The smint package

user's guide

2022-09-17

smint version 0.4.3

Yves Deville and Yann Richet

Yves Deville Statistical consultant Alpestat deville.yves@alpestat.com

Yann Richet Technical Advisor IRSN yann.richet@irsn.fr

Contents

The smint package: outlook

Goals

The **smint** package has been initiated and financed by the french institute IRSN[1]. The main goal is to provide *fast* methods of interpolation for typical dimensions between 3 and 7, as required for instance in the study of nuclear cross-sections. The two classical contexts of interpolation, namely *gridded data* and *scattered data*, are of interest.

We assume to be given n distinct vectors $\mathbf{x}_i$ called *nodes* in the d-dimensional space $\mathbb{R}^d$ and n real values f_i. The goal is to find a function f defined on a domain containing the nodes and such that the n interpolation conditions $f(\mathbf{x}_i) = f_i$ hold for $i = 1, 2, \ldots, n$. The function must be smooth, and at least continuous. It will be obtained in a form allowing the evaluation of $f(\mathbf{x}_j^{\text{new}})$ for n^{new} arbitrary new points in $\mathbb{R}^d$.

A boldface notation will be used for vectors and matrices as in $\mathbf{x}_i = [x_{i1}, x_{i2}, \ldots, x_{id}]^\top$ for the i-th node. The notations $x_1, x_2, \ldots, x_d$ will be used to denote the coordinates or variables matching the d dimensions. In the classical contexts where $d \leqslant 3$, the dimensions can for the sake of clarity be named x, y, z in place of x_1, x_2 and x_3 and the function values can be denoted by f_i. In the general context, the prescribed function values at the nodes will sometimes be denoted as y_i rather than f_i.

Scattered data

For the *scattered data* context, the data to be interpolated are likely to be given as a data frame or matrix `X` with a numeric vector of response `f` or `y`.

Grid data

A d-dimensional grid is a finite set in the d-dimensional space which is the tensor product of d finite sets, one for each dimension. The n_j elements for the dimension j may be called *levels* of the variable x_j and can be assumed to be given in increasing order

$$x^\star_{j,1} < x^\star_{j,2} < \cdots < x^\star_{j,n_j} \qquad j = 1, 2, \ldots, d.$$

The total number of nodes is $n = n_1 \times n_2 \times \cdots \times n_d$.

The levels are conveniently stored in R as a list of d numeric vectors. Often the grid range will be the hyper-cube of interest, so nodes having one of their levels equal to the minimum or maximal level are boundary points.

When working with grid data, a particular ordering of the nodes must be chosen so that each of the n elements in the response vector can be related to the corresponding node $\mathbf{x}_i$.

[1] Institut de Radioprotection et de Sûreté Nucléaire

Multi-response

In some cases it will be needed to interpolate several functions rather than one, still using the same set of nodes $\mathbf{x}_i$ and the same set of new evaluation points x_j^{new}. We call this context *multi-response* interpolation, since multivariate interpolation is ambiguous.

If m response functions are of interest, the n prescribed function values can be seen as forming a $n \times m$ matrix $\mathbf{F}$.

Chapter 1

The `Grid` class

1.1 Motivation: grids as data frames

The popular `expand.grid` function from the **base** package provides a representation of a grid as a data frame object.

```
df <- expand.grid(x = c(0.0, 0.2, 1.0), y = c(1.0, 2.5, 3.0), z = c(0.2, 0.4))
nrow(df)

## [1] 18

head(df)

##     x   y   z
## 1 0.0 1.0 0.2
## 2 0.2 1.0 0.2
## 3 1.0 1.0 0.2
## 4 0.0 2.5 0.2
## 5 0.2 2.5 0.2
## 6 1.0 2.5 0.2

class(df)

## [1] "data.frame"
```

Note the rule: *first index varies faster*, which will also be retained in **smint**. We could as well have used a single *list* formal argument

```
df2 <- expand.grid(list("x" = c(0.0, 0.2, 1.0), "y" = c(1.0, 2.5, 3.0),
                        "z" = c(0.2, 0.4)))
identical(df, df2)

## [1] TRUE
```

This second form is convenient to deal with grids in an arbitrary dimension d.

The grid described by `df` could be shown on a three dimensional plot using the package **scatterplot3d** or **rgl**, see left panel of figure 1.1. But an useful diagnostic is straightforwardly given by the `plot` method, which provides the *pairs plot* shown on the right of figure 1.1.

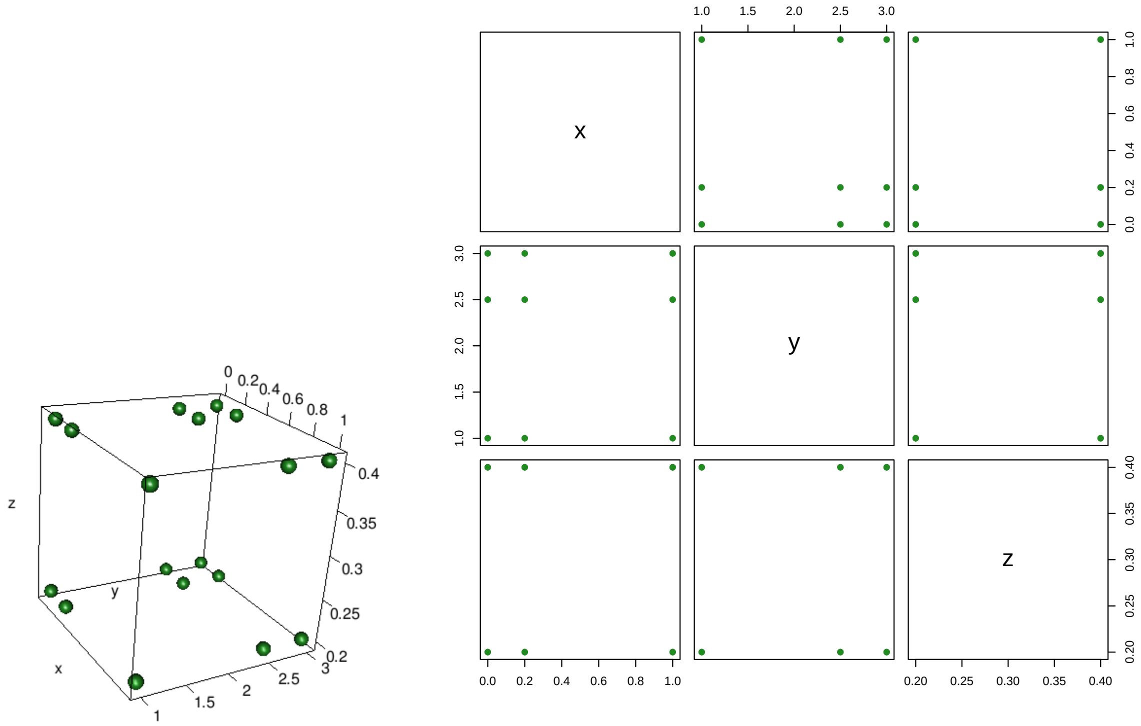

Figure 1.1: Left: three-dimensional representation of the grid using **rgl**. Right: using the `plot` method for data frames.

Each point shown in a pair panel corresponds to *several* grid points having the same two-dimensional projection. This problem gets more crucial with grids in higher dimension, since many points can then be collapsed into one. Using semi-transparent (translucent) colours can be of some help, see later.

Recall that even when their columns are numeric, data frames are very different from matrices. They are *frames* in which columns can be used in expression, like objects in an environment. This is usually achieved by using the methods `with`, `within`

```
f <- with(df, x^3 + 2 * y + z^2)
```

Yet a function can easily be applied to each row of the data frame in a matrix fashion, for instance $f(x, y, z) := x^3 + 2y + z^2$.

```
my3dFun <- function(x) x[1]^3 + 2 * x[2] + x[3]^2
f2 <- apply(df, MARGIN = 1, FUN = my3dFun)
max(abs(f - f2))

## [1] 0
```

Note that the order of the columns is essential here, while it was immaterial in the computation of `f` above. Since each grid dimension is mapped to a column, it is quite easy to have a permutation of the dimensions.

```
dfP <- df[ , c(2, 3, 1)]
head(dfP)

##     y   z   x
## 1 1.0 0.2 0.0
## 2 1.0 0.2 0.2
## 3 1.0 0.2 1.0
## 4 2.5 0.2 0.0
## 5 2.5 0.2 0.2
## 6 2.5 0.2 1.0
```

The order of the rows also matters. In some contexts, the nodes have a particular ordering, e.g. because they are associated to successive experiments. Then it is important to keep a numbering of the nodes attached with the grid, as is naturally done in the data frame representation.

The data frame representation is convenient for many common operations involving grids. However some apparently simple operations can be either tricky or inefficient. For instance, retrieving the grid levels from the data frame representation is quite tedious, since it requires something like a `tapply`. Although the list of the levels was provided in the creation of the grid, this information is no longer readily accessible from the object, and it would be convenient to have it attached to the object. As another example, finding the boundary points of the grid from the data frame representation is no more straightforward. Using a data frame with attributes could help in many problems. However, it should then be checked that the order of the columns remains the same in the data frame and in the list, and also that the data frame contains a tensor product of finite sets as assumed.

In the **smint** package, it was decided to define a S4 *class* for the grid objects, under the name `Grid`, in the aim that standard *methods* give the frequently needed information. As in the data frame representation, the nodes will be considered as ordered – so a `Grid` object can be thought of as a *traveller salesman's path* visiting each node of the grid exactly once.

1.2 Creating a `Grid` object

A `Grid` object can be created using the quite versatile `Grid` creator, similar to the list version of `expand.grid`

```
myGrid1 <- Grid(level = list("x" = c(0.0, 0.2, 1.0), "y" = c(1.0, 2.5, 3.0),
                             "z" = c(0.2, 0.4)))
myGrid1

## Grid Data object
##   o dimension : 3
##   o dim names : x, y, z
##   o number of nodes : 3 (x), 3 (y), 2 (z)
##   o total number of nodes : 18
```

An object named `myGrid` and class `"Grid"` is created; as is usual with S4 classes, by typing the name of an object one invokes the `show` method[1] A number of methods can be invoked

[1] This is similar to the invocation of `print` for S3 classes/objects.

```r
length(myGrid1)

## [1] 18

nlevels(myGrid1)

## x y z
## 3 3 2

levels(myGrid1)

## $x
## [1] 0.0 0.2 1.0
##
## $y
## [1] 1.0 2.5 3.0
##
## $z
## [1] 0.2 0.4

dimnames(myGrid1)

## [1] "x" "y" "z"
```

A more concise call to the creator can sometimes be used. By default, the standard hypercube $[0, 1]^d$ and regularly spaced coordinates along the axes will be used.

```r
myGrid2 <- Grid(nlevels = c(3, 3, 2))
myGrid2

## Grid Data object
##   o dimension : 3
##   o dim names : X1, X2, X3
##   o number of nodes : 3 (X1), 3 (X2), 2 (X3)
##   o total number of nodes : 18

myGrid2 <- Grid(nlevels = c("a" = 3, "b" = 3, "c" = 2))
myGrid2

## Grid Data object
##   o dimension : 3
##   o dim names : a, b, c
##   o number of nodes : 3 (a), 3 (b), 2 (c)
##   o total number of nodes : 18

dimnames(myGrid2) <- c("A", "B", "C")
myGrid2

## Grid Data object
##   o dimension : 3
##   o dim names : A, B, C
##   o number of nodes : 3 (A), 3 (B), 2 (C)
##   o total number of nodes : 18
```

The dimnanes are by default `"X1"`, `"X2"` and so on, but they can be specified by using a named vector for the number of levels. They can also be changed by using the replacement method `dimnames<-` illustrated before.

Another possibility to create a `Grid` object is to use the `randGrid` which generates a random `Grid` object. This can be of some help to test the results of interpolation methods.

```
set.seed(123)
rGrid <- randGrid()
rGrid

## Grid Data object
##   o dimension : 3
##   o dim names : X1, X2, X3
##   o number of nodes : 5 (X1), 3 (X2), 6 (X3)
##   o total number of nodes : 90

levels(rGrid)

## $X1
## [1] 0.0455565 0.5281055 0.5514350 0.8924190 0.9404673
##
## $X2
## [1] 0.4533342 0.4566147 0.9568333
##
## $X3
## [1] 0.04205953 0.10292468 0.24608773 0.57263340 0.67757064 0.89982497
```

The user can control some features of the object by using the optional arguments of `randGrid` such as `dim` or `nlevels`. Be aware that a `Grid` object with moderate dimension, e.g. 6 or 7 might then have a considerable number of nodes.

Back to data frames

One of the most common operation is the transformation of the object into a data frame.

```
df <- as.data.frame(myGrid1)
head(df)

##     x   y   z
## 1 0.0 1.0 0.2
## 2 0.2 1.0 0.2
## 3 1.0 1.0 0.2
## 4 0.0 2.5 0.2
## 5 0.2 2.5 0.2
## 6 1.0 2.5 0.2
```

Conversely, a `Grid` in arbitrary dimension can be created by coercing a data frame or a matrix with suitable content using `as.Grid`. Obviously, not every data frame can be used, and an error will occur when the data frame is not suitable. The two coercions are nearly reversible.

```
identical(as.Grid(as.data.frame(myGrid1)), myGrid1)

## [1] TRUE
```

However, we do not recommend to compare `Grid` objects with `identical`, since the result can misleadingly be `FALSE` only because of the rounding of the levels.

Note that `as.Grid` will be quite slow for a large grid object. As a major difference with the data frame representation, a `Grid` object stores all the n possible combinations of the levels as vectors of length d with *integer* values rather than doubles. So for a large grid, the `Grid` representation needs only about 1/8 of the space required by the data frame representation.

Plot method

The trained R user might have guessed at this point that the `plot` method for the class `Grid` will produce exactly the plot that would have resulted from using the `plot` method after a coercion to a data frame using `as.data.frame`. This is true as far as only one formal is used. When the grid dimension is > 2 we noticed that the pairs representation can be very misleading since several points in the d-dimensional space collapse into the same projection in a particular pair plot. A simple way to avoid this is to jitterise the points. A `jitter` argument was added to the `as.data.frame` coercion method and to the `plot` method. Combined with semi-transparent colours, an improved representation results, see figure 1.2. The points can be given different colours, size, ... by using vectors. Remind that the order of points is then essential.

1.3 Operations with `Grid` objects

Flat grids

A grid in which one dimension has only one level will be said to be flat. This can be compared to a flat array, e.g. a matrix with one row or one column. By default, R drops the unnecessary dimensions of flat arrays when subsetting, and the user can also do this on purpose with the `drop` function. A similar operation is possible for `Grid` objects by using the `drop_Grid` function which drops the dimensions of flat grids

```
flatGrid <- Grid(nlevels = c(2, 1, 3))
flatGrid

## Grid Data object
##   o dimension : 3
##   o dim names : X1, X2, X3
##   o number of nodes : 2 (X1), 1 (X2), 3 (X3)
##   o total number of nodes : 6

drop_Grid(flatGrid)

## Grid Data object
##   o dimension : 2
##   o dim names : X1, X3
##   o number of nodes : 2 (X1), 3 (X3)
##   o total number of nodes : 6
```

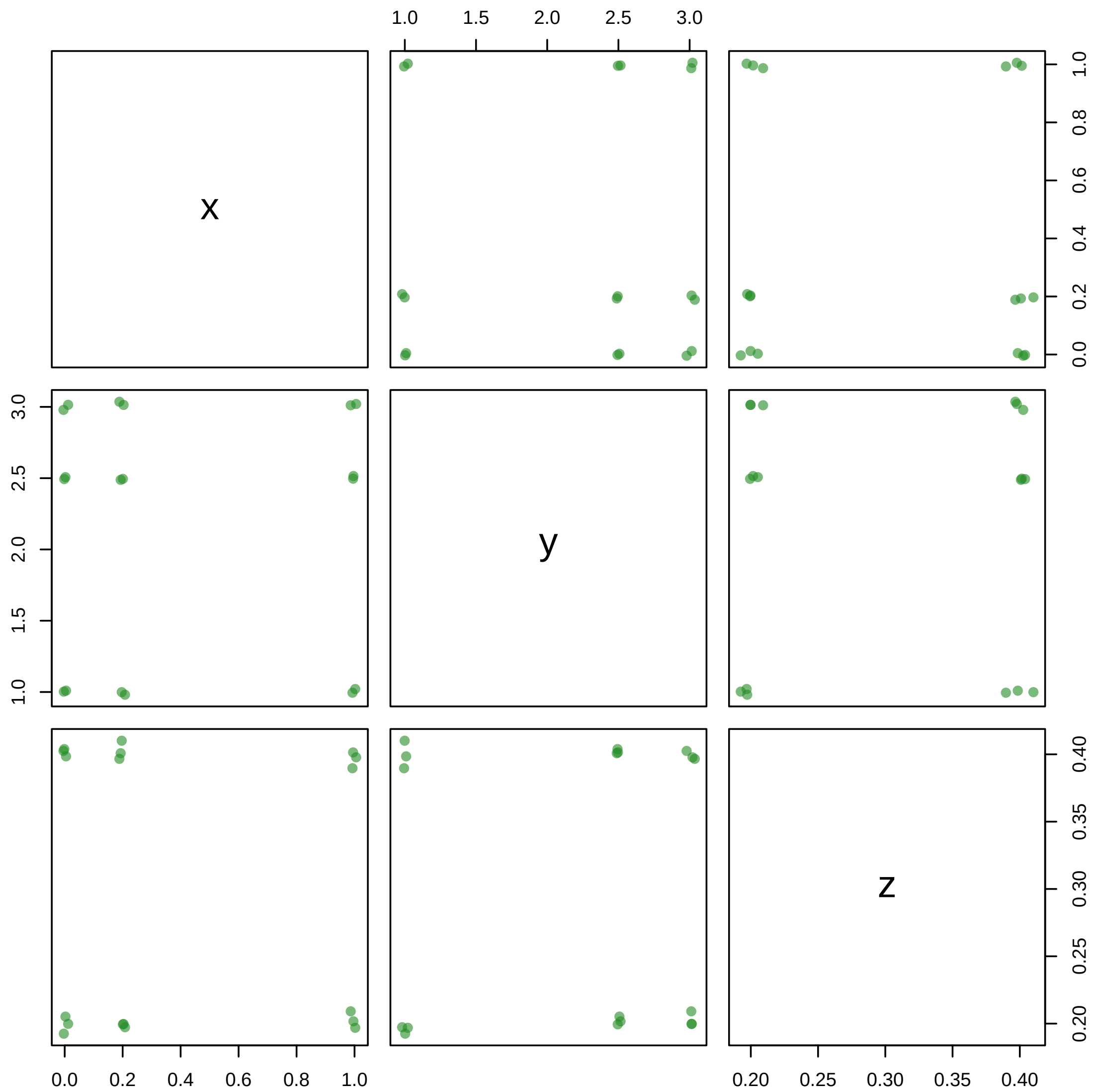

Figure 1.2: Pairs plot produced by the `plot` method of the `Grid` class with the formal argument `jitter` set to `TRUE`.

The drop_Grid function can be used from the matrix or data frame representation; however it will then make use of the coercion method as.Grid, hence of tapply.

Applying a function

The apply_Grid function can be used with its first formal object of class "Grid". Using the function my3dFun defined in section 1.1 page 5

```
fGrid <- apply_Grid(myGrid1, fun = my3dFun)
max(abs(fGrid - f))

## [1] 0
```

So we get the same result as with apply.

Generalised transposition

A useful operation is the *generalised transpose*, corresponding to the aperm method in the **base** package. This is equivalent to a permutation of the columns in the data frame or matrix representation, and the *order of nodes* remains unchanged. So, if a function or response has been computed on the nodes and is stored as a response vector f, this will remain unchanged when aperm is used on the Grid object X with a subsequent modification of the function.

```
myGrid1p <- aperm(myGrid1, perm = c(3, 1, 2))
myGrid1p

## Grid Data object
##   o dimension : 3
##   o dim names : z, x, y
##   o number of nodes : 2 (z), 3 (x), 3 (y)
##   o total number of nodes : 18

my3dFunp <- function(x) x[2]^3 + 2 * x[3] + x[1]^2
fGridp <- apply_Grid(myGrid1p, fun = my3dFunp)
max(abs(fGridp - f))

## [1] 0
```

In the definition of the function, the elements of the formal x must correspond to z, x, and y in that order. It is possible to make the function independent of the order of the dimensions by using a named vector as formal.

```
my3dFuni <- function(x) x["x"]^3 + 2 * x["y"] + x["z"]^2
fGridi <- apply_Grid(myGrid1p, fun = my3dFuni)
max(abs(fGridi - f))

## [1] 0

fGridi0 <- apply_Grid(myGrid1, fun = my3dFuni)
max(abs(fGridi0 - f))

## [1] 0
```

This works because `apply_Grid` relies on the `apply` function for matrices, in which the first function formal can be named. Using a named vector as formal should be preferred when possible.

Remark. The `with` method is not yet implemented for a `data` formal with class `"Grid"`. It can easily be used through a coercion to data frame.

Range and scale

The `range_Grid` and `scale_Grid` functions can be used to get or set the range of a `Grid` object

```
range_Grid(myGrid1)

##     x y   z
## min 0 1 0.2
## max 1 3 0.4

myGrid1s <- scale_Grid(myGrid1)
range_Grid(myGrid1s)

##     x y z
## min 0 0 0
## max 1 1 1
```

The scaling transformation can be controlled with the `fromRange` and `toRange` formal arguments.

Boundary points

The `boundary_Grid` function identifies the points located on the boundary, assuming that the smallest and largest levels are boundary levels for each of the d dimensions. When the number of levels is $\leqslant 2$ for one dimension or more, all points are boundary points.
See figure 1.3. The number of interior (non-boundary) points is $\prod_{i=1}^{d}[n_i - 2]$, i.e. $3 \times 5 \times 4 = 60$ for this example.

Remark. When choosing the same number of points by dimension, say n_1, the proportion of interior (non-boundary) points is given by $[1 - 2/n_1]^d$ and turns out to be small for dimensions $d \geqslant 5$. This is one of the effects of the *curse of dimensionality*: when the dimension d is large, most grid points are located on the boundary of the hyper-rectangle.

Sampling

A common need when working with a grid is to draw random points, either at grid points or within the hyper-rectangle of interest. The `sampleIn` method was written for that. Using our `myGrid3` object defined earlier with 210 nodes.

```
X3s <- sampleIn(myGrid3, size = 100)
head(X3s)

##               X          Y         Z
## [1,] 0.21506675 0.74678791 0.6305725
## [2,] 0.86953229 0.41931352 0.3908899
```

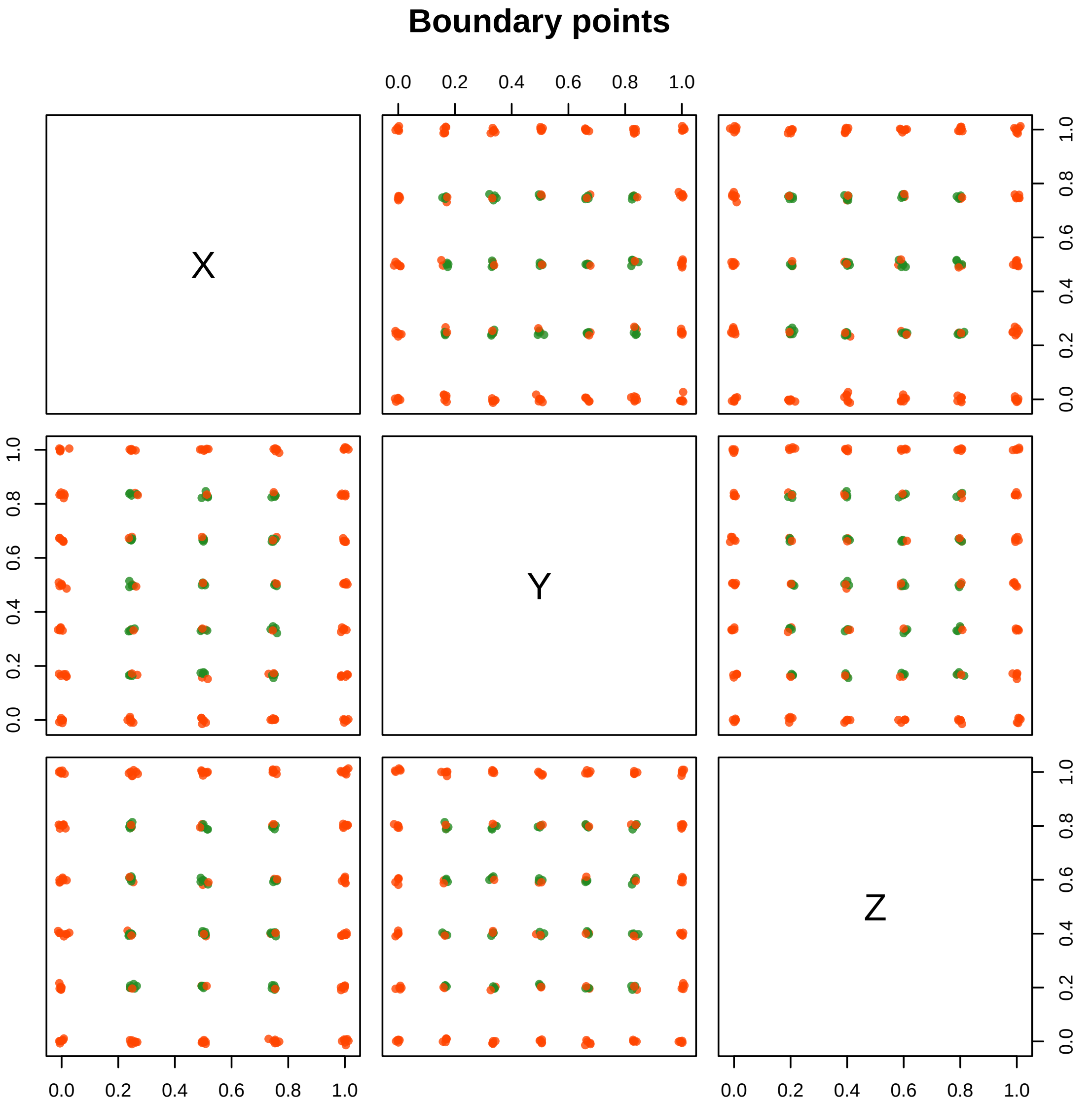

Figure 1.3: Grid data with $n_x = 5$, $n_y = 7$ and $n_z = 6$ levels. Boundary points are plotted in orange and non-boundary points are in green. Among the 210 nodes, 150 are on the boundary.

```
## [3,] 0.06083806 0.58214390 0.4322536
## [4,] 0.23489960 0.83073406 0.3486340
## [5,] 0.99301551 0.01353063 0.7935578
## [6,] 0.67819773 0.01152176 0.8869361

X3ss <- sampleIn(myGrid3, size = 100, atSample = TRUE)
head(X3ss)

##        X         Y   Z
## 143 0.50 0.0000000 0.8
## 201 0.00 0.8333333 1.0
## 88  0.50 0.5000000 0.4
## 83  0.50 0.3333333 0.4
## 139 0.75 1.0000000 0.6
## 33  0.50 1.0000000 0.0
```

The result is a matrix that can be coerced to a data frame when needed.

Reshaping response(s) to an array

When $d = 2$, a common practice is to provide a response as a matrix with row i corresponding to the i-th value of the first dimension x_1 (or x) and the column j matching the j-th value of the second dimension x_2 (or y). This form is often required to produce a contour plot or a perspective plot (not shown here). The `array_Grid` function can be used for that.

```
plotGrid <- Grid(nlevels = c(10, 10))
F <- apply_Grid(plotGrid, branin)
aF <- array_Grid(X = plotGrid, Y = F)
round(aF)

##        X2=0 X2=0.1 X2=0.2 X2=0.3 X2=0.4 X2=0.6 X2=0.7 X2=0.8 X2=0.9 X2=1
## X1=0    306    252    203    160    122     90     63     43     27   17
## X1=0.1  162    123     89     60     37     20      8      2      1    6
## X1=0.2   90     63     41     25     15     10     10     16     28   45
## X1=0.3   56     38     27     21     20     25     36     52     73  101
## X1=0.4   23     13      9     11     18     31     49     73    102  137
## X1=0.6    5      1      2      9     21     39     63     92    127  167
## X1=0.7   14     13     17     27     43     63     90    122    160  203
## X1=0.8   20     19     24     35     51     72    100    132    171  214
## X1=0.9    8      6      9     18     32     52     77    108    145  187
## X1=1     10      3      2      7     17     33     55     81    114  152

contour(aF, nlevels = 20)
```

This rule obviously generalises to a larger dimension d: a response can be reshaped into a d-dimensional array with dimension $[n_1, n_2, \ldots, n_d]$. Moreover, when m responses are available we can use a $d+1$-dimensional array with the response index as the slice index in the $d+1$ dimension.

Subgrid

The `subset_Grid` function allows the selection of a sub-grid by selecting the nodes using a clause for *one* dimension, using the `subset` argument.

```
subset_Grid(myGrid1, subset = y > 2)

##  [1]  4  5  6  7  8  9 13 14 15 16 17 18
```

We get here the indices of the nodes in the subset, which would be convenient e.g. to find the corresponding responses in a vector. Alternatively, one can return the result as a `Grid`

```
subset_Grid(myGrid1, subset = y > 2, type = "Grid")

## Grid Data object
##    o dimension : 3
##    o dim names : x, y, z
##    o number of nodes : 3 (x), 2 (y), 2 (z)
##    o total number of nodes : 12
```

If the sub-grid turns out to be flat because only one node is selected, the dimension used in the selection will be dropped by default. This would here have happened with `subset = y > 3`.

Why uppercase X?

The `*_Grid` functions (see table 1.1) are "pseudo-methods", and are intended to work for `Grid` objects as well as for their data frame or matrix representation, and they have their first argument named `X` to remind of that.

1.4 Summary

- The **smint** package provides a `Grid` S4 class with some methods and dedicated functions.
- A `Grid` object is efficiently coerced into a data frame when needed but it also contains information about the grid characteristics: number of levels, levels, ...
- A `Grid` object contains a numbering of the grid points which is used to match the grid points and the responses. Several functions are provided to apply a (test) function on a grid, reshape responses to an array, remove unneeded dimensions and more.

Method	*df?*	*Goal*
`aperm(x, perm)`		Generalised transposition.
`closest(X, XNew, ...)`		Find the points in `X` that are closest to those in `XNew`.
`dim`		Get the dimension d.
`dimnames` `dimnames<-`		Get or set the dimension names.
`nlevels` `levels`		Get the number and values of the levels.
`plot(x, y, jitter = FALSE, ...)`		Pairs plot for a `Grid` object `x`.
`sampleIn`	`y`	Draw `size` random points in a grid. Formal `atSample`.
`show`		Show information about the object.

Function	*df?*	*Goal*
`apply_Grid(X, fun)`	`n`	Apply the function `fun` to each node of a grid `X`.
`array_Grid(X, Y)`	`n`	Reshape as array the response(s) `Y` for a grid `X`.
`boundary_Grid(X)`	`n`	Identify the boundary points in a grid `X`.
`drop_Grid(X)`	`n`	Drop "flat" dimensions with only one level.
`range_Grid(X)`	`y`	Get the ranges as a 2-rows matrix.
`scale_Grid(X, fromRange, toRange)`	`y`	Transform to $[0,1]^d$ or to a given hyper-rectangle.
`subset_Grid(X, subset, type, drop)`	`n`	Extract a sub-grid.

Table 1.1: Methods and functions. For the "`*_Grid` functions" with name ending by `_Grid`, the first part of the name is most of time not a method name, e.g. `drop` is not a method as long as only base packages are used. The functions for which `X` can be a matrix or data frame are shown by a `y` in the column *df?*.

Index