Introduction to the 'raster' package (Version 1.5-16)

Robert J. Hijmans

October 10, 2010

1 Introduction

This vignette describes the R package '`raster`'. A raster is a spatial (geographic) data structure that divides an area into rectangles. Such a data structure is also referred to as a 'grid' and is often contrasted with 'vector' data (which represents points, lines, and polygons). The `raster` package has functions for reading, manipulating, and writing raster data. The package provides (1) general low-level raster data manipulation functions (e.g. read raster values by row), convert cell numbers to coordinates and back, that can easily be used to develop higher level specific functions; (2) 'high level' functions for raster data manipulation that are common in other spatial data analysis software (often referred to as 'GIS'); (3) to provide a raster algebra implementation.

A notable feature of the pakcage is that its functions can work with very large raster datasets that are stored on disk and cannot be loaded into memory. The raster package can work with extremely large files because it does not load all the contents of these files into memory when a Raster type object is created from a file. Instead, the object gathers some basic information, such as the number of rows and columns, and the spatial extent. In computations with a Raster object, data then will be read and processes in chunks. If no output filename specified, and the output raster is too large to keep in memory, the results are written to a temporary file.

The package is built around a number of 'S4' classes of which the RasterLayer, RasterBrick, and RasterStack classes are the most important.

2 Classes

This package is built around a number of 'S4' classes. The three most important classes are: RasterLayer, RasterStack and RasterBrick.

2.1 RasterLayer

Most functions in this pacakge operate on objects of the RasterLayer class. A RasterLayer describes a single-variable raster dataset. A RasterLayer object always has values for a number of fundamental parameters such as the number of columns and rows, the coordinates of its spatial extent ('bounding box'), the coordinate reference system (the 'map projection', although this can be NA). In addition, a RasterLayer can store information about the filename where the raster cell values are stored (if there is such a file), and it can store some or all of the raster cell values.

Because a RasterLayer is a single-variable dataset it can easily be used for raster algebra (arithmetic and other mathematical operations). Also, some raster file systems only allow a single variable per file and this class matches that system. There are, however, also many cases where multi-variable raster data sets are more useful. The `raster` package has two classes for that, the `RasterStack` and the `RasterBrick`

2.2 RasterStack

A RasterStack is a collection of RasterLayer objects with the same spatial extent and resolution. In essence it is a like a list of individual RasterLayer objects. A RasterStack can easily be formed form a collection of files in different locations and mixed with RasterLayer objects that only exist in memory.

There are a number of methods (functions) available for RasterStack objects. These include `calc`, which lets you compute summary statistics (e.g., `sum` or `mean`) for cells across all layers.

2.3 RasterBrick

A RasterBrick is truly multilayered RasterLayer. It is more efficient than a RasterStack, but it can only refer to a single file. A typical example of such a file would be a multi-band satellite image.

2.4 Other classes

The three classes described above inherit from the Raster class which inherits from the BasicRaster class. The BasicRaster only has a few properties ('slots' in S4 speak): the number of columns and rows, the coordinate reference system (which itself is an object of class CRS, which is defined in package 'sp') and the spatial extent (which is an object of class Extent). Raster is a virtual class. This means that it cannot be instantiated (you cannot create objects from this class). It is used so that methods can be defined for that class. These methods will be dispatched when called with a descendent of this class (i.e. when the method is called with a RasterLayer, RasterBrick or RasterStack object as argument). This allows for efficient code writing and documentation. RasterStackBrick is a class union of the RasterStack and RasterBrick class. This is a also a

virtual class. It allows defining methods that apply to both a RasterStack and to a RasterBrick. Like with the Raster class its purpose is efficiency in code and documentation. Class 'Extent' has four slots for the extreme x and y (or longitude and latitude) coordinates of the raster.

3 Creating Raster objects

A RasterLayer can easily be created from scratch using the function **raster**. 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 nrow, ncol, to the function. You can also change these parameters after creating the object. In some cases, for example when you change the number of columns or rows, you will loose the values associated with the RasterLayer (or the link to a file if there was one). If you set the projection, this is only to properly define it, not to change it. To tranform a RasterLayer to another coordinate reference system (projection) you can can use the function **projectRaster**.

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

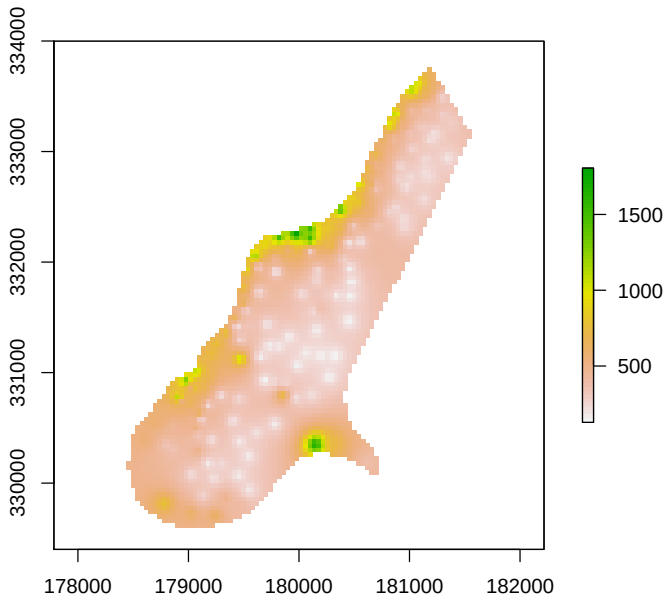
```
> library(raster)

raster version 1.5-16 (8-October-2010)

> r <- raster()
> r

class       : RasterLayer
filename    :
nrow        : 180
ncol        : 360
ncell       : 64800
min value   :
max value   :
projection  : +proj=longlat +datum=WGS84
xmin        : -180
xmax        : 180
ymin        : -90
ymax        : 90
xres        : 1
yres        : 1

> r <- raster(ncol=36, nrow=18, xmn=-1000, xmx=1000, ymn=-100, ymx=900)
> res(r)

[1] 55.55556 55.55556
```

```
> res(r) <- 100
> res(r)

[1] 100 100

> ncol(r) <- 18
> res(r)

[1] 111.1111 100.0000

> projection(r) <- "+proj=utm +zone=48 +datum=WGS84"
> r

class       : RasterLayer
filename    :
nrow        : 10
ncol        : 18
ncell       : 180
min value   :
max value   :
projection  : +proj=utm +zone=48 +datum=WGS84
xmin        : -1000
xmax        : 1000
ymin        : -100
ymax        : 900
xres        : 111.1111
yres        : 100
```

The function **raster** also allows you to create a RasterLayer from another object, including another RasterLayer, RasterStack and RasterBrick, as well as from a SpatialPixels* and SpatialGrid* object (defined in the sp package), an extent object, and from a matrix.

It is more common, however, to create RasterLayer objects from files. The raster package can read raster files in several formats, including some 'natively' supported formats and other formats via the rgdal package. Supported formats for reading include ESRI, ENVI, and ERDAS grids and geoTiff. Most formats supported for reading can also be written too. Here is an example using the 'Meuse' dataset (taken from the sp package), using a file in the native 'rasterfile' format:

```
> r <- raster(system.file("external/test.grd", package="raster"))
> plot(r)
```



Multi-layer objects can be created in memory (from RasterLayer objects) or from file(s).

```
> r1 <- raster(nrow=10, ncol=10)
> # set random cell values
> r1 <- setValues(r1, runif(ncell(r1)))
> r2 <- setValues(r1, runif(ncell(r1)))
> r3 <- setValues(r1, runif(ncell(r1)))
> # combine three RasterLayer objects into a RasterStack
> s <- stack(r1, r2, r3)
> s

class       : RasterStack
filename    :
nlayers     : 3
nrow        : 10
ncol        : 10
ncell       : 100
projection  : +proj=longlat +datum=WGS84
min value   : 0.0012 0.0071 0.0077
max value   : 0.98 1.00 1.00
xmin        : -180
xmax        : 180
```

```
ymin        : -90
ymax        : 90
xres        : 36
yres        : 18
```

```
> nlayers(s)
```

```
[1] 3
```

```
> # combine three RasterLayer objects into a RasterBrick
> b1 <- brick(r1, r2, r3)
> b2 <- brick(s)
> # create a RasterBrick from file
> b <- brick(system.file("external/rlogo.grd", package="raster"))
> b
```

```
class       : RasterBrick
filename    : /tmp/Rinst1198023405/raster/external/rlogo.grd
nlayers     : 3
nrow        : 77
ncol        : 101
ncell       : 7777
projection  : '+proj=utm +zone=1 +ellps=WGS84'
min value   : 0 0 0
max value   : 255 255 255
xmin        : 0
xmax        : 101
ymin        : 0
ymax        : 77
xres        : 1
yres        : 1
```

```
> nlayers(b)
```

```
[1] 3
```

```
> # extract a single RasterLayer
> r <- raster(b, layer=2)
> # equivalent to creating it from disk
> r <- raster(system.file("external/rlogo.grd", package="raster"), band=2)
```

4 Raster algebra

Many generic functions have been implemented for RasterLayer objects (but not for mutli-layer objects), including the normal algebraic and logical operators and functions such as **abs, round, ceiling, floor, trunc, sqrt, log, log10, exp, cos, sin, max, min, range, prod, sum, any, all**. These functions

allows for simple and elegant raster algebra. In these functions you can mix RasterLayer objects with numbers, as long as the first argument is a RasterLayer. Summary functions **(min, max, mean, prod, sum, Median, cv, range, any, all)** can also be used with a RasterStack or RasterBrick as argument.

In raster algebra, the result of a computation is always a RasterLayer. This is probably obvious when mutliplying two RasterLayer objects, but perhaps this is not obvious when using functions like min, sum or mean. Use cellStats if instead of a RasterLayer you want a single number summarizaing the cell values of a single RasterLayer.

```
> r <- raster(ncol=36, nrow=18)
> r[] <- 1:ncell(r)
> s <- r + 1
> s <- sqrt(s)
> s <- s * r + 5
> r[] <- round(runif(ncell(r)))
> r <- r == 1
> s[r] <- -0.5
> s[!r] <- 5
> s[s == 5] <- runif(length(s[s==5]))
> a <- sum(r,s)
> b <- round(mean(r,s,10))
> st <- stack(r, s, a, b)
> sst <- sum(st)
```

5 'High-level' functions

Several 'high level' functions have been implemented for RasterLayer objects. With 'high level' functions we refer to those functions that you would normally find in a GIS program that supports raster data (such as IDRISI, GRASS, or GRID module in ArcInfo workstation). All these functions work for raster datasets that cannot be loaded into memory. Here we briefly discuss some of these functions. See the help files for more detailed descriptions of each function.

The high-level functions have some arguments in common. The first argument is typically 'x' or 'object' and can be a RasterLayer, and in some cases a RasterStack or RasterBrick. It is followed by one or more arguments specific to the function (either additional RasterLayer objects or parameters), 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 function is to return a RasterLayer that only exists in memory. However, if the function will create a RasterLayer that is 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, and a logical value indicating whether existing files should be overwritten.

5.1 Structural modification

There are several functions that deal with modifying the structure of RasterLayer objects. **aggregate** and **disaggregate** allow for changing the resolution of a RasterLayer. 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 **disaggregate** are the best functions when adjusting cells size only, and with an integer fraction (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** function might be used. It can do either nearest neighbor assignments (for categorical data) or bilinear interpolation (for non-categorical data). Simple linear shifts of a Raster object can be accomplished with the **shift** function or with the **extent** function.

The **crop** function lets you take a geographic subset of a larger RasterLayer. You can crop a RasterLayer 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 the larger RasterLayer and then use **drawExtent()** to visually determine the new extent (bounding box) to provide to the crop function.

trim crops a RasterLayer by removing the outer rows and columns that only contain NA values. In contrast, **expand** adds new rows and/or columns with NA values. The purpose of this could be to create a new RasterLayer of the same extent of another larger RasterLayer such that the can be used in raster algebra.

The **merge** function lets you merge 2 or more RasterLayer objects into a single new object. The input objects must have the same resolution and origin (that is their cells neatly fit in a single larger RasterLayer).

With the **projectRaster** function you can transform values of RasterLayer to a new coordinate reference system.

```
> r <- raster()
> r[] <- 1:ncell(r)
> ra <- aggregate(r, 10)
> r1 <- crop(r, extent(-180,0,0,30))
> r2 <- crop(r, extent(-10,180,-20,10))
> m <- merge(r1, r2, filename='test.grd', overwrite=TRUE)
```

flip 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 to the standard -180 to 180 degrees system.

5.2 Overlay

As an alternative to the raster algebra discussed above, the following 'high-level' functions are available to accomplish the same things: **overlay**, **calc**, **reclass**, **subs**, **cover** and **mask**. These provide either easy to use short-hand, or more efficient computation (for disk based RasterLayers). **calc** allows you to do a computation for a single RasterLayer whereas with **overlay** you can combine multiple layers. Use **reclass** to replace ranges of values with single values, or **subs** to substitute (replace) single values with other values. Function **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.

```
> r <- raster(ncol=10, nrow=10)
> r[] <- round(runif(ncell(r))*10)
> s <- calc(r, function(x){ x[x < 2] <- NA; return(x)} )
> t <- overlay(r, s, fun=function(x, y){ x / (2 * sqrt(y)) + 5 } )
> u <- mask(r, t)
> v = u==s
> w <- cover(t, r)
> x <- reclass(w, c(0,1,1,  1,5,2, 4,10,3))
> y <- subs(w, data.frame(id=c(0,2), v=c(40,50)))
```

5.3 Focal functions

There are three focal (neighorhood) functions: **focal**, **focalFilter**, **focalNA**. These functions make a computation using values in a neighborhood of cells around a focal cell, and putting the result in the focal cell of the output RasterLayer. With **focal**, the neighborhood can only be a rectangle. With **focalFilter**, the neighborhood is a user-defined a matrix of weights and could approximate any shape by giving some cells zero weight. The **focalNA** function only computes new values for cells that are NA in the input RasterLayer.

5.4 Distance

There are a number of distance related functions. **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. exluding 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 adanced distance calculations (cost distance, resistance distance)

5.5 Spatial configuration

Function **clump** identifies groups of cells that are connected. **edge** identifies edges, that is transitions beween cell values. **area** computes the size of each grid cell (for unprojected rasters)

5.6 Predictions

The package has two functions 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. Function **interpolate** is similar but is for models that use coordinates as predictor variables, for example in kriging and spline interpolation.

5.7 Vector to raster conversion

The raster packages supports point, line, and polygon to raster conversion. For vector type data (points, lines, polygons), objects of 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. **pointsToRaster** takes a RasterLayer 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 (with **polygonsToRaster**) is typically done to create a RaterLayer that can act as a mask, i.e. to set to NA a set of cells of a RasterLayer, 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 state.

It is also possible to convert the valus of a RasterLayer to points or polygons, using **rasterToPoints** and **rasterToPolygons**. Both functions only return values for cells that are not NA. Unlike **rasterToPolygons**, **rasterToPoints** is reasonably efficient and allows you to provide a function to subset the output before it is produced (which can be necessary for very large rasters as the point object is created in memory).

6 Summary functions

When used with a RasterLayer as first argument, normal summary statistics functions such as min, max and mean return a RasterLayer. To, instead, obtain a summary for all cells of a single RasterLayer you can use cellStats. You can use **freq** to make a frequency table, or **count** to count the number of cells with a specified value. Use **zonal** to summarize a RasterLayer using zones (areas

with the same integer number) defined in another RasterLayer and **crosstab** to cross-tabulate two RasterLayer objects.

```
> r <- raster(ncol=36, nrow=18)
> r[] <- runif(ncell(r))
> cellStats(r, mean)

[1] 0.4923572

> s = r
> s[] <- round(runif(ncell(r)) * 5)
> zonal(r,s,median)

  zone    median
1    0 0.4289517
2    1 0.4930421
3    2 0.5005314
4    3 0.5297031
5    4 0.4370234
6    5 0.4204555

> freq(s)

     value count
[1,]     0    61
[2,]     1   122
[3,]     2   130
[4,]     3   138
[5,]     4   132
[6,]     5    65

> count(s, 3)

[1] 138

> crosstab(r*3, s)

     second
first  0  1  2  3  4  5
    0 15 19 23 18 28 14
    1 22 43 42 43 47 26
    2 11 39 48 49 37 15
    3 13 21 17 28 20 10
```

7 Plotting

Several generic functions have been implemented for Raster* objects to create maps and other plot types. Use 'plot' to create a map of a RasterLayer.

When plot is used with a RasterLayer, it uses code taken from the image.plot function in the fields package, which calls the function 'image' (but, by default, adds a legend). 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). After plotting a RasterLayer 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 Raster* object, all layers are plotted (up to 16), unless the layers desired are indicated with an additional argument.

You can also use the following functions with a RasterLayer as argument: **hist, persp, contour**, and **density**. See the help files for more info. You can use **plot3D** to create an interactive 3D plot (you need the rgl package for this).

With **click** it is possible to interactively query a Raster* object by clicking once or several times on a map plot.

8 Row, column and cell numbers

The cell number is an important concept in the raster package. Raster data can be thought of as a matrix, but in a RasterLayer 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(raster)
> r <- raster(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, c(0,0))

[1] 343

> colFromX(r, 0)

[1] 19

> rowFromY(r, 0)

[1] 10
```

9 Accessing cell values

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

```
> r <- raster(system.file("external/test.grd", package="raster"))
> getValues(r, 50)[35:39]

[1] 456.878 485.538 550.788 580.339 590.029

> getValuesBlock(r, 50, 1, 35, 5)

[1] 456.878 485.538 550.788 580.339 590.029
```

You can also read values using cell numbers or coordinates (xy).

```
> cells <- cellFromRowCol(r, 50, 35:39)
> cells

[1] 3955 3956 3957 3958 3959

> cellValues(r, cells)

[1] 456.878 485.538 550.788 580.339 590.029

> xy = xyFromCell(r, cells)
> xy
```

```
          x      y
[1,] 179780 332020
[2,] 179820 332020
[3,] 179860 332020
[4,] 179900 332020
[5,] 179940 332020
```

```
> xyValues(r, xy)
```

```
[1] 456.878 485.538 550.788 580.339 590.029
```

In addition, you can use standard R indexing to access values. You can also to replace values (assign new values to cells). If you replace a value in a RasterLayer 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.

```
> r[cells]
```

```
[1] 456.878 485.538 550.788 580.339 590.029
```

```
> r[1:4]
```

```
[1] NA NA NA NA
```

```
> filename(r)
```

```
[1] "/tmp/Rinst1198023405/raster/external/test.grd"
```

```
> r[2:3] <- 10
> r[1:4]
```

```
[1] NA 10 10 NA
```

```
> filename(r)
```

```
[1] ""
```

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. To do so you need to use double brackets. As in ordinary R matrices, the first index represents the row number, the second the column number.

```
> r[1]
```

```
[1] NA
```

```
> r[[1,1]]
```

```
[1] NA
```

```
> r[[1:3,1:3]]
```

```
     [,1] [,2] [,3]
[1,]   NA   10   10
[2,]   NA   NA   NA
[3,]   NA   NA   NA
```

```
> r[[75:90, 15]]
```

```
 [1]       NA  946.029 1274.400 1103.470  746.170  706.596
 [7]  662.454  608.678  610.869  566.225  530.387  495.301
[13]  475.535  461.082  439.506  419.139
```

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

10 Writing files

10.1 File format

Raster can read most, and write several raster file formats, via the `rgdal` package. However, it directly reads and writes a native 'rasterfile' format. A rasterfile consists of two files: a binary sequential data file and a text header file. The header file is of the "windows .ini" type. When reading, you do not have to specify the file format, but you do need to do that when writing (except when using the default native format). This file format is also used in DIVA-GIS (`http://www.diva-gis.org/`). See the help files for functions **writeRaster** and **saveAs**.

11 Session options

There are a number of session options that can be set and these can be saved to make them persistent in between sessions. We would advice against changing the default values unless you have pressing need to do so. They all have to do with reading and writing files. You can set the preferred file format and data type. You can set the default value for overwrite to TRUE (be careful with that one!), and you specify a default progress-bar. All of these values can also be provided as arguments of functions where they apply. 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. You can also set the tmpdir, the location where such files are written. The option chunksize determines 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.

12 Coercion to objects of other classes

Although the raster package defines its own set of classes, it is easy to coerce objects of these classes to objects of the 'spatial' family defined in the sp package. This allows for using functions defined by sp (e.g. spplot) and for using other packages that expect spatial* objects. To create a Raster object from variable n in a SpatialGrid* x use `raster(x, n)` or `stack(x)` or `brick(x)`. Vice versa use `as( , )`

You can also convert objects of class "im" (spatstat) and "asc" (adehabitat) to a RasterLayer and "kasc" (adehabitat) to a RasterStack or Brick using the `raster(x)`, `stack(x)` or `brick(x)` function.

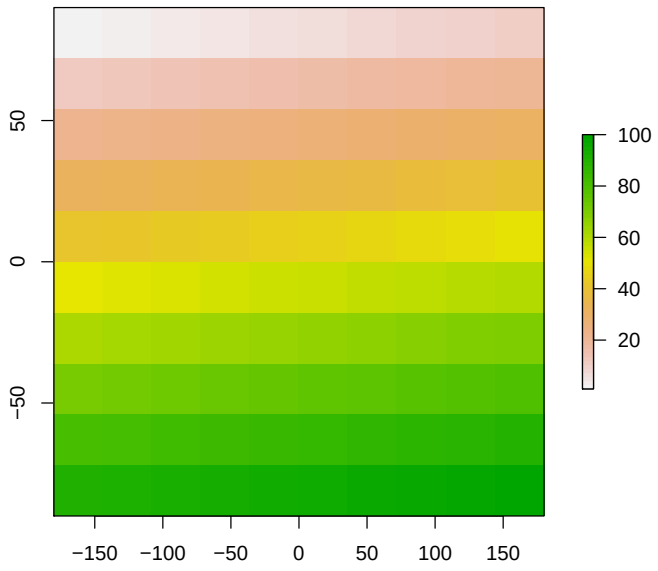
```
> r1 <- raster(ncol=36, nrow=18)
> r2 <- r1
> r1[] <- runif(ncell(r1))
> r2[] <- runif(ncell(r1))
> s <- stack(r1, r2)
> sgdf <- as(s, 'SpatialGridDataFrame')
> newr2 <- raster(sgdf, 2)
> news <- stack(sgdf)
```

13 Extending raster objects

It is straightforward to build on the Raster* objecs using the S4 inheritance mechanism. Say you need objects that behave like a RasterLayer, but have some additional proerties that you need to use in your own functions (S4 methods). See John Chambers' book "Software for Data Analysis, Programming with R" (Springer, 2008), and the help pages of the Methods package for more info. Below is an example:

```
> setClass ('myRaster',
+         contains = 'RasterLayer',
+         representation (
+                 important = 'data.frame',
+                 essential = 'character'
+         ) ,
+         prototype (
+                 important = data.frame(),
+                 essential = ''
+         )
+ )

[1] "myRaster"

> r = raster(nrow=10, ncol=10)
> m <- as(r, 'myRaster')
```

```
> m@important <- data.frame(id=1:10, value=runif(10))
> m@essential <- 'my own slot'
> m[] <- 1:ncell(m)
> plot(m)
```




```
> setMethod ('show' , 'myRaster',
+         function(object) {
+                 callNextMethod(object) # call show(RasterLayer)
+                 cat('essential:', object@essential, '\n')
+                 cat('important information:\n')
+                 print( object@important)
+         })

[1] "show"

> m

class       : myRaster
filename    :
nrow        : 10
ncol        : 10
ncell       : 100
min value   : 1
```

```
max value   : 100
projection  : +proj=longlat +datum=WGS84
xmin        : -180
xmax        : 180
ymin        : -90
ymax        : 90
xres        : 36
yres        : 18

essential: my own slot
important information:
   id      value
1   1 0.80961027
2   2 0.26272279
3   3 0.14511804
4   4 0.78071258
5   5 0.99111485
6   6 0.97080702
7   7 0.03102034
8   8 0.86724888
9   9 0.54974185
10 10 0.88190015
```