The terra package

Robert J. Hijmans

Nov 30, 2023

CONTENTS

CHAPTER

ONE

THE TERRA PACKAGE

This vignette describes the *R* package `terra`. A raster is a spatial (geographic) data structure that divides a region into rectangles called "cells" (or "pixels") that can store one or more values for each of these cells. Such a data structure is also referred to as a "grid" and is often contrasted with "vector" data that is used to represent points, lines, and polygons.

The `terra` package has functions for creating, reading, manipulating, and writing raster data. The package provides, among other things, general raster data manipulation functions that can easily be used to develop more specific functions. For example, there are functions to read a chunk of raster values from a file or to convert cell numbers to coordinates and back. The package also implements raster algebra and most functions for raster data manipulation.

A notable feature of the `terra` package is that it can work with raster datasets that are stored on disk and are too large to be loaded into memory (RAM). The package can work with large files because the objects it creates from these files only contain information about the structure of the data, such as the number of rows and columns, the spatial extent, and the filename, but it does not attempt to read all the cell values in memory. In computations with these objects, data is processed in chunks. If no output filename is specified to a function, and the output raster is too large to keep in memory, the results are written to a temporary file.

To understand what is covered in this vignette, you must understand the basics of the *R* language. There is a multitude of on-line and other resources that can help you to get acquainted with it.

In the next section, some general aspects of the design of the `terra` package are discussed, notably the structure of the main classes, and what they represent. The use of the package is illustrated in subsequent sections. `terra` has a large number of functions, not all of them are discussed here, and those that are discussed are mentioned only briefly. See the help files of the package for more information on individual functions and `help("terra")` for an index of functions by topic.

CHAPTER

TWO

CLASSES

The package is built around a number of "classes" of which the `SpatRaster` and `SpatVector` are the most important.

2.1 SpatRaster

A `SpatRaster` represents multi-layer (variable) raster data. A `SpatRaster` object stores a number of fundamental parameters that describe it. These include the number of columns and rows, the coordinates of its spatial extent ('bounding box'), and the coordinate reference system (the 'map projection'). In addition, a `SpatRaster` can store information about the file(s) in which the raster cell values are stored (if there are such files) — as raster cell values can also be held in memory.

2.2 SpatVector

A `SpatVector` represents "vector" data, that is, points, lines or polygon geometries and their tabular attributes.

2.3 SpatExtent

Class for spatial extent

CHAPTER

THREE

CREATING SPATRASTER OBJECTS

A `SpatRaster` can easily be created from scratch using the function `rast`. The default settings will create a global raster data structure with a longitude/latitude coordinate reference system and 1 by 1 degree cells. You can change these settings by providing additional arguments such as **xmin**, **nrow**, **ncol**, and/or **crs**, to the function. You can also change these parameters after creating the object. If you set the projection, this is only to properly define it, not to change it. To transform a `SpatRaster` to another coordinate reference system (projection) you can use the function `warp`.

Here is an example of creating and changing a `SpatRaster` object 'r' from scratch.

SpatRaster with default geometry parameters

```
library(terra)
## terra 1.7.62
x <- rast()
x
## class       : SpatRaster
## dimensions  : 180, 360, 1  (nrow, ncol, nlyr)
## resolution  : 1, 1  (x, y)
## extent      : -180, 180, -90, 90  (xmin, xmax, ymin, ymax)
## coord. ref. : lon/lat WGS 84 (CRS84) (OGC:CRS84)
```

With other parameters

```
x <- rast(ncol=36, nrow=18, xmin=-1000, xmax=1000, ymin=-100, ymax=900)
res(x)
## [1] 55.55556 55.55556
```

Change the spatial resolution of an existing object

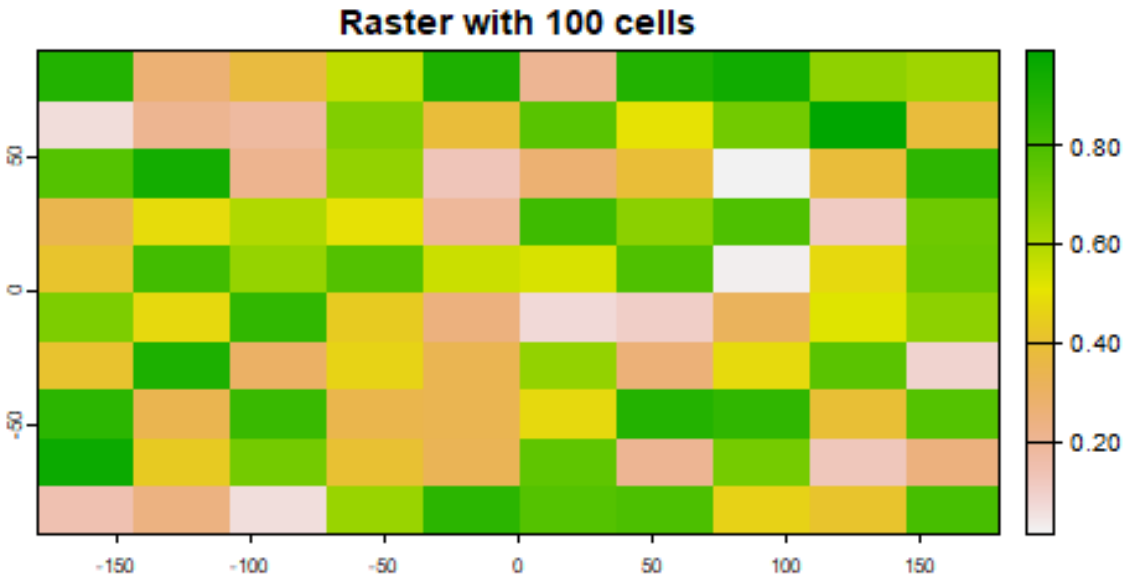
```
res(x) <- 100
res(x)
## [1] 100 100
ncol(x)
## [1] 20
# change the numer of columns (affects resolution)
ncol(x) <- 18
ncol(x)
## [1] 18
res(x)
## [1] 111.1111 100.0000
```

Set the coordinate reference system (CRS) (define the projection)

```
crs(x) <- "+proj=utm +zone=48 +datum=WGS84"
x
## class       : SpatRaster
## dimensions  : 10, 18, 1  (nrow, ncol, nlyr)
## resolution  : 111.1111, 100  (x, y)
## extent      : -1000, 1000, -100, 900  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=utm +zone=48 +datum=WGS84 +units=m +no_defs
```

The object **x** created in the example above only consist of a "skeleton", that is, we have defined the number of rows and columns, and where the raster is located in geographic space, but there are no cell-values associated with it. Setting and accessing values is illustrated below.

```
r <- rast(ncol=10, nrow=10)
ncell(r)
## [1] 100
hasValues(r)
## [1] FALSE

# use the 'values' function, e.g.,
values(r) <- 1:ncell(r)
# or
set.seed(0)
values(r) <- runif(ncell(r))

hasValues(r)
## [1] TRUE
sources(r)
## [1] ""
values(r)[1:10]
##  [1] 0.8966972 0.2655087 0.3721239 0.5728534 0.9082078 0.2016819 0.8983897
##  [8] 0.9446753 0.6607978 0.6291140

plot(r, main='Raster with 100 cells')
```



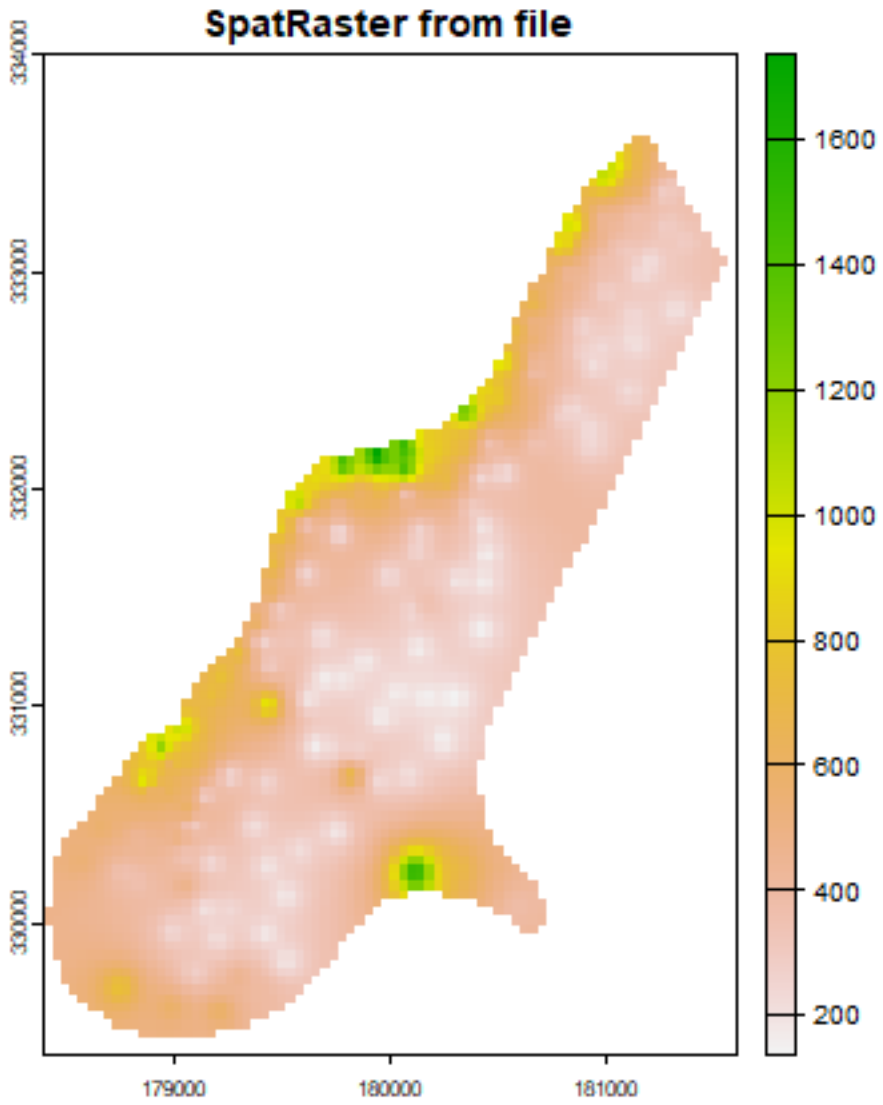
In some cases, for example when you change the number of columns or rows, you will lose the values associated with the SpatRaster if there were any (or the link to a file if there was one). The same applies, in most cases, if you change the resolution directly (as this can affect the number of rows or columns). Values are not lost when changing the extent as this change adjusts the resolution, but does not change the number of rows or columns.

```
hasValues(r)
## [1] TRUE
res(r)
## [1] 36 18
dim(r)
## [1] 10 10  1
xmax(r)
## [1] 180

# change the maximum x coordinate of the extent (bounding box) of the SpatRaster
xmax(r) <- 0

hasValues(r)
## [1] TRUE
res(r)
## [1] 18 18
dim(r)
## [1] 10 10  1

ncol(r) <- 6
hasValues(r)
## [1] FALSE
res(r)
## [1] 30 18
dim(r)
## [1] 10  6  1
xmax(r)
## [1] 0
```

The function terra also allows you to create a SpatRaster from another object, including another SpatRaster, or SpatRaster objects from the "terra" package.

It is more common, however, to create a SpatRaster object from a file. The raster package can use raster files in several formats, including some 'natively' supported formats and other formats via the rgdal package. Supported formats for reading include GeoTIFF, ESRI, ENVI, and ERDAS. Most formats supported for reading can also be written to.

```
# get the name of an example file installed with the package
# do not use this construction of your own files
filename <- system.file("ex/meuse.tif", package="terra")
filename
## [1] "C:/soft/R/R-4.3.2/library/terra/ex/meuse.tif"
r <- rast(filename)
sources(r)
## [1] "C:/soft/R/R-4.3.2/library/terra/ex/meuse.tif"
hasValues(r)
## [1] TRUE
plot(r, main='SpatRaster from file')
```



Multi-layer objects can be created in memory (from `SpatRaster` objects) or from files.

```
# create three identical SpatRaster objects
r1 <- r2 <- r3 <- rast(nrow=10, ncol=10)
# Assign random cell values
values(r1) <- runif(ncell(r1))
values(r2) <- runif(ncell(r2))
values(r3) <- runif(ncell(r3))
```

Combine the three SpatRaster objects into a single object with three layers.

```
s <- c(r1, r2, r3)
s
## class       : SpatRaster
## dimensions  : 10, 10, 3  (nrow, ncol, nlyr)
## resolution  : 36, 18  (x, y)
## extent      : -180, 180, -90, 90  (xmin, xmax, ymin, ymax)
```

(continues on next page)

(continued from previous page)

```
## coord. ref. : lon/lat WGS 84 (CRS84) (OGC:CRS84)
## source(s)   : memory
## names       :      lyr.1,      lyr.1,      lyr.1
## min values  : 0.01307758, 0.02778712, 0.06380247
## max values  : 0.99268406, 0.98156346, 0.99607737
nlyr(s)
## [1] 3
```

Create a multilayer SpatRaster from file

```
filename <- system.file("ex/logo.tif", package="terra")
filename
## [1] "C:/soft/R/R-4.3.2/library/terra/ex/logo.tif"
b <- rast(filename)
b
## class       : SpatRaster
## dimensions  : 77, 101, 3  (nrow, ncol, nlyr)
## resolution  : 1, 1  (x, y)
## extent      : 0, 101, 0, 77  (xmin, xmax, ymin, ymax)
## coord. ref. : Cartesian (Meter)
## source      : logo.tif
## colors RGB  : 1, 2, 3
## names       : red, green, blue
## min values  :   0,     0,    0
## max values  : 255,   255,  255
nlyr(b)
## [1] 3
```

Extract a layer

```
r <- b[[2]]
```

CHAPTER

FOUR

RASTER ALGEBRA

Many generic functions that allow for simple and elegant raster algebra have been implemented for `SpatRaster` objects, including the normal algebraic operators such as +, -, *, /, logical operators such as >, >=, <, ==, ! } and functions such as `abs`, `round`, `ceiling`, `floor`, `trunc`, `sqrt`, `log`, `log10`, `exp`, `cos`, `sin`, `max`, `min`, `range`, `prod`, `sum`, `any`, `all`. In these functions you can mix `terra` objects with numbers, as long as the first argument is a `terra` object.

```
library(terra)
## terra 1.7.62
# create an empty SpatRaster
r <- rast(ncol=10, nrow=10)
# assign values to cells
values(r) <- 1:ncell(r)
s <- r + 10
s <- sqrt(s)
s <- s * r + 5
values(r) <- runif(ncell(r))
r <- round(r)
r <- r == 1
```

You can also use replacement functions (not yet supported)

```
s[r] <- -0.5
s[!r] <- 5
s[s == 5] <- 15
```

If you use multiple `SpatRaster` objects (in functions where this is relevant, such as range), these must have the same resolution and origin. The origin of a `SpatRaster` object is the point closest to (0, 0) that you could get if you moved from a corners of a `SpatRaster` object towards that point in steps of the `x` and ``y` resolution. Normally these objects would also have the same extent, but if they do not, the returned object covers the spatial intersection of the objects used.

When you use multiple multi-layer objects with different numbers or layers, the 'shorter' objects are 'recycled'. For example, if you multiply a 4-layer object (a1, a2, a3, a4) with a 2-layer object (b1, b2), the result is a four-layer object (a1b1, a2b2, a3b1, a3b2).

```
r <- rast(ncol=5, nrow=5)
values(r) <- 1
s <- c(r, r+1)
q <- c(r, r+2, r+4, r+6)
x <- r + s + q
x
## class       : SpatRaster
```

(continues on next page)

(continued from previous page)

```
## dimensions  : 5, 5, 4  (nrow, ncol, nlyr)
## resolution  : 72, 36  (x, y)
## extent      : -180, 180, -90, 90  (xmin, xmax, ymin, ymax)
## coord. ref. : lon/lat WGS 84 (CRS84) (OGC:CRS84)
## source(s)   : memory
## names       : lyr1, lyr2, lyr3, lyr4
## min values  :    3,    6,    7,   10
## max values  :    3,    6,    7,   10
```

Summary functions (**min, max, mean, prod, sum, Median, cv, range, any, all**) always return a `SpatRaster` object. Perhaps this is not obvious when using functions like **min, sum or mean**.

```
a <- mean(r, s, 10)
b <- sum(r, s)
st <- c(r, s, a, b)
sst <- sum(st)
sst
## class       : SpatRaster
## dimensions  : 5, 5, 1  (nrow, ncol, nlyr)
## resolution  : 72, 36  (x, y)
## extent      : -180, 180, -90, 90  (xmin, xmax, ymin, ymax)
## coord. ref. : lon/lat WGS 84 (CRS84) (OGC:CRS84)
## source(s)   : memory
## name        :      sum
## min value   : 17.33333
## max value   : 17.33333
```

Use `global` if instead of a `SpatRaster` you want a single number summarizing the cell values of each layer.

```
global(st, 'sum')
##              sum
## lyr.1    25.0000
## lyr.1.1  25.0000
## lyr.1.2  50.0000
## lyr1    100.0000
## lyr2    108.3333
## lyr1.1   50.0000
## lyr2.1   75.0000
global(sst, 'sum')
##          sum
## sum 433.3333
```

CHAPTER

FIVE

HIGH-LEVEL METHODS

Several 'high level' methods (functions) have been implemented for `SpatRaster` objects. 'High level' refers to methods that you would normally find in a GIS program that supports raster data. Here we briefly discuss some of these. See the help files for more detailed descriptions.

The high-level methods have some arguments in common. The first argument is typically 'x' or 'object' and in most cases it is a `SpatRaster` or a `SpatVector`. It is followed by one or more arguments specific to the method (either additional `SpatRaster` objects or other arguments), followed by a filename="" and "..." arguments.

The default filename is an empty character "". If you do not specify a filename, the default action for the method is to return a `terra` object that only exists in memory. However, if the method deems that the `terra` object to be created would be too large to hold memory it is written to a temporary file instead.

The "..." argument allows for setting additional arguments that are relevant when writing values to a file: the file format, datatype (e.g. integer or real values), and a to indicate whether existing files should be overwritten.

5.1 Modifying a SpatRaster object

There are several methods that deal with modifying the spatial extent of `SpatRaster` objects. The `crop` method lets you take a geographic subset of a larger `terra` object. You can crop a `SpatRaster` by providing an extent object or another spatial object from which an extent can be extracted (objects from classes deriving from Raster and from Spatial in the sp package). An easy way to get an extent object is to plot a `SpatRaster` and then use `drawExtent` to visually determine the new extent (bounding box) to provide to the crop method.

`trim` crops a `SpatRaster` by removing the outer rows and columns that only contain `NA` values. In contrast, `extend` adds new rows and/or columns with `NA` values. The purpose of this could be to create a new `SpatRaster` with the same Extent of another larger `SpatRaster` such that the can be used together in other methods.

The `merge` method lets you merge 2 or more `SpatRaster` objects into a single new object. The input objects must have the same resolution and origin (such that their cells neatly fit into a single larger raster). If this is not the case you can first adjust one of the `SpatRaster` objects with use `(dis)aggregate` or `resample`.

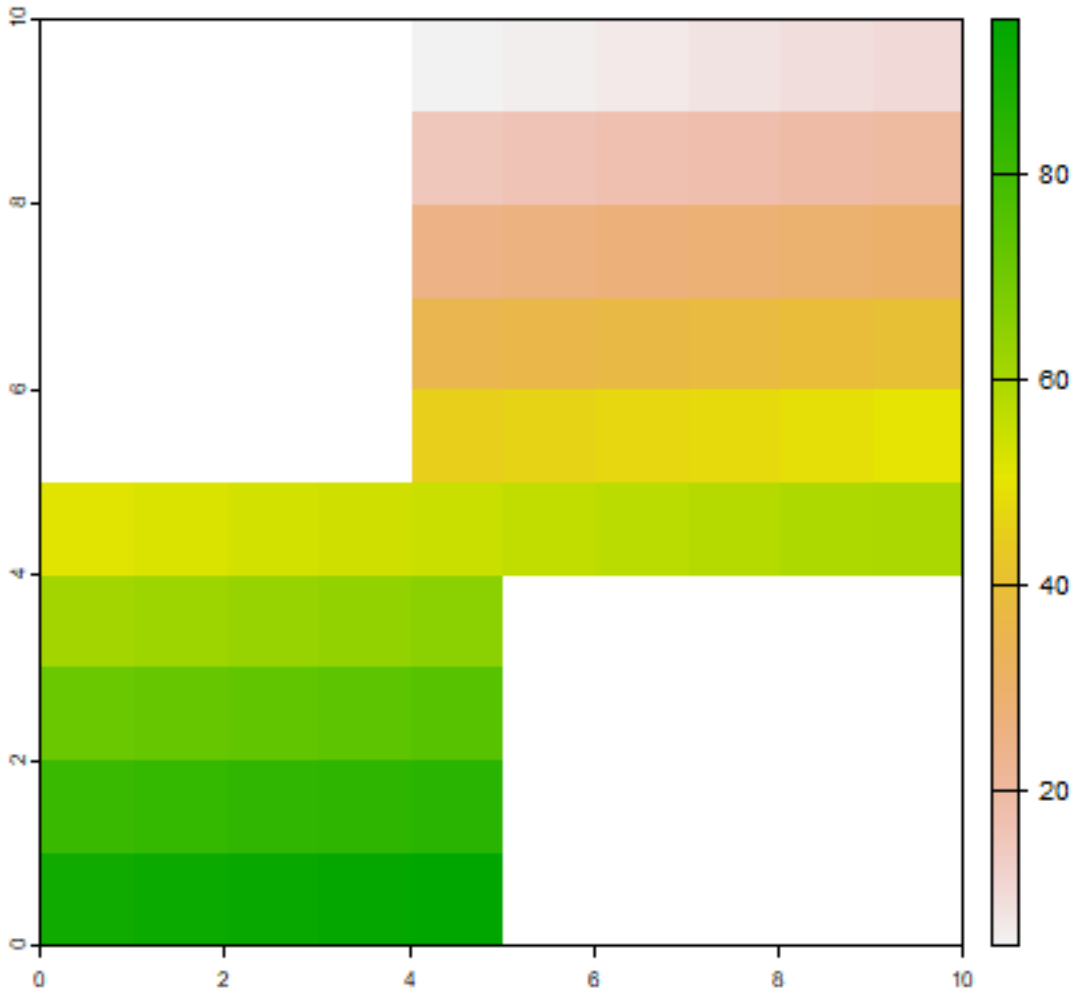
`aggregate` and `disagg` allow for changing the resolution (cell size) of a `SpatRaster` object. In the case of `aggregate`, you need to specify a function determining what to do with the grouped cell values (e.g. `mean`). It is possible to specify different (dis)aggregation factors in the x and y direction. `aggregate` and `disagg` are the best methods when adjusting cells size only, with an integer step (e.g. each side 2 times smaller or larger), but in some cases that is not possible.

For example, you may need nearly the same cell size, while shifting the cell centers. In those cases, the `resample` method can be used. It can do either nearest neighbor assignments (for categorical data) or bilinear interpolation (for numerical data). Simple linear shifts of a Raster object can be accomplished with the `shift` method or with the `extent` method. `resample` should not be used to create a SpatRaster object with much larger resolution. If such adjustments need to be made then you can first use aggregate.

With the `warp` method you can transform values of a `SpatRaster` to a new object with a different coordinate reference system.

Here are some simple examples.

```
library(terra)
## terra 1.7.62
r <- rast(ncol=10, nrow=10, xmin=0, xmax=10, ymin=0, ymax=10)
values(r) <- 1:ncell(r)
ra <- aggregate(r, 2)
r1 <- crop(r, ext(0, 5, 0, 5))
r2 <- crop(r, ext(4, 10, 4, 10))
m <- merge(r1, r2, filename='test.tif', overwrite=TRUE)
plot(m)
```



`bf` lets you flip the data (reverse order) in horizontal or vertical direction – typically to correct for a 'communication problem' between different R packages or a misinterpreted file. `rotate` lets you rotate longitude/latitude rasters that have longitudes from 0 to 360 degrees (often used by climatologists) to the standard -180 to 180 degrees system. With

t you can rotate a SpatRaster object 90 degrees.

5.2 lapp

The lapp (for layer-apply) method can be used as an alternative to the raster algebra discussed above. Like the methods discussed in the following subsections provide either easy to use short-hand, or more efficient computation for large (file based) objects.

With lapp you can combine multiple SpatRaster objects. The related method mask removes all values from one layer that are NA in another layer, and cover combines two layers by taking the values of the first layer except where these are NA.

5.3 app

The app method allows you to do a computation across the layers of a terra object by providing a function (like apply on a matrix or data.frame). If you supply a SpatRaster, another SpatRaster is returned. tapp computes summary type layers for subsets of a SpatRaster (like tapply on a matrix or data.frame).

5.4 classify

You can use cut or classify to replace ranges of values with single values, or subs to substitute (replace) single values with other values.

```
r <- rast(ncol=3, nrow=2)
values(r) <- 1:ncell(r)
values(r)
##      lyr.1
## [1,]     1
## [2,]     2
## [3,]     3
## [4,]     4
## [5,]     5
## [6,]     6
s <- app(r, fun=function(x){ x[x < 4] <- NA; return(x)} )
as.matrix(s)
##      lyr.1
## [1,]    NA
## [2,]    NA
## [3,]    NA
## [4,]     4
## [5,]     5
## [6,]     6
t <- lapp(c(r, s), fun=function(x, y){ x / (2 * sqrt(y)) + 5 } )
as.matrix(t)
##          lyr1
## [1,]       NA
## [2,]       NA
## [3,]       NA
## [4,] 6.000000
```

(continues on next page)

(continued from previous page)

```
## [5,] 6.118034
## [6,] 6.224745
u <- mask(r, t)
as.matrix(u)
##      lyr.1
## [1,]    NA
## [2,]    NA
## [3,]    NA
## [4,]     4
## [5,]     5
## [6,]     6
v <- u==s
as.matrix(v)
##      lyr.1
## [1,]    NA
## [2,]    NA
## [3,]    NA
## [4,]  TRUE
## [5,]  TRUE
## [6,]  TRUE
w <- cover(t, r)
as.matrix(w)
##          lyr1
## [1,] 1.000000
## [2,] 2.000000
## [3,] 3.000000
## [4,] 6.000000
## [5,] 6.118034
## [6,] 6.224745
x <- classify(w, c(0,2,1,  2,5,2, 4,10,3))
as.matrix(x)
##      lyr1
## [1,]    0
## [2,]    1
## [3,]    4
## [4,]    7
## [5,]    7
## [6,]    7
y <- classify(x, cbind(id=c(2,3), v=c(40,50)))
as.matrix(y)
##      lyr1
## [1,]    0
## [2,]    1
## [3,]    4
## [4,]    7
## [5,]    7
## [6,]    7
```

5.5 Focal

The `focal` method currently only works for (single layer) SpatRaster objects. It uses values in a neighborhood of cells around a focal cell, and computes a value that is stored in the focal cell of the output SpatRaster. The neighborhood is a user-defined a matrix of weights and could approximate any shape by giving some cells zero weight. It is possible to only compute new values for cells that are `NA` in the input SpatRaster.

5.6 Distance

There are a number of distance related methods. `distance` computes the shortest distance to cells that are not `NA`. `pointDistance` computes the shortest distance to any point in a set of points. `gridDistance` computes the distance when following grid cells that can be traversed (e.g. excluding water bodies). `direction` computes the direction towards (or from) the nearest cell that is not `NA`. `adjacency` determines which cells are adjacent to other cells, and `pointDistance` computes distance between points. See the `gdistance` package for more advanced distance calculations (cost distance, resistance distance)

5.7 Spatial configuration

The `clump` method identifies groups of cells that are connected. `boundaries` identifies edges, that is, transitions between cell values. `area` computes the size of each grid cell (for unprojected rasters), this may be useful to, e.g. compute the area covered by a certain class on a longitude/latitude raster.

```
r <- rast(nrow=45, ncol=90)
values(r) <- round(runif(ncell(r))*3)
a <- cellSize(r)
zonal(a, r, "sum")
##   lyr.1         area
## 1     0 8.448284e+13
## 2     1 1.718097e+14
## 3     2 1.682943e+14
## 4     3 8.547879e+13
```

5.8 Predictions

The package has two methods to make model predictions to (potentially very large) rasters. `predict` takes a multilayer raster and a fitted model as arguments. Fitted models can be of various classes, including glm, gam, randomforest, and brt. method `interpolate` is similar but is for models that use coordinates as predictor variables, for example in kriging and spline interpolation.

5.9 Vector to raster conversion

The raster packages supports point, line, and polygon to raster conversion with the `rasterize` method. For vector type data (points, lines, polygons), objects of Spatial* classes defined in the `sp` package are used; but points can also be represented by a two-column matrix (x and y).

Point to raster conversion is often done with the purpose to analyze the point data. For example to count the number of distinct species (represented by point observations) that occur in each raster cell. `rasterize` takes a `SpatRaster` object to set the spatial extent and resolution, and a function to determine how to summarize the points (or an attribute of each point) by cell.

Polygon to raster conversion is typically done to create a `SpatRaster` that can act as a mask, i.e. to set to `NA` a set of cells of a `terra` object, or to summarize values on a raster by zone. For example a country polygon is transferred to a raster that is then used to set all the cells outside that country to `NA`; whereas polygons representing administrative regions such as states can be transferred to a raster to summarize raster values by region.

It is also possible to convert the values of a `SpatRaster` to points or polygons, using `as.points` and `as.polygons`. Both methods only return values for cells that are not `NA`.

5.10 Summarize

When used with a `SpatRaster` object as first argument, normal summary statistics functions such as min, max and mean return a SpatRaster. You can use global if, instead, you want to obtain a summary for all cells of a single `SpatRaster` object. You can use `freq` to make a frequency table, or to count the number of cells with a specified value. Use `zonal` to summarize a `SpatRaster` object using zones (areas with the same integer number) defined in a `SpatRaster` and `crosstab` to cross-tabulate two `SpatRaster` objects.

```
r <- rast(ncol=36, nrow=18)
values(r) <- runif(ncell(r))
global(r, mean)
##            mean
## lyr.1 0.4961895
s <- r
values(s) <- round(runif(ncell(r)) * 5)
zonal(r, s, 'mean')
##   lyr.1   lyr.1.1
## 1     0 0.4922367
## 2     1 0.4815709
## 3     2 0.5392224
## 4     3 0.4580574
## 5     4 0.4816334
## 6     5 0.5544121
freq(s)
##   layer value count
## 1     1     0    71
## 2     1     1   135
## 3     1     2   130
## 4     1     3   125
## 5     1     4   130
## 6     1     5    57
freq(s, value=3)
##   layer value count
## 1     1     3   125
```

(continues on next page)

(continued from previous page)

```
crosstab(c(r*3, s))
##      lyr.1.1
## lyr.1  0  1  2  3  4  5
##     0 17 23 16 24 25 10
##     1 17 48 35 48 42 12
##     2 25 47 56 38 42 21
##     3 12 17 23 15 21 14
```

CHAPTER

SIX

PLOTTING

Several generic functions have been implemented for SpatRaster objects to create maps and other plot types. Use 'plot' to create a map of a SpatRaster object. When plot is used with a `SpatRaster`, it calls the function 'rasterImage' (but, by default, adds a legend; using code from fields::image.plot). It is also possible to directly call `image`. You can zoom in using 'zoom' and clicking on the map twice (to indicate where to zoom to). With `click` it is possible to interactively query a SpatRaster object by clicking once or several times on a map plot.

After plotting a `SpatRaster` you can add vector type spatial data (points, lines, polygons). You can do this with functions points, lines, polygons if you are using the basic R data structures or plot(object, add=TRUE) if you are using Spatial* objects as defined in the sp package. When plot is used with a multi-layer SpatRaster object, all layers are plotted (up to 16), unless the layers desired are indicated with an additional argument. You can also plot SpatRaster objects with `ggplot` (via the "tidyterra" package). The `rasterVis` package has several other `lattice` based plotting functions for SpatRaster objects.

Multi-layer SpatRasters can be plotted as a single plot if they channels are declared as RGB channels (red, green blue), see `?RGB`

```
library(terra)
## terra 1.7.62
b <- rast(system.file("ex/logo.tif", package="terra"))
nlyr(b)
## [1] 3
RGB(b)
## [1] 1 2 3
plot(b)
```

You can also use the a number of other plotting functions with a `terra` object as argument, including `hist`, `persp`, `contour`, and `density`. See the help files for more info.

CHAPTER

SEVEN

WRITING FILES

7.1 File format

"terra" can read and write most file formats, via the `GDAL` library. For netCDF files, use `writeCDF`

CHAPTER

EIGHT

CELL-LEVEL FUNCTIONS

8.1 Introduction

The cell number is an important concept in the raster package. Raster data can be thought of as a matrix, but in a `SpatRaster` it is more commonly treated as a vector. Cells are numbered from the upper left cell to the upper right cell and then continuing on the left side of the next row, and so on until the last cell at the lower-right side of the raster. There are several helper functions to determine the column or row number from a cell and vice versa, and to determine the cell number for x, y coordinates and vice versa.

```
library(terra)
## terra 1.7.62
r <- rast(ncol=36, nrow=18)
ncol(r)
## [1] 36
nrow(r)
## [1] 18
ncell(r)
## [1] 648
rowFromCell(r, 100)
## [1] 3
colFromCell(r, 100)
## [1] 28
cellFromRowCol(r,5,5)
## [1] 149
xyFromCell(r, 100)
##       x  y
## [1,] 95 65
cellFromXY(r, cbind(0,0))
## [1] 343
colFromX(r, 0)
## [1] 19
rowFromY(r, 0)
## [1] 10
```

8.2 Accessing cell values

Cell values can be accessed with several methods. Use `values` to get all values or a single row; and `valuesBlock` to read a block (rectangle) of cell values.

```
r <- rast(system.file("ex/meuse.tif", package="terra"))
v <- values(r)
v[708:712]
## [1] NA NA NA NA NA
```

You can also read values using cell numbers or coordinates (xy) using the `extract` method.

```
cells <- cellFromRowCol(r, 50, 35:39)
cells
## [1] 3955 3956 3957 3958 3959
r[cells]
##   meuse
## 1   743
## 2   706
## 3   646
## 4   686
## 5   758
xy <- xyFromCell(r, cells)
xy
##           x      y
## [1,] 179780 332020
## [2,] 179820 332020
## [3,] 179860 332020
## [4,] 179900 332020
## [5,] 179940 332020
extract(r, xy)
##   meuse
## 1   743
## 2   706
## 3   646
## 4   686
## 5   758
```

You can also extract values using SpatialPolygons* or SpatialLines*. The default approach for extracting raster values with polygons is that a polygon has to cover the center of a cell, for the cell to be included. However, you can use argument "weights=TRUE" in which case you get, apart from the cell values, the percentage of each cell that is covered by the polygon, so that you can apply, e.g., a "50% area covered" threshold, or compute an area-weighted average.

In the case of lines, any cell that is crossed by a line is included. For lines and points, a cell that is only 'touched' is included when it is below or to the right (or both) of the line segment/point (except for the bottom row and right-most column).

In addition, you can use standard *R* indexing to access values, or to replace values (assign new values to cells) in a `terra` object. If you replace a value in a `terra` object based on a file, the connection to that file is lost (because it now is different from that file). Setting raster values for very large files will be very slow with this approach as each time a new (temporary) file, with all the values, is written to disk. If you want to overwrite values in an existing file, you can use `update` (with caution!)

```
#r[cells]
#r[1:4]
#sources(r)
#r[2:3] <- 10
#r[1:4]
#sources(r)
```

Note that in the above examples values are retrieved using cell numbers. That is, a raster is represented as a (one-dimensional) vector. Values can also be inspected using a (two-dimensional) matrix notation. As for *R* matrices, the first index represents the row number, the second the column number.

```
#r[1]
#r[2,2]
#r[1,]
#r[,2]
#r[1:3,1:3]

# keep the matrix structure
#r[1:3,1:3, drop=FALSE]
```

Accessing values through this type of indexing should be avoided inside functions as it is less efficient than accessing values via functions like `values`.

CHAPTER

NINE

SPATIAL PREDICTION

This chapters shows some examples for making spatial prediction with different types of models. Using the `predict` and `interpolate` methods.

The is the data we use.

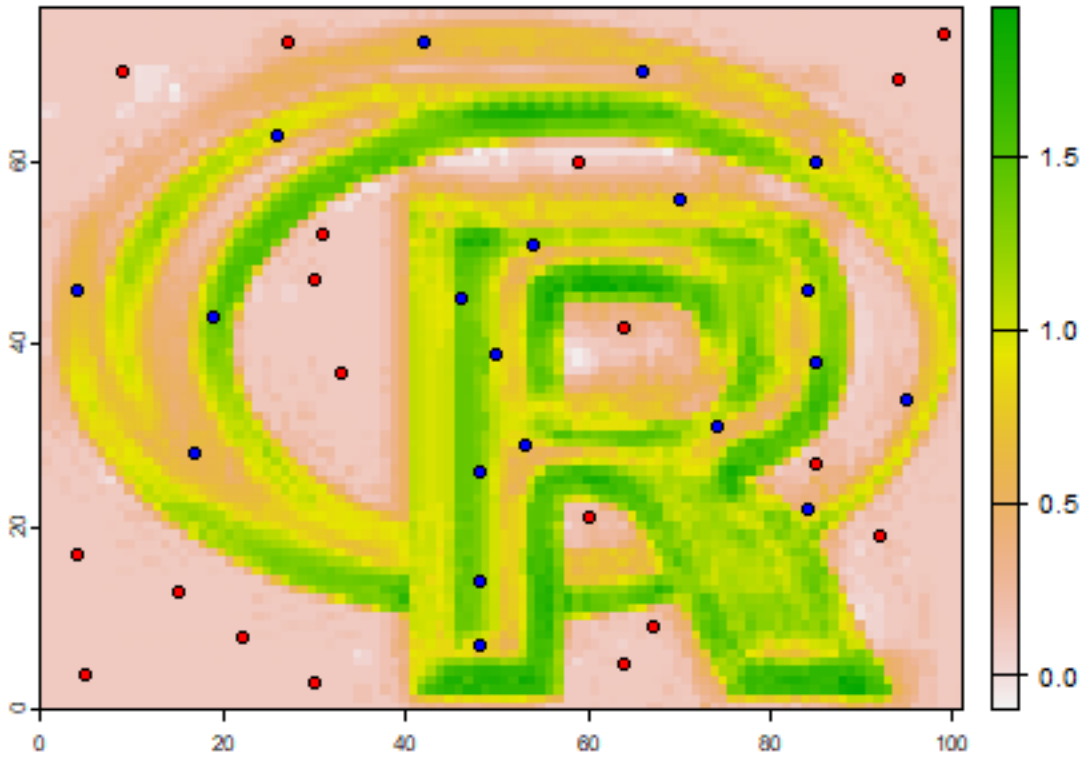
```
library(terra)
logo <- rast(system.file("ex/logo.tif", package="terra"))
names(logo) <- c("red", "green", "blue")
p <- matrix(c(48, 48, 48, 53, 50, 46, 54, 70, 84, 85, 74, 84, 95, 85,
   66, 42, 26, 4, 19, 17, 7, 14, 26, 29, 39, 45, 51, 56, 46, 38, 31,
   22, 34, 60, 70, 73, 63, 46, 43, 28), ncol=2)

a <- matrix(c(22, 33, 64, 85, 92, 94, 59, 27, 30, 64, 60, 33, 31, 9,
   99, 67, 15, 5, 4, 30, 8, 37, 42, 27, 19, 69, 60, 73, 3, 5, 21,
   37, 52, 70, 74, 9, 13, 4, 17, 47), ncol=2)

xy <- rbind(cbind(1, p), cbind(0, a))

# extract predictor values for points
e <- extract(logo, xy[,2:3])

# combine with response
v <- data.frame(cbind(pa=xy[,1], e))
```

9.1 Predict

9.1.1 GLM

A general linear model (GLM)

```
#build a model, here with glm
model <- glm(formula=pa~., data=v)

#predict to a raster
r1 <- predict(logo, model)

plot(r1)
points(p, bg='blue', pch=21)
points(a, bg='red', pch=21)
```

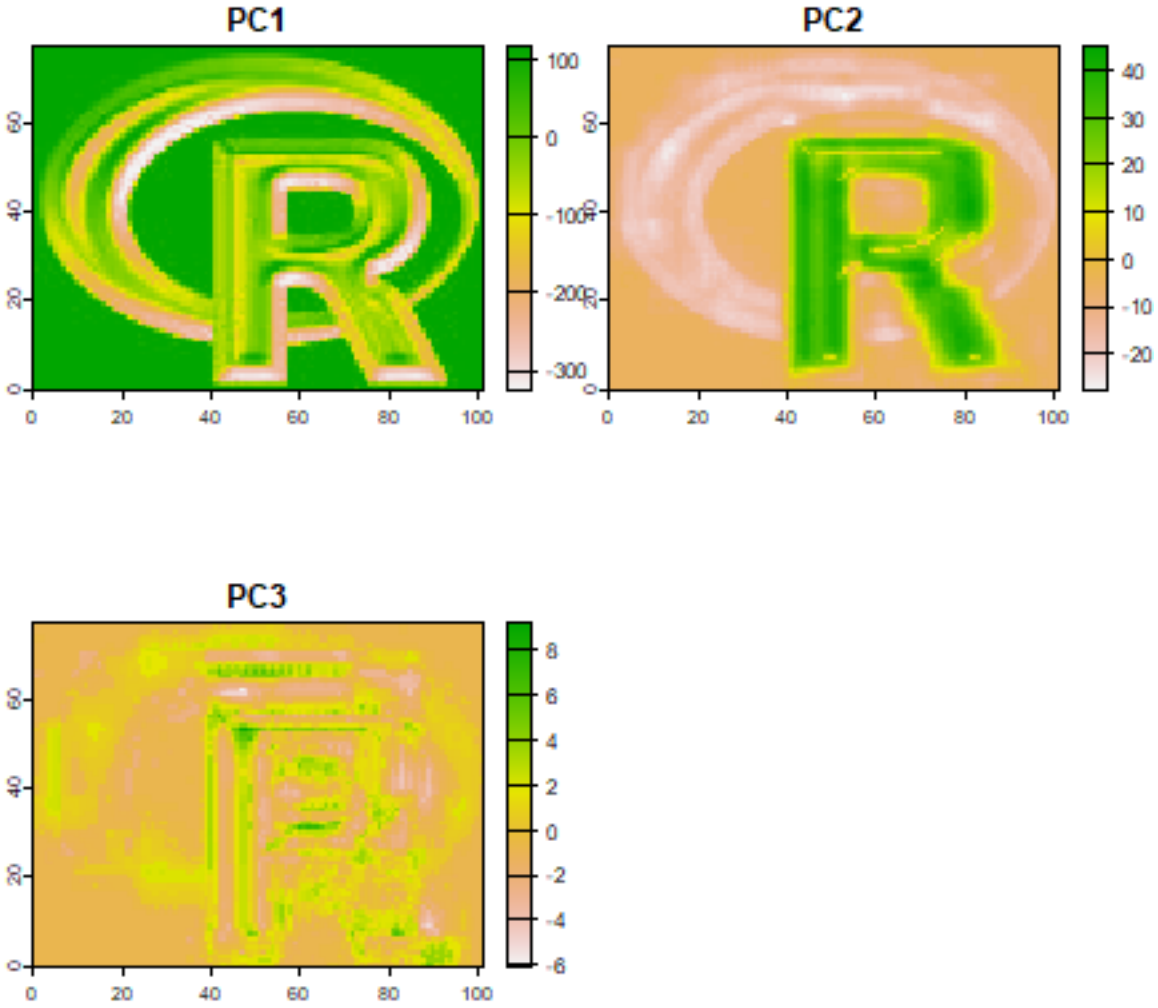



```
# logistic regression
model <- glm(formula=pa~., data=v, family="binomial")
## Warning: glm.fit: algorithm did not converge
## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
r1log <- predict(logo, model, type="response")

# use a modified function to get the probability and standard error
# from the glm model. The values returned by "predict" are in a list,
# and this list needs to be transformed to a matrix

predfun <- function(model, data) {
  v <- predict(model, data, se.fit=TRUE)
  cbind(p=as.vector(v$fit), se=as.vector(v$se.fit))
}

r2 <- predict(logo, model, fun=predfun)
```

9.1.2 Principal components

Here using sampling to simulate an object too large to feed all its values to prcomp

```
sr <- values(spatSample(logo, 100, as.raster=TRUE))
pca <- prcomp(sr)

x <- predict(logo, pca)
plot(x)
```




```
library(pls)
##
## Attaching package: 'pls'
## The following object is masked from 'package:stats':
##
##     loadings
```

(continues on next page)

(continued from previous page)

```
model <- plsr(formula=pa~., data=v)
# this returns an array:
predict(model, v[1:5,])
## , , 1 comps
##
##          pa
## 1 0.4918092
## 2 0.7463392
## 3 0.7774598
## 4 0.3499635
## 5 0.6490389
##
## , , 2 comps
##
##          pa
## 1 0.6875604
## 2 1.0224901
## 3 1.0499689
## 4 0.5055191
## 5 0.9808813
##
## , , 3 comps
##
##          pa
## 1 0.8172886
## 2 1.1435330
## 3 1.1717481
## 4 0.4949081
## 5 0.8458842
# write a function to turn that into a matrix
pfun <- function(x, data) {
   y <- predict(x, data)
   d <- dim(y)
   dim(y) <- c(prod(d[1:2]), d[3])
   y
}

pp <- predict(logo, model, fun=pfun)
```

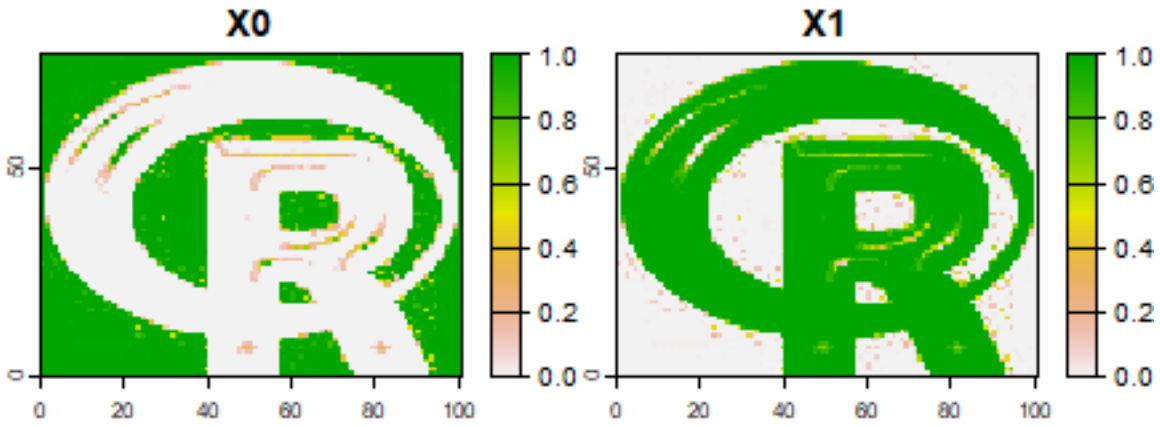
9.1.3 Random Forest

```
library(randomForest)
## randomForest 4.7-1.1
## Type rfNews() to see new features/changes/bug fixes.
rfmod <- randomForest(pa ~., data=v)
## Warning in randomForest.default(m, y, ...): The response has five or fewer
## unique values.  Are you sure you want to do regression?

## note the additional argument "type='response'" that is
## passed to predict.randomForest
```

(continues on next page)

(continued from previous page)

```
r3 <- predict(logo, rfmod, type='response')

## get class membership probabilities
vv <- v
vv$pa <- as.factor(vv$pa)
rfmod2 <- randomForest(pa ~., data=vv)
r4 <- predict(logo, rfmod2, type='prob')
plot(r4, range=c(0,1))
```



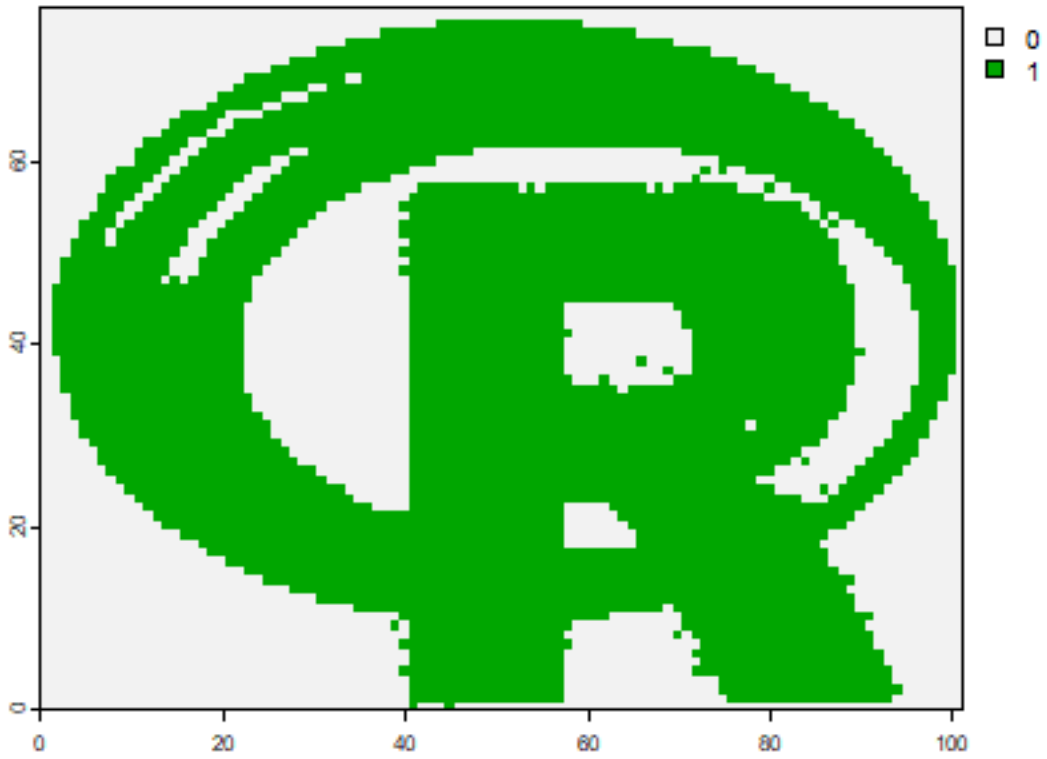
9.2 cforest

cforest is an alternative Random Forest implementation. Here an example with a `factors` argument

```
library(party)
## Loading required package: grid
##
## Attaching package: 'grid'
## The following object is masked from 'package:terra':
##
##     depth
## Loading required package: mvtnorm
## Loading required package: modeltools
## Loading required package: stats4
## Loading required package: strucchange
## Loading required package: zoo
##
## Attaching package: 'zoo'
## The following object is masked from 'package:terra':
##
##     time<-
## The following objects are masked from 'package:base':
##
##     as.Date, as.Date.numeric
## Loading required package: sandwich
m <- cforest(pa~., control=cforest_unbiased(mtry=3), data=v)
# the second argument in party:::predict.RandomForest
# is "OOB", and not "newdata" or similar. We need to write a wrapper
# predict function to deal with this
predfun <- function(m, d, ...) predict(m, newdata=d, ...)

pc <- predict(logo, m, OOB=TRUE, fun=predfun)
```

With a knn model, we can use "app" instead of "predict"

```
library(class)
cl <- factor(c(rep(1, nrow(p)), rep(0, nrow(a))))
train <- extract(logo, rbind(p, a))
k <- app(logo, function(x) as.integer(as.character(knn(train, x, cl))))
plot(k)
```



9.3 Interpolate

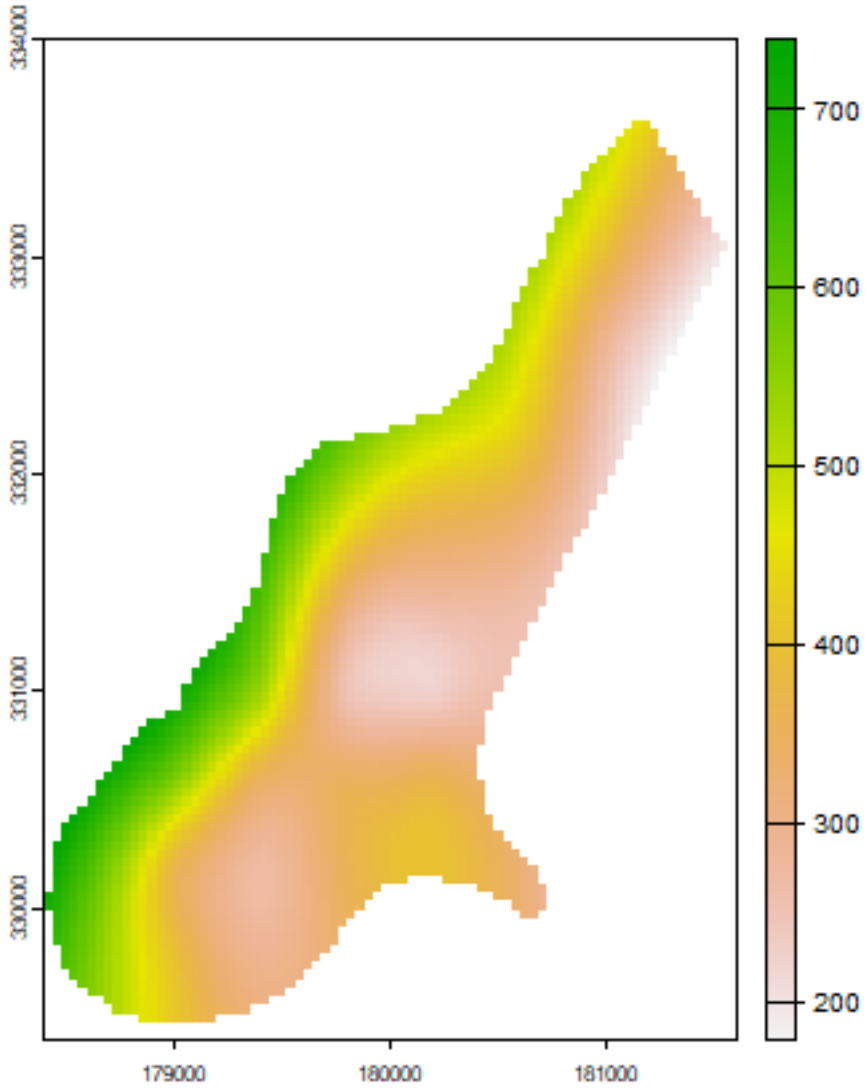
9.3.1 Thin plate spline interpolation with x and y only

```
library(terra)
# example data
r <-rast(system.file("ex/meuse.tif", package="terra"))
ra <- aggregate(r, 10)
xy <- data.frame(xyFromCell(ra, 1:ncell(ra)))
v <- values(ra)

# Thin plate spline model
library(fields)
tps <- Tps(xy, v)
```

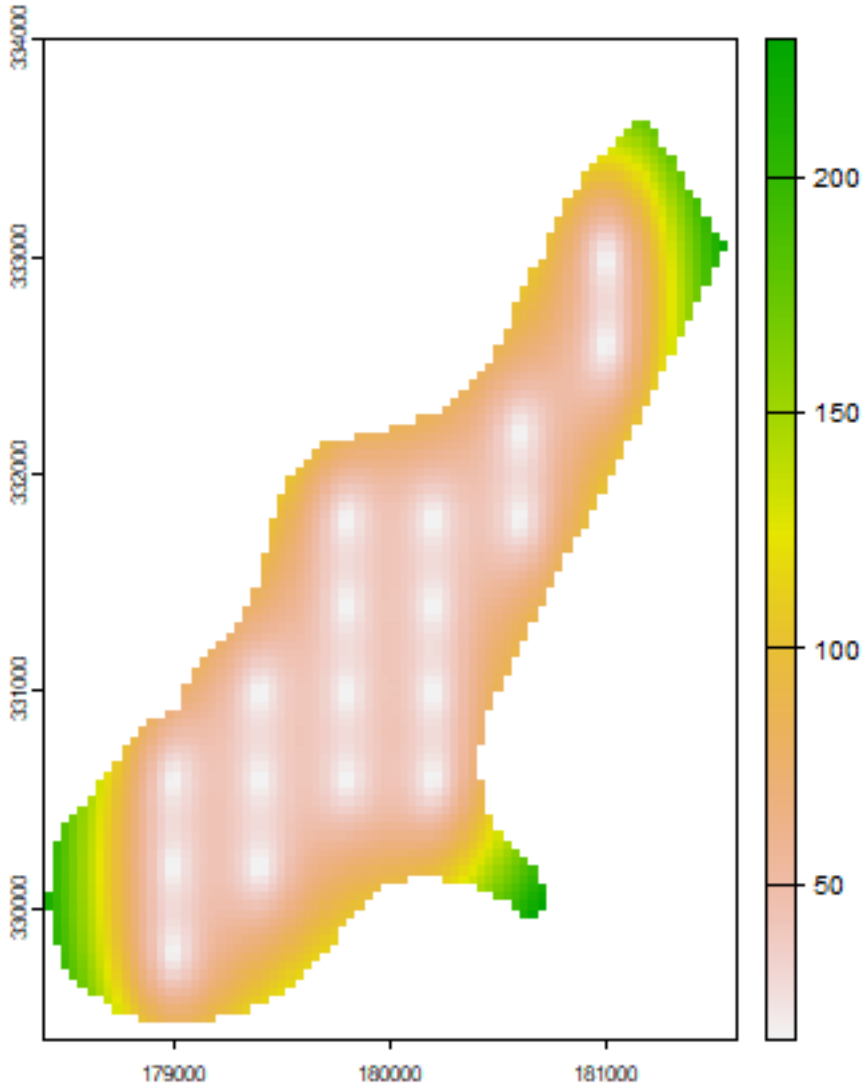
(continues on next page)

(continued from previous page)

```
## Warning:
## Grid searches over lambda (nugget and sill variances) with  minima at the endpoints:
##   (GCV) Generalized Cross-Validation
##    minimum at  right endpoint  lambda  =  6.369487e-05 (eff. df= 17.10001 )
x <- rast(r)

# use model to predict values at all locations
p <- interpolate(x, tps)
p <- mask(p, r)
plot(p)
```




```
# change the fun from predict to fields::predictSE to get the TPS standard error
se <- interpolate(x, tps, fun=predictSE)
se <- mask(se, r)
plot(se)
```



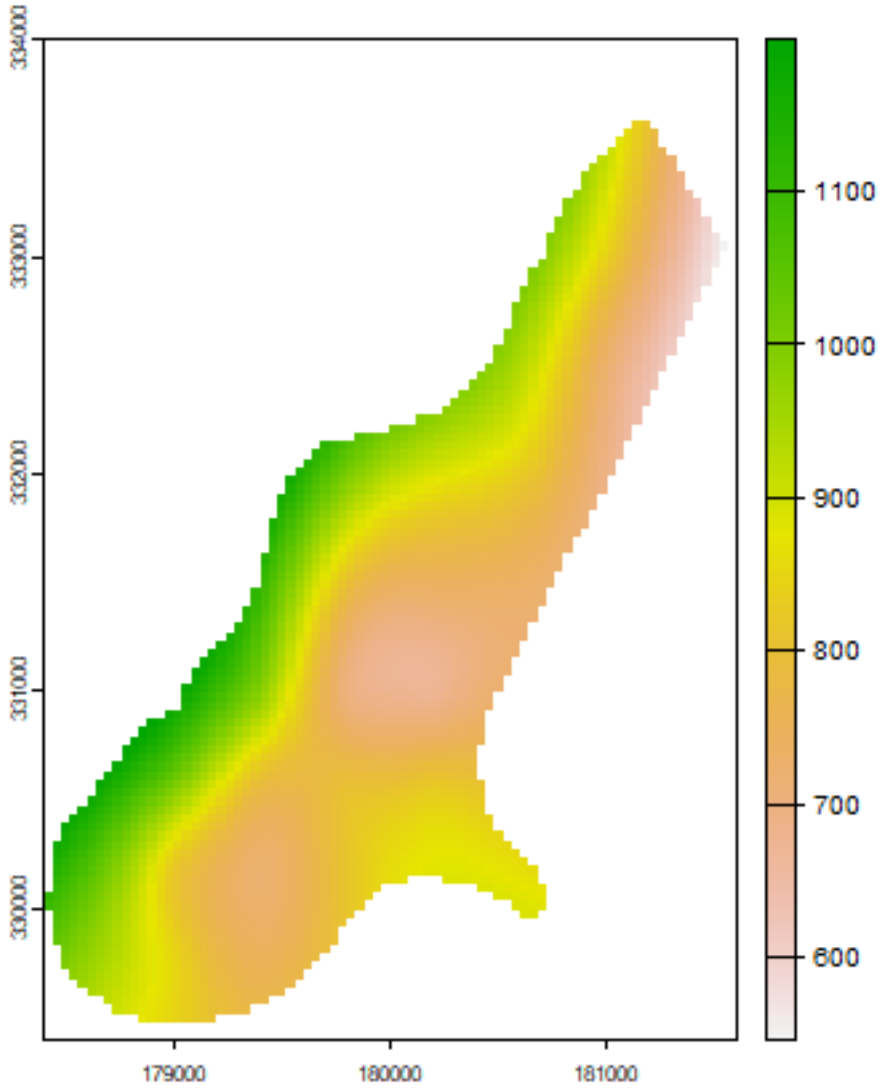
Add another predictor variable; let"s call it elevation

```
elevation <- (init(r, "x") * init(r, "y")) / 100000000
names(elevation) <- "elev"
elevation <- mask(elevation, r)

z <- extract(elevation, vect(xy, c("x", "y")), fun=function(x)x[1])
z <- z[,2,drop=FALSE]

# add as another independent variable
vv <- na.omit(cbind(xy, z, v))
tps2 <- Tps(vv[,1:3], vv[,4])
## Warning:
## Grid searches over lambda (nugget and sill variances) with  minima at the endpoints:
##   (GCV) Generalized Cross-Validation
##    minimum at  right endpoint  lambda  =  0.0003764795 (eff. df= 17.10001 )
#p2 <- interpolate(elevation, tps2)
```

(continues on next page)

(continued from previous page)

```
#plot(p2)

# as a linear coveriate
tps3 <- Tps(vv[,1:2], vv[,4], Z=vv[,3])
## Warning:
## Grid searches over lambda (nugget and sill variances) with  minima at the endpoints:
##   (GCV) Generalized Cross-Validation
##    minimum at  right endpoint  lambda  =  6.464444e-05 (eff. df= 17.10047 )

# Z is a separate argument in Krig.predict, so we need a new function
# Internally (in interpolate) a matrix is formed of x, y, and elev (Z)

pfun <- function(model, x, ...) {
   predict(model, x[,1:2], Z=x[,3], ...)
}
p3 <- interpolate(elevation, tps3, fun=pfun)
plot(p3)
```



9.3.2 inverse distance weighted (IDW)

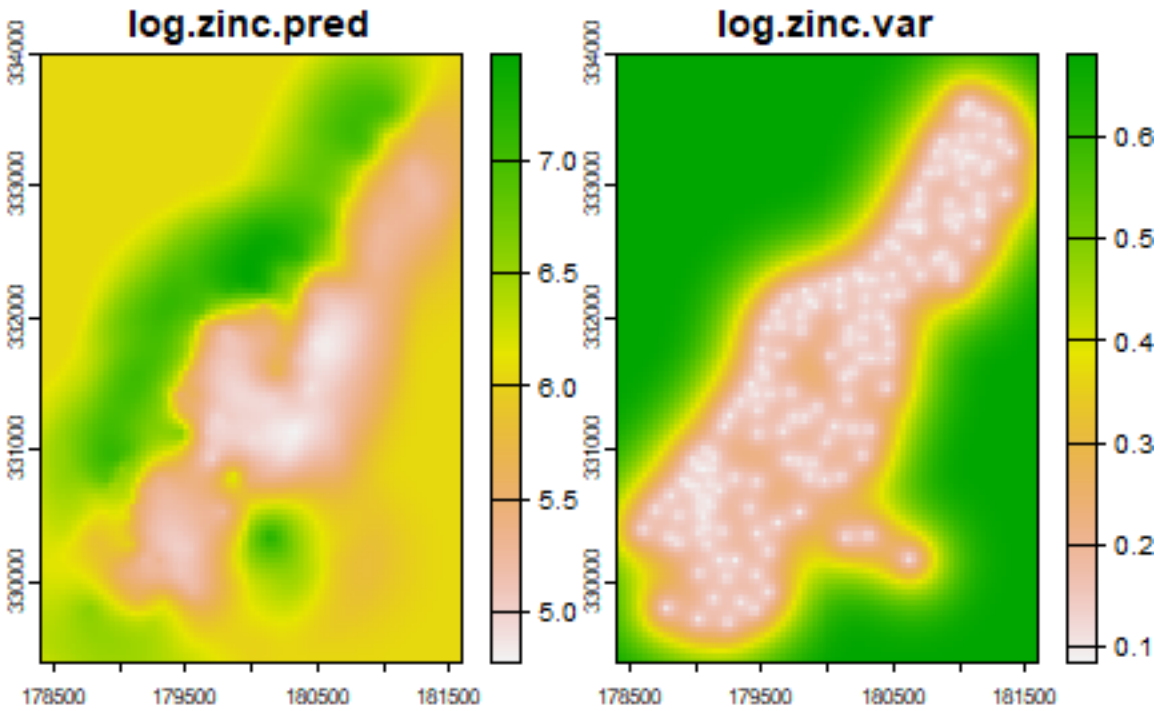
```
library(gstat)
data(meuse, package="sp")

r <- rast(system.file("ex/meuse.tif", package="terra"))
mg <- gstat(id = "zinc", formula = zinc~1, locations = ~x+y, data=meuse,
            nmax=7, set=list(idp = .5))
z <- interpolate(r, mg, debug.level=0)
```

9.3.3 Kriging

Kriging with gstat examples. Examples provided by Maurizio Marchi

orinary kriging

```
v <- variogram(log(zinc)~1, ~x+y, data=meuse)
mv <- fit.variogram(v, vgm(1, "Sph", 300, 1))
gOK <- gstat(NULL, "log.zinc", log(zinc)~1, meuse, locations=~x+y, model=mv)
OK <- interpolate(r, gOK, debug.level=0)
plot(OK)
```



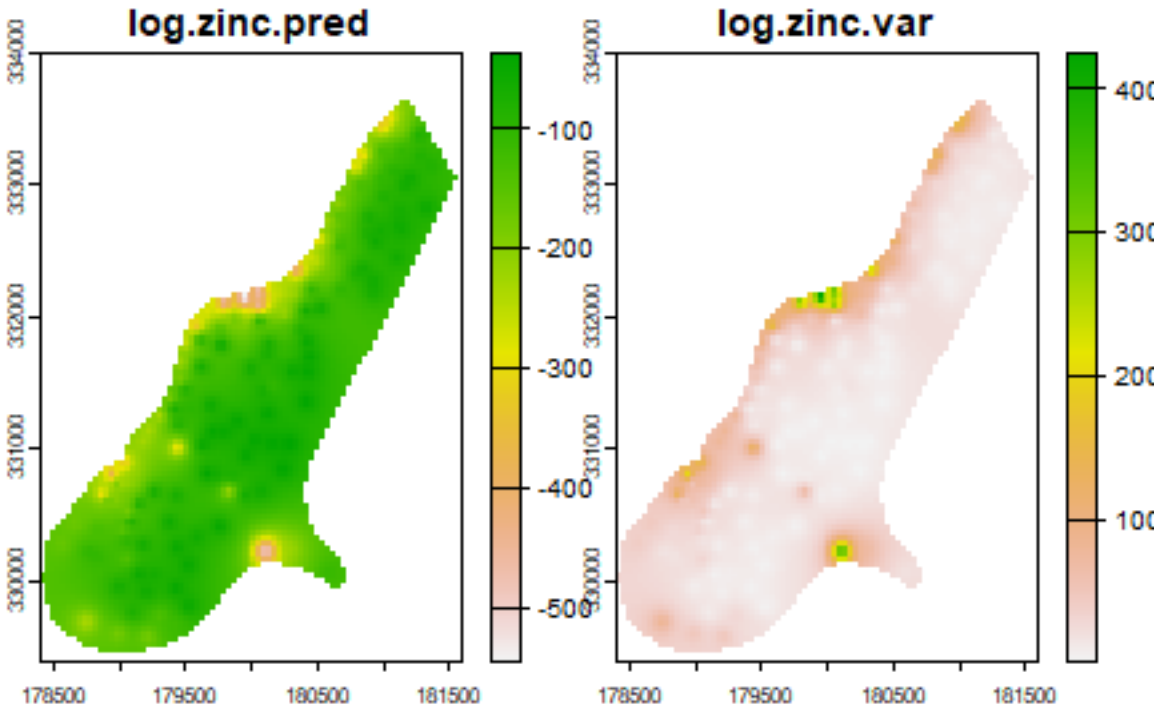
universial kriging

```
vu <- variogram(log(zinc)~elev, ~x+y, data=meuse)
mu <- fit.variogram(vu, vgm(1, "Sph", 300, 1))
gUK <- gstat(NULL, "log.zinc", log(zinc)~elev, meuse, locations=~x+y, model=mu)
```

(continues on next page)

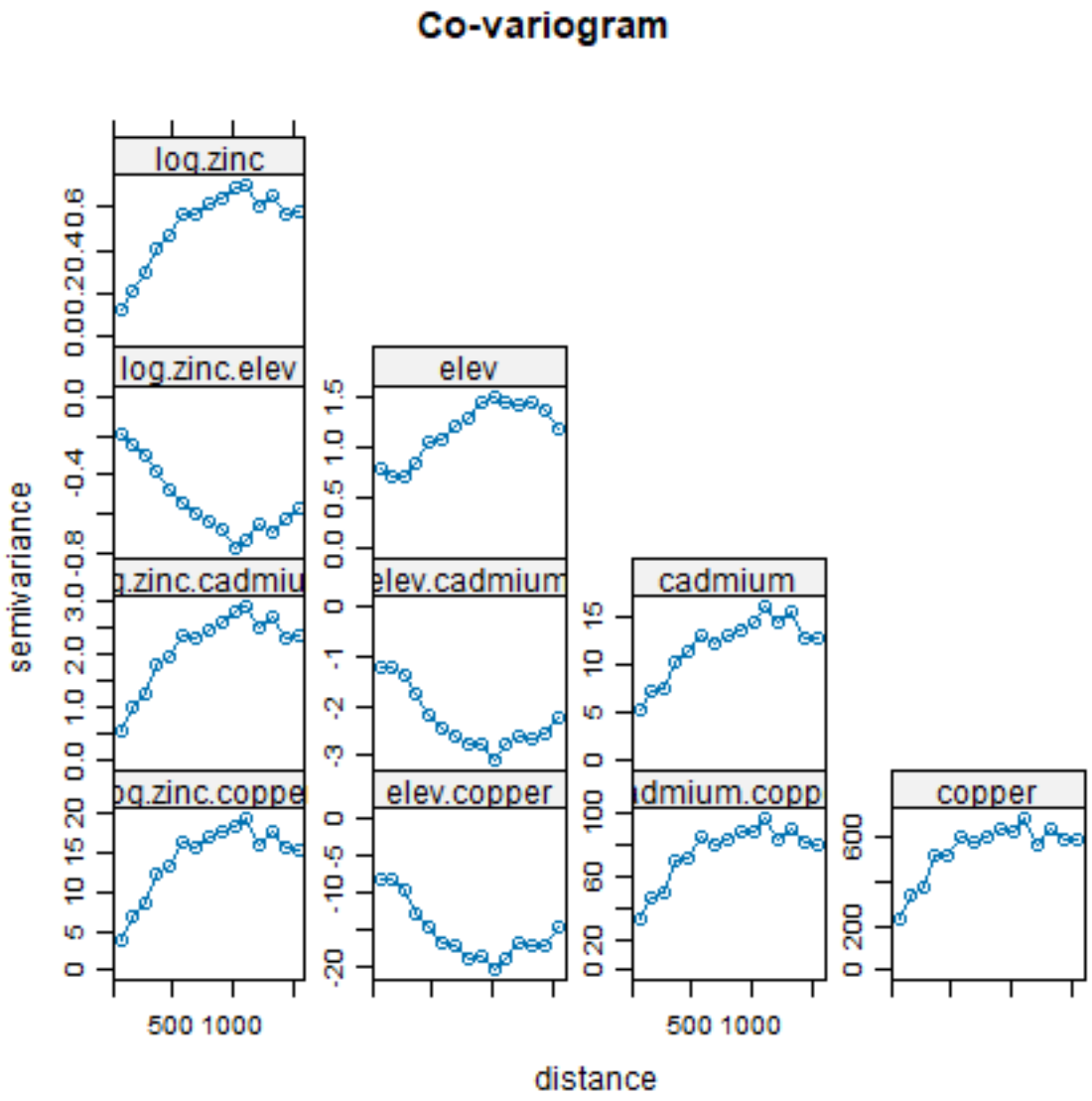
(continued from previous page)

```
names(r) <- "elev"
UK <- interpolate(r, gUK, debug.level=0)
plot(UK)
```



co-kriging

```
gCoK <- gstat(NULL, 'log.zinc', log(zinc)~1, meuse, locations=~x+y)
gCoK <- gstat(gCoK, 'elev', elev~1, meuse, locations=~x+y)
gCoK <- gstat(gCoK, 'cadmium', cadmium~1, meuse, locations=~x+y)
gCoK <- gstat(gCoK, 'copper', copper~1, meuse, locations=~x+y)
coV <- variogram(gCoK)
plot(coV, type='b', main='Co-variogram')
```

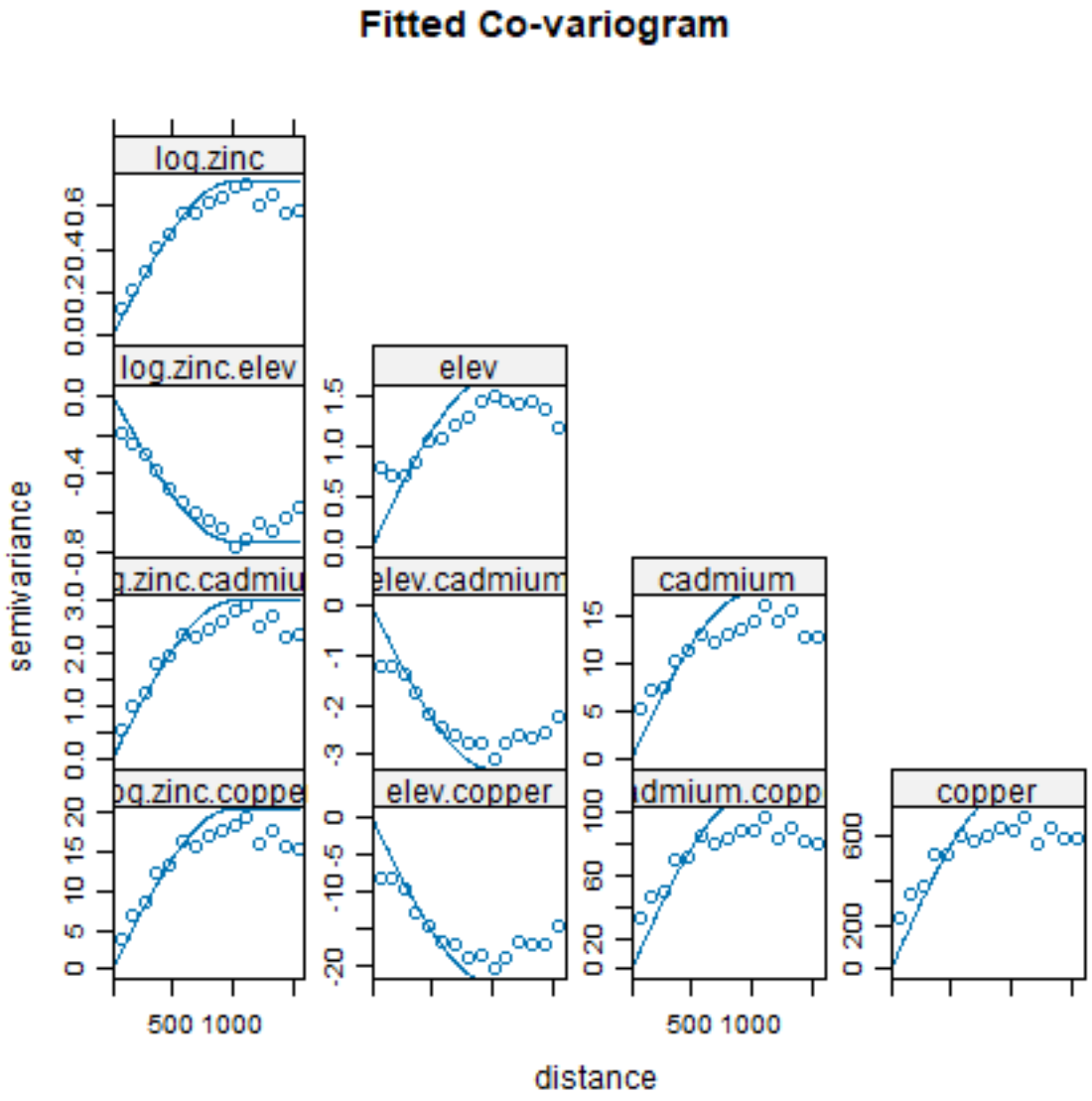



```
coV.fit <- fit.lmc(coV, gCoK, vgm(model='Sph', range=1000))
coV.fit
## data:
## log.zinc : formula = log(zinc)`~`1 ; data dim = 155 x 12
## elev : formula = elev`~`1 ; data dim = 155 x 12
## cadmium : formula = cadmium`~`1 ; data dim = 155 x 12
## copper : formula = copper`~`1 ; data dim = 155 x 12
## variograms:
##                   model       psill range
## log.zinc            Sph   0.7132435  1000
## elev                Sph   1.6908552  1000
## cadmium             Sph  17.4957356  1000
## copper              Sph 809.4027563  1000
## log.zinc.elev       Sph  -0.7404289  1000
## log.zinc.cadmium    Sph   2.9802854  1000
## elev.cadmium        Sph  -3.2983554  1000
```

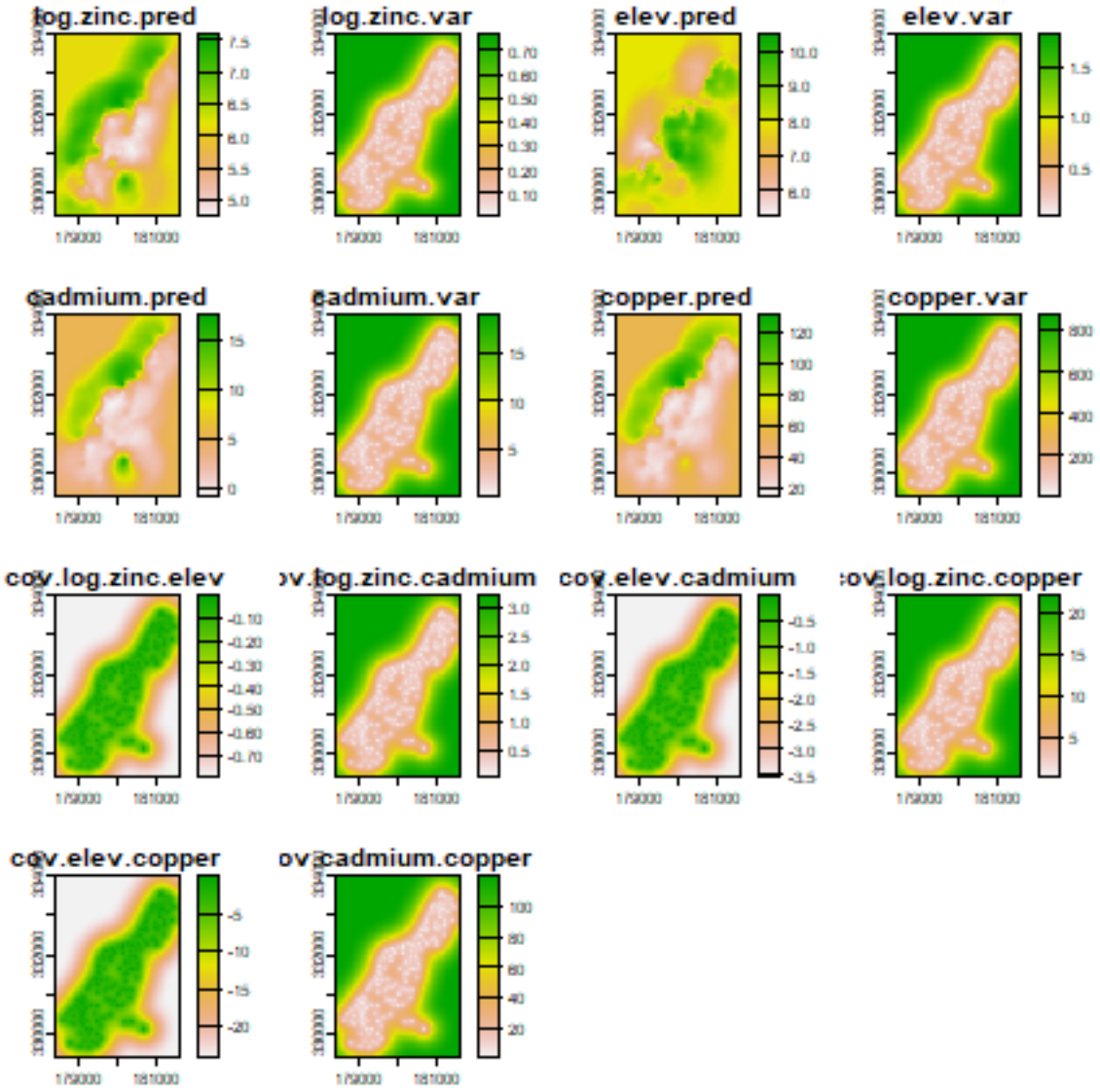
(continues on next page)

(continued from previous page)

```
## log.zinc.copper     Sph  20.4199742  1000
## elev.copper         Sph -22.4955673  1000
## cadmium.copper      Sph 111.1393673  1000
## ~x + y
## <environment: 0x000001c174b59208>
plot(coV, coV.fit, main='Fitted Co-variogram')
```




```
coK <- interpolate(r, coV.fit, debug.level=0)
plot(coK)
```


log.zinc.pred
log.zinc.var
elev.pred
elev.var
cadmium.pred
cadmium.var
copper.pred
copper.var
cov.log.zinc.elev
ov.log.zinc.cadmium
cov.elev.cadmium
ov.log.zinc.copper
cov.elev.copper
ov.cadmium.copper

CHAPTER

TEN

MISCELLANEOUS

10.1 Session options

There is a number of session options that influence reading and writing files. These can be set in a session, with `terraOptions`, and saved to make them persistent in between sessions. But you probably should not change the default values unless you have pressing need to do so. You can, for example, set the directory where temporary files are written, and set your preferred default file format and data type. Some of these settings can be overwritten by arguments to functions where they apply (with arguments like filename, datatype, format). Except for generic functions like mean, '+', and sqrt. These functions may write a file when the result is too large to hold in memory and then these options can only be set through the session options. The options chunksize and maxmemory determine the maximum size (in number of cells) of a single chunk of values that is read/written in chunk-by-chunk processing of very large files.

```
library(terra)
## terra 1.7.62
terraOptions()
## memfrac   : 0.6
## tolerance : 0.1
## verbose   : FALSE
## todisk    : FALSE
## tempdir   : C:/temp/RtmpmC1Xr6
## datatype  : FLT4S
## memmin    : 1
## progress  : 3
```