew(~wet:sprinkler:rain, levels = lev, values = c(0.99, 0.01, :  could not find
function "tabNew"

r

## Error in eval(expr, envir, enclos):  object 'r' not found

s_r  %>% flat

## Error in s_r %>% flat:  could not find function "%>%"

w_sr %>% flat

## Error in w_sr %>% flat:  could not find function "%>%"
```

The joint distribution $p(r, s, w) = p(r)p(s|r)p(w|s, r)$ can be obtained with `tabProd()`[gRbase]: ways:

```
joint <- tabProd(r, s_r, w_sr); joint %>% flat

## Error in tabProd(r, s_r, w_sr):  could not find function "tabProd"
## Error in joint %>% flat:  could not find function "%>%"
```

What is the probability that it rains given that the grass is wet? We find $p(r, w) = \sum_s p(r, s, w)$ and then $p(r|w) = p(r, w)/p(w)$. Can be done in various ways: with `tabDist()`[gRbase]

```r
tabDist(joint, marg=~rain, cond=~wet)

## Error in tabDist(joint, marg = ~rain, cond = ~wet):  could not find function "tabDist"
```

```r
## Alternative:
rw <- tabMarg(joint, ~rain + wet)

## Error in tabMarg(joint, ~rain + wet):  could not find function "tabMarg"

tabDiv(rw, tabMarg(rw, ~wet))

## Error in tabDiv(rw, tabMarg(rw, ~wet)):  could not find function "tabDiv"

## or
rw %a/% (rw %a_% ~wet)

## Error in rw %a/% (rw %a_% ~wet):  could not find function "%a/%"
```

```r
## Alternative:
x <- tabSliceMult(rw, slice=list(wet="y")); x

## Error in tabSliceMult(rw, slice = list(wet = "y")):  could not find function "tabSliceMult"
## Error in eval(expr, envir, enclos):  object 'x' not found

tabDist(x, marg=~rain)

## Error in tabDist(x, marg = ~rain):  could not find function "tabDist"
```

4.2 Iterative Proportional Scaling (IPS)

We consider the 3–way `lizard` data from `gRbase`:

```r
data(lizard, package="gRbase")
lizard %>% flat

## Error in lizard %>% flat:  could not find function "%>%"
```

Consider the two factor log–linear model for the `lizard` data. Under the model the expected counts have the form

$$\log m(d,h,s) = a_1(d,h) + a_2(d,s) + a_3(h,s)$$

If we let $n(d,h,s)$ denote the observed counts, the likelihood equations are: Find $m(d,h,s)$ such that

$$m(d,h) = n(d,h), \quad m(d,s) = n(d,s), \quad m(h,s) = n(h,s)$$

where $m(d,h) = \sum_s m(d,h.s)$ etc. The updates are as follows: For the first term we have

$$m(d,h,s) \leftarrow m(d,h,s)\frac{n(d,h)}{m(d,h)}$$

After iterating the updates will not change and we will have equality: $m(d,h,s) = m(d,h,s)\frac{n(d,h)}{m(d,h)}$ and summing over s shows that the equation $m(d,h) = n(d,h)$ is satisfied.

A rudimentary implementation of iterative proportional scaling for log–linear models is straight forward:

```
myips <- function(indata, glist){
    fit   <- indata
    fit[] <-  1
    ## List of sufficient marginal tables
    md    <- lapply(glist, function(g) tabMarg(indata, g))

    for (i in 1:4){
        for (j in seq_along(glist)){
            mf  <- tabMarg(fit, glist[[j]])
            # adj <- tabDiv( md[[ j ]], mf)
            # fit <- tabMult( fit, adj )
            ## or
            adj <- md[[ j ]] %a/% mf
            fit <- fit %a*% adj
        }
    }
    pearson <- sum((fit - indata)^2 / fit)
    list(pearson=pearson, fit=fit)
}

glist <- list(c("species", "diam"),c("species", "height"),c("diam", "height"))

fm1 <- myips(lizard, glist)

## Error in tabMarg(indata, g):  could not find function "tabMarg"

fm1$pearson

## Error in eval(expr, envir, enclos):  object 'fm1' not found

fm1$fit %>% flat

## Error in fm1$fit %>% flat:  could not find function "%>%"

fm2 <- loglin(lizard, glist, fit=T)
## 4 iterations: deviation 0.009619

fm2$pearson
## [1] 0.1506

fm2$fit %>% flat

## Error in fm2$fit %>% flat:  could not find function "%>%"
```

5 Some low level functions

For e.g. a $2 \times 3 \times 2$ array, the entries are such that the first variable varies fastest so the ordering of the cells are $(1,1,1)$, $(2,1,1)$, $(1,2,1)$, $(2,2,1)$,$(1,3,1)$ and so on. To find the value of such a cell, say, (j,k,l) in the array (which is really just a vector), the cell is mapped into an entry of a vector.

For example, cell $(2,3,1)$ (`Hair=Brown`, `Eye=Hazel`, `Sex=Male`) must be mapped to entry 4 in

```
hec
## , , Sex = Male
##
##        Eye
## Hair    Brown Blue Hazel
##   Black    32   11    10
##   Brown    53   50    25
##
## , , Sex = Female
##
##        Eye
## Hair    Brown Blue Hazel
##   Black    36    9     5
##   Brown    66   34    29

c(hec)
##  [1] 32 53 11 50 10 25 36 66  9 34  5 29
```

For illustration we do:

```
cell2name <- function(cell, dimnames){
    unlist(lapply(1:length(cell), function(m) dimnames[[m]][cell[m]]))
}
cell2name(c(2,3,1), dimnames(hec))
## [1] "Brown" "Hazel" "Male"
```

5.1 `cell2entry()`, `entry2cell()` and `next_cell()`

The map from a cell to the corresponding entry is provided by `cell2entry()`[gRbase]. The reverse operation, going from an entry to a cell (which is much less needed) is provided by `entry2cell()`[gRbase].

```
cell2entry(c(2,3,1), dim=c(2, 3, 2))

## Error in cell2entry(c(2, 3, 1), dim = c(2, 3, 2)):  could not find function "cell2entry"

entry2cell(6, dim=c(2, 3, 2))

## Error in entry2cell(6, dim = c(2, 3, 2)):  could not find function "entry2cell"
```

Given a cell, say $i=(2,3,1)$ in a $2\times 3\times 2$ array we often want to find the next cell in the table following the convention that the first factor varies fastest, that is $(1,1,2)$. This is provided by `next_cell()`[gRbase].

```
next_cell(c(2,3,1), dim=c(2, 3, 2))

## Error in next_cell(c(2, 3, 1), dim = c(2, 3, 2)):  could not find function "next_cell"
```

5.2 `next_cell_slice()` and `slice2entry()`

Given that we look at cells for which for which the index in dimension 2 is at level 3 (that is `Eye=Hazel`), i.e. cells of the form $(j, 3, l)$. Given such a cell, what is then the next cell that also satisfies this constraint. This is provided by `next_cell_slice()`[gRbase].[2]

```
next_cell_slice(c(1,3,1), slice_marg=2, dim=c( 2, 3, 2 ))

## Error in next_cell_slice(c(1, 3, 1), slice_marg = 2, dim = c(2, 3, 2)):  could not find
function "next_cell_slice"

next_cell_slice(c(2,3,1), slice_marg=2, dim=c( 2, 3, 2 ))

## Error in next_cell_slice(c(2, 3, 1), slice_marg = 2, dim = c(2, 3, 2)):  could not find
function "next_cell_slice"
```

Given that in dimension 2 we look at level 3. We want to find entries for the cells of the form $(j, 3, l)$.[3]

```
slice2entry(slice_cell=3, slice_marg=2, dim=c( 2, 3, 2 ))

## Error in slice2entry(slice_cell = 3, slice_marg = 2, dim = c(2, 3, 2)):  could not find
function "slice2entry"
```

To verify that we indeed get the right cells:

```
r <- slice2entry(slice_cell=3, slice_marg=2, dim=c( 2, 3, 2 ))

## Error in slice2entry(slice_cell = 3, slice_marg = 2, dim = c(2, 3, 2)):  could not find
function "slice2entry"

lapply(lapply(r, entry2cell, c( 2, 3, 2 )),
       cell2name, dimnames(hec))

## Error in eval(expr, envir, enclos):  object 'entry2cell' not found
```

5.3 `fact_grid()` – Factorial grid

Using the operations above we can obtain the combinations of the factors as a matrix:

```
head( fact_grid( c(2, 3, 2) ), 6 )

## Error in fact_grid(c(2, 3, 2)):  could not find function "fact_grid"
```

A similar dataframe can also be obtained with the standard `R` function `expand.grid` (but `factGrid` is faster)

```
head( expand.grid(list(1:2, 1:3, 1:2)), 6 )
```

[2]FIXME: sliceset should be called margin.
[3]FIXME:slicecell and sliceset should be renamed

```
##   Var1 Var2 Var3
## 1    1    1    1
## 2    2    1    1
## 3    1    2    1
## 4    2    2    1
## 5    1    3    1
## 6    2    3    1
```

A More about slicing

Slicing using standard R code can be done as follows:

```
hec[, 2:3, ]  %>% flat  ## A 2 x 2 x 2 array

## Error in hec[, 2:3, ] %>% flat:  could not find function "%>%"

hec[1, , 1]             ## A vector
## Brown  Blue Hazel
##    32    11    10

hec[1, , 1, drop=FALSE] ## A 1 x 3 x 1 array
## , , Sex = Male
##
##        Eye
## Hair    Brown Blue Hazel
##   Black    32   11    10
```

Programmatically we can do the above as

```
do.call("[", c(list(hec), list(TRUE, 2:3, TRUE)))  %>% flat

## Error in do.call("[", c(list(hec), list(TRUE, 2:3, TRUE))) %>% flat:  could not find function
"%>%"

do.call("[", c(list(hec), list(1, TRUE, 1)))
do.call("[", c(list(hec), list(1, TRUE, 1), drop=FALSE))
```

gRbase provides two alterntives for each of these three cases above:

```
tabSlicePrim(hec, slice=list(TRUE, 2:3, TRUE))  %>% flat

## Error in tabSlicePrim(hec, slice = list(TRUE, 2:3, TRUE)) %>% flat:  could not find function
"%>%"

tabSlice(hec, slice=list(c(2, 3)), margin=2) %>% flat

## Error in tabSlice(hec, slice = list(c(2, 3)), margin = 2) %>% flat:  could not find function
"%>%"

tabSlicePrim(hec, slice=list(1, TRUE, 1))

## Error in tabSlicePrim(hec, slice = list(1, TRUE, 1)):  could not find function "tabSlicePrim"
```

```r
tabSlice(hec, slice=list(1, 1), margin=c(1, 3))

## Error in tabSlice(hec, slice = list(1, 1), margin = c(1, 3)):  could not find function "tabSlice"

tabSlicePrim(hec, slice=list(1, TRUE, 1), drop=FALSE)

## Error in tabSlicePrim(hec, slice = list(1, TRUE, 1), drop = FALSE): could not find function
"tabSlicePrim"

tabSlice(hec, slice=list(1, 1), margin=c(1, 3), drop=FALSE)

## Error in tabSlice(hec, slice = list(1, 1), margin = c(1, 3), drop = FALSE): could not find
function "tabSlice"
```