# R: Visualization

*Roger Bivand*

*Thursday, 5 September 2019, 11:35-12:00*

**Required current contributed CRAN packages:**

I am running R 3.6.1, with recent `update.packages()`.

```
needed <- c("sf", "stars", "sp", "classInt", "raster", "colorspace", "RColorBrewer", "ggplot2", "cartogr
```

```
library(sf)
```

```
## Linking to GEOS 3.7.2, GDAL 3.0.1, PROJ 6.2.0
```

```
lux_tmerc <- st_read("../data/lux_tmerc.gpkg")
```

```
## Reading layer `lux_tmerc' from data source `/home/rsb/presentations/ectqg19-workshop/data/lux_tmerc.g
## Simple feature collection with 102 features and 16 fields
## geometry type:  MULTIPOLYGON
## dimension:      XY
## bbox:           xmin: 48930.89 ymin: 57015.29 xmax: 106113.8 ymax: 138759.2
## epsg (SRID):    NA
## proj4string:    +proj=tmerc +lat_0=49.8333333333333 +lon_0=6.16666666666667 +k=1 +x_0=80000 +y_0=100
```

## Non-spatial visualisation

```
names(lux_tmerc)
```

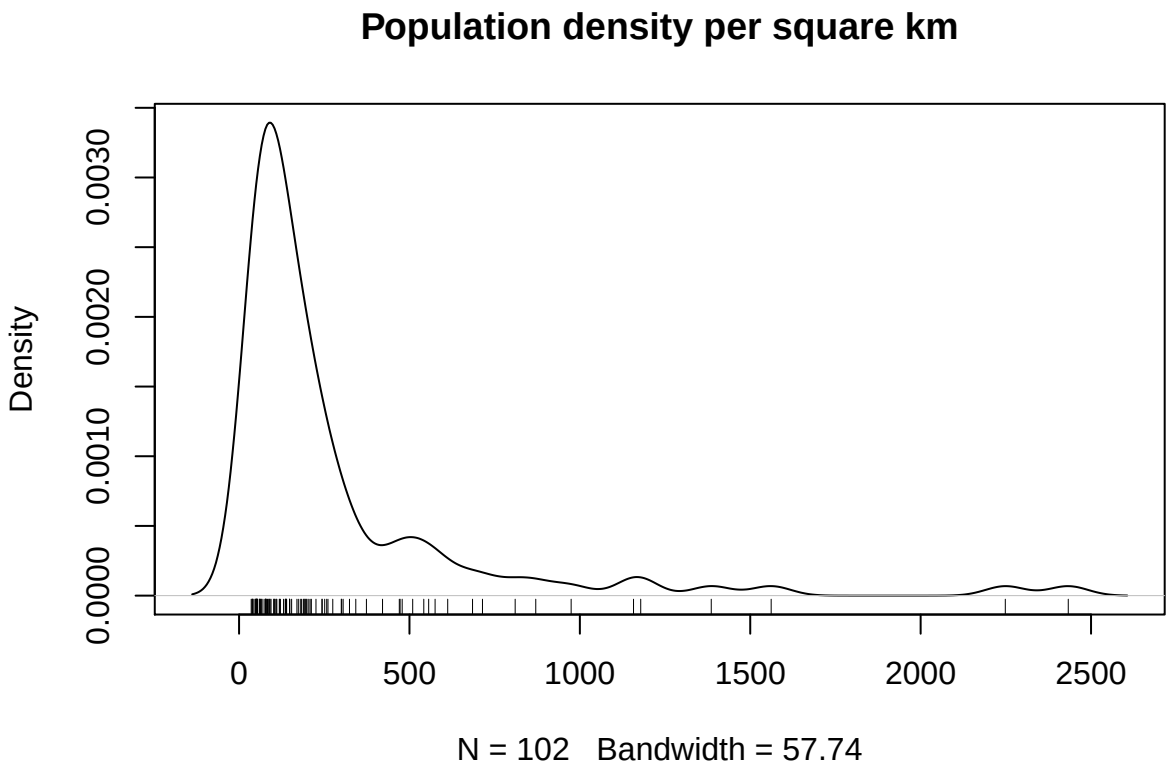
```
##  [1] "POPULATION"     "COMMUNE_1"      "LAU2"           "X_subtype"
##  [5] "COMMUNE"        "DISTRICT"       "CANTON"         "tree_count"
##  [9] "ghsl_pop"       "light_level"    "area"           "area_err"
## [13] "pop_den"        "ghsl_den"       "ghsl_tiff"      "ghsl_warp_diff"
## [17] "geom"
```

```
sapply(lux_tmerc, function(x) class(x)[1])
```

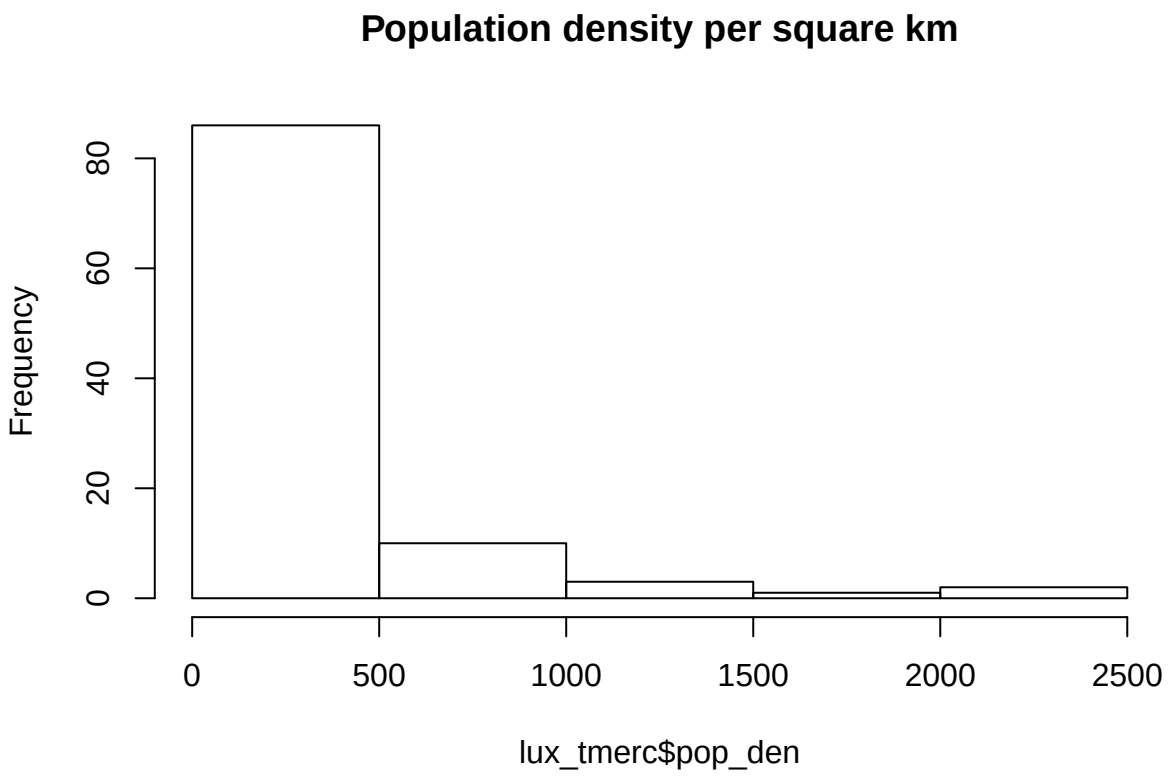
```
##         POPULATION          COMMUNE_1               LAU2
##          "numeric"           "factor"           "factor"
##          X_subtype            COMMUNE           DISTRICT
##           "factor"           "factor"           "factor"
##             CANTON         tree_count           ghsl_pop
##           "factor"          "numeric"          "numeric"
##        light_level               area           area_err
##          "numeric"          "numeric"          "numeric"
##            pop_den           ghsl_den          ghsl_tiff
##          "numeric"          "numeric"          "numeric"
##     ghsl_warp_diff               geom
##          "numeric" "sfc_MULTIPOLYGON"
```

**Univatiate continuous**

```
plot(density(lux_tmerc$pop_den), main="Population density per square km")
rug(lux_tmerc$pop_den)
```

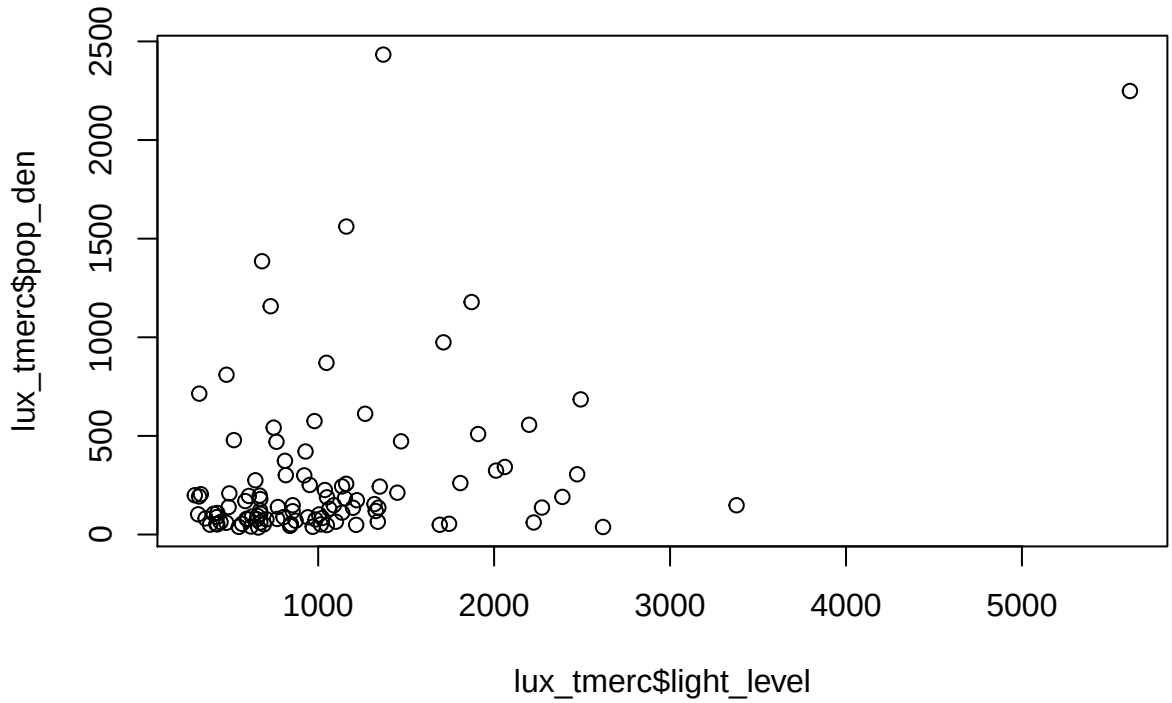



```
hist(lux_tmerc$pop_den, main="Population density per square km")
```

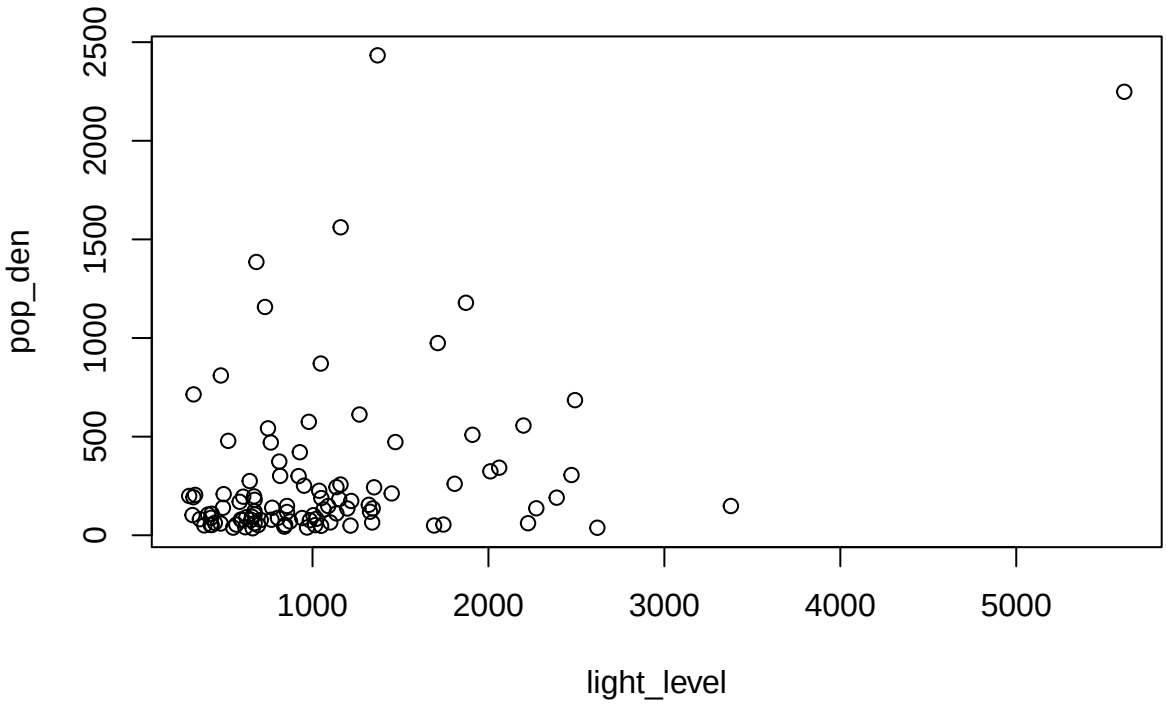



**Bivariate continuous**

```
plot(lux_tmerc$light_level, lux_tmerc$pop_den)
```

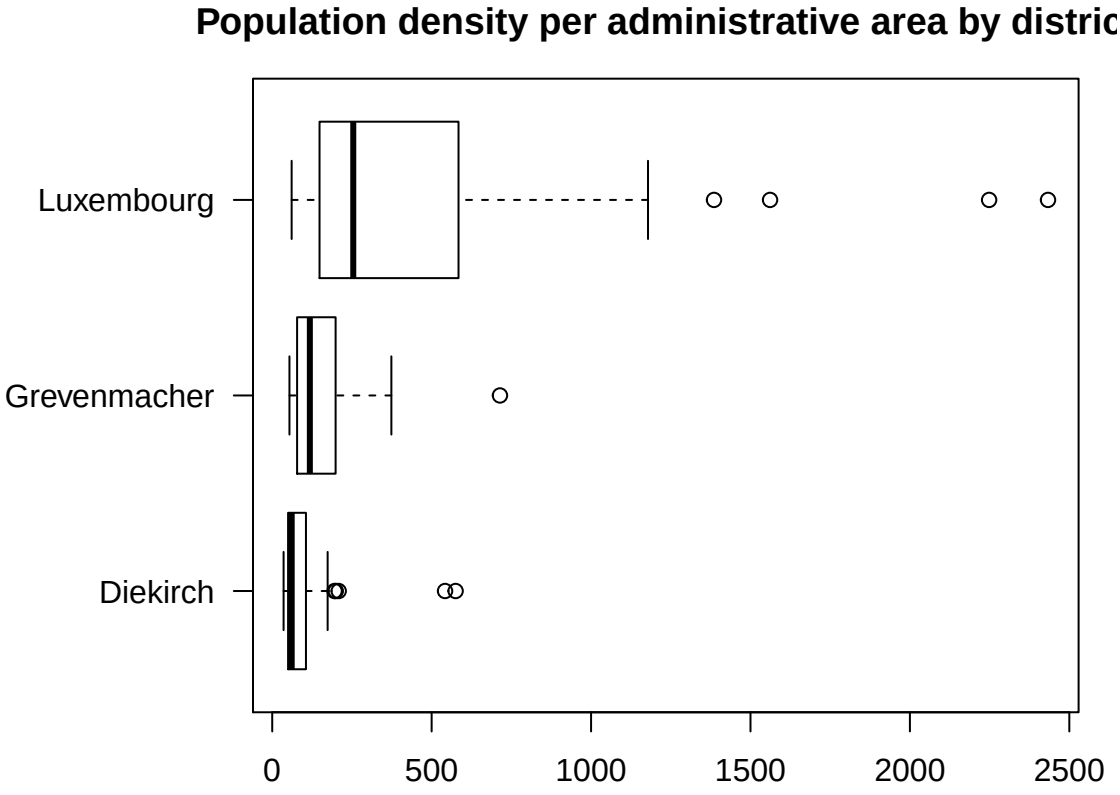



```
plot(pop_den ~ light_level, lux_tmerc)
```




**Categorical plots**

```
opar <- par(mar=c(3, 10, 3, 1), las=1)
boxplot(pop_den ~ DISTRICT, lux_tmerc, horizontal=TRUE, ylab="", main="Population density per administra
```



```
par(opar)
```

## Spatial visualization

We have already seen some plot methods for `"sf"`, `"sfc"` and `"nb"` objects in several packages for static plots, and **mapview** for interactive visualization. Let's run through available packages, functions and methods quickly.

### Thematic mapping

**classInt** provides the key class interval determination for thematic mapping of continuous variables. The `classIntervals()` function takes a numeric vector (now also of classes POSIXt or units), a target number of intervals, and a style of class interval. Other arguments control the closure and precision of the intervals found.

```
library(classInt)
args(classIntervals)
```

```
## function (var, n, style = "quantile", rtimes = 3, ..., intervalClosure = c("left",
##     "right"), dataPrecision = NULL, warnSmallN = TRUE, warnLargeN = TRUE,
##     largeN = 3000L, samp_prop = 0.1, gr = c("[", "]"))
## NULL
```

We'll find 7 intervals using Fisher natural breaks for the pop_denation variable:

```
(cI <- classIntervals(lux_tmerc$pop_den, n=7, style="fisher"))
```

```
## style: fisher
##   one of 1,267,339,920 possible partitions of this variable into 7 classes
## [35.60505,161.9526) [161.9526,358.0949) [358.0949,648.6537)
##                  56                  25                  10
## [648.6537,922.5014) [922.5014,1282.103) [1282.103,1904.928)
##                   4                   3                   2
## [1904.928,2433.165]
##                   2
```

```
(cI_pr <- classIntervals(lux_tmerc$pop_den, n=7, style="pretty"))
```

```
## style: pretty
##   one of 4,082,925 possible partitions of this variable into 5 classes
##     [0,500)  [500,1000) [1000,1500) [1500,2000) [2000,2500]
##          86          10           3           1           2
```

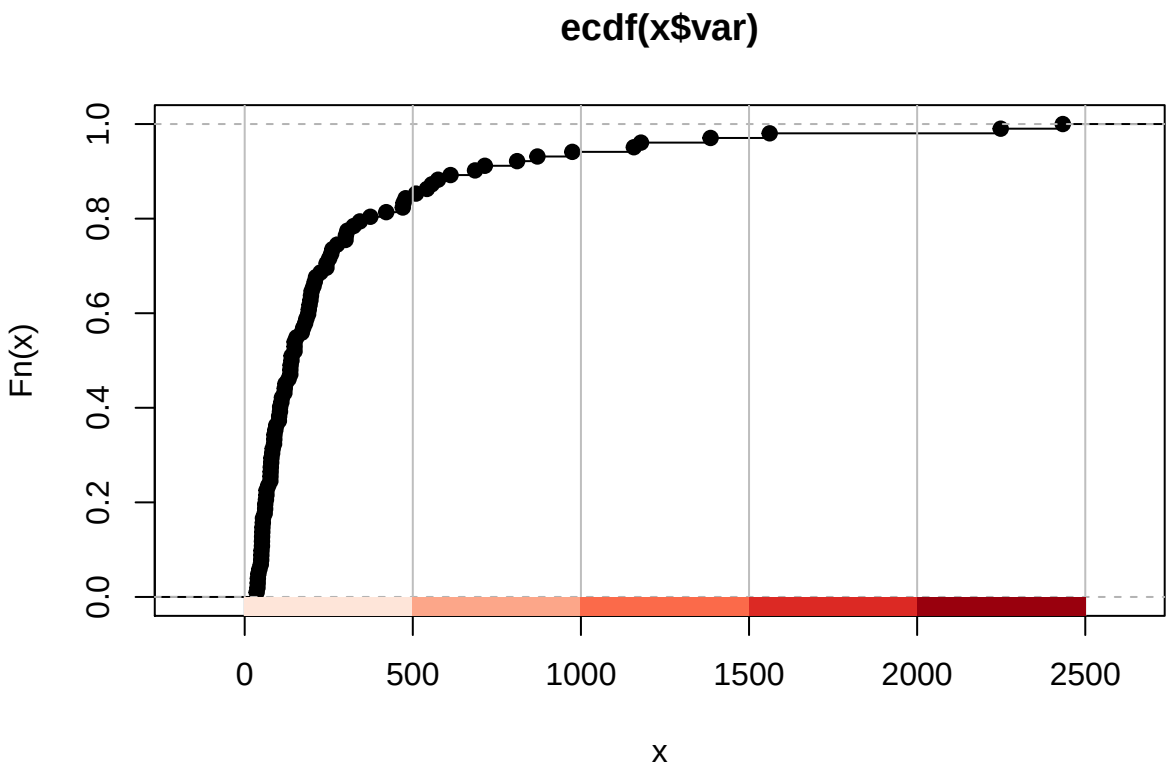
```
(cI_qu <- classIntervals(lux_tmerc$pop_den, n=7, style="quantile"))
```

```
## style: quantile
##   one of 1,267,339,920 possible partitions of this variable into 7 classes
## [35.60505,53.34652) [53.34652,79.53126) [79.53126,118.3855)
##                  15                  14                  15
## [118.3855,177.9923) [177.9923,252.0616) [252.0616,528.0658)
##                  14                  15                  14
## [528.0658,2433.165]
##                  15
```

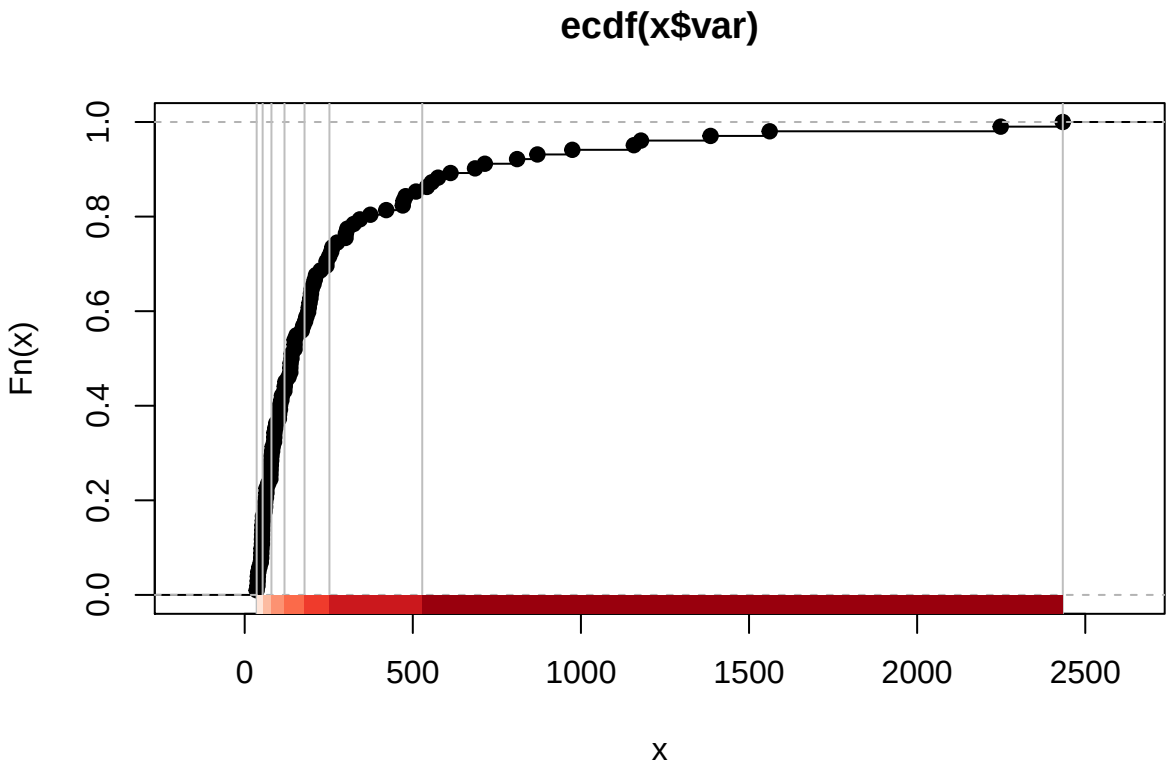
We also need to assign a palette of graphical values, most often colours, to use to fill the intervals, and can inspect the intervals and fill colours with a plot method:

The **RColorBrewer** package gives by permission access to the ColorBrewer palettes accesible from the ColorBrewer website. Note that ColorBrewer limits the number of classes tightly, only 3–9 sequential classes

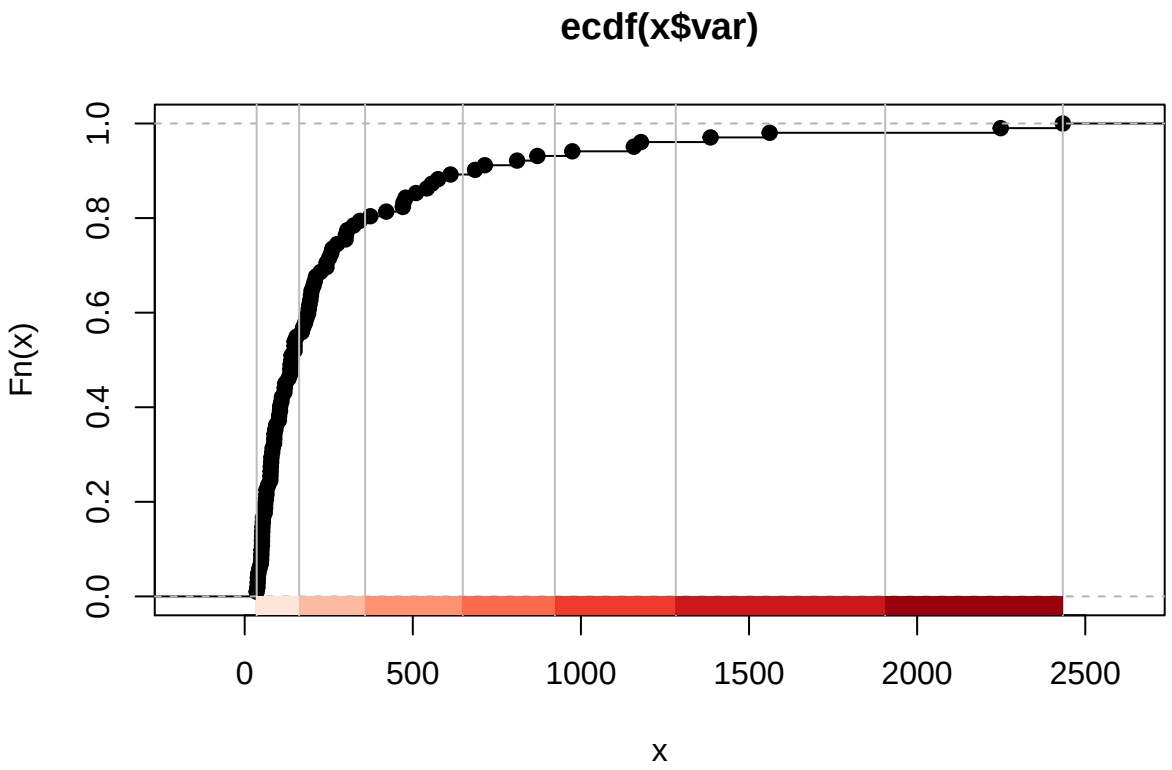
```
library(RColorBrewer)
pal <- RColorBrewer::brewer.pal((length(cI$brks)-1), "Reds")
plot(cI_pr, pal)
```




```
plot(cI_qu, pal)
```

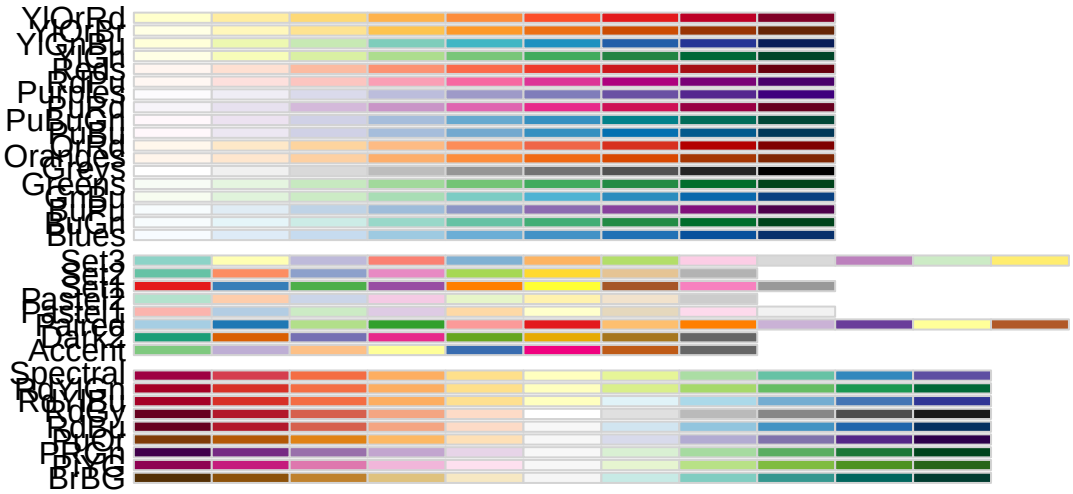



```
plot(cI, pal)
```




We can also display all the ColorBrewer palettes:

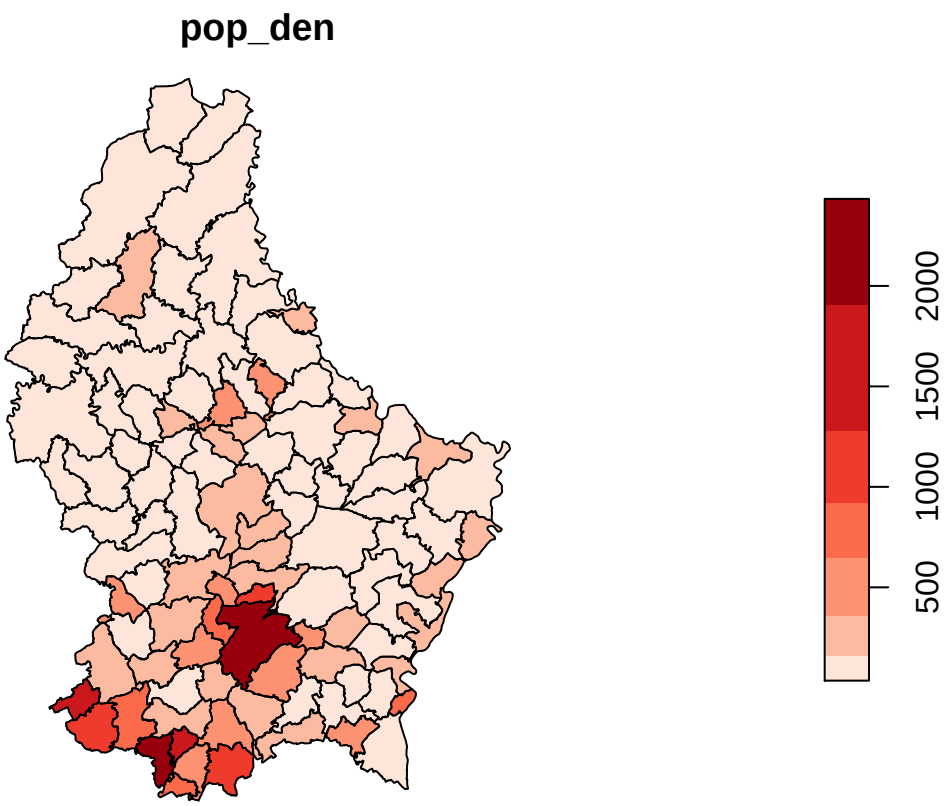
```
display.brewer.all()
```




### Package-specific plot and image methods

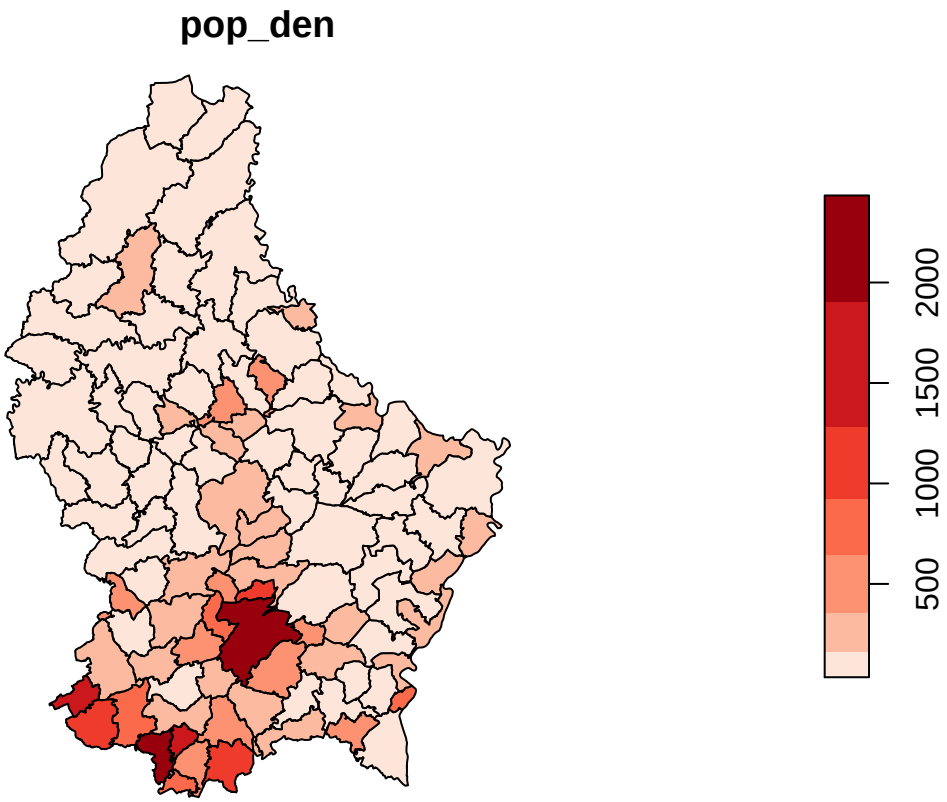
The **sp** package provided base graphics plot and image methods. **sf** provides plot methods using base graphics; the method for `"sf"` objects re-arranges the plot window to provide a colour key, so extra steps are needed if overplotting is needed:

```
plot(lux_tmerc[,"pop_den"], breaks=cI$brks, pal=pal)
```




(returns current `par()` settings); the method also supports direct use of **classInt**:

```
plot(lux_tmerc[,"pop_den"], nbreaks=7, breaks="fisher", pal=pal)
```

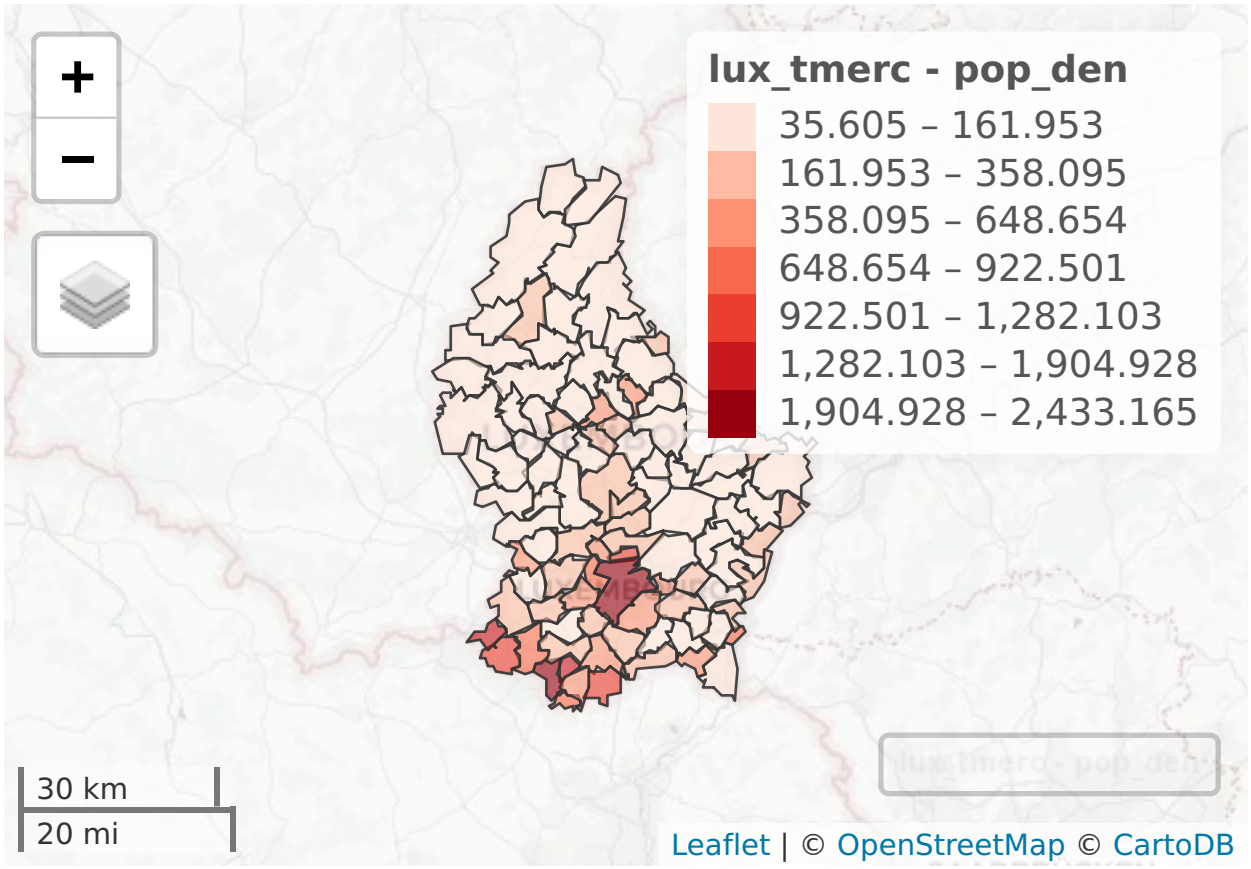



Earlier we used the plot method for `"sfc"` objects which does not manipulate the graphics device, and is easier for overplotting.

### The mapview package

**mapview**: Quickly and conveniently create interactive visualisations of spatial data with or without background maps. Attributes of displayed features are fully queryable via pop-up windows. Additional functionality includes methods to visualise true- and false-color raster images, bounding boxes, small multiples and 3D raster data cubes. It uses **leaflet** and other HTML packages.

```
library(mapview)
mapview(lux_tmerc, zcol="pop_den", col.regions=pal, at=cI$brks)
```




**The tmap package**

**tmap**: Thematic maps show spatial distributions. The theme refers to the phenomena that is shown, which is often demographical, social, cultural, or economic. The best known thematic map type is the choropleth, in which regions are colored according to the distribution of a data variable. The R package tmap offers a coherent plotting system for thematic maps that is based on the layered grammar of graphics. Thematic maps are created by stacking layers, where per layer, data can be mapped to one or more aesthetics. It is also possible to generate small multiples. Thematic maps can be further embellished by configuring the map layout and by adding map attributes, such as a scale bar and a compass. Besides plotting thematic maps on the graphics device, they can also be made interactive as an HTML widget. In addition, the R package **tmaptools** contains several convenient functions for reading and processing spatial data. See (Tennekes 2018) and Chapter 8 in (Lovelace, Nowosad, and Muenchow 2019).

The **tmap** package provides cartographically informed, grammar of graphics (gg) based functionality now, like **ggplot2** using **grid** graphics. John McIntosh tried with ggplot2, with quite nice results. I suggested he look at **tmap**, and things got better, because **tmap** can switch between interactive and static viewing. **tmap** also provides direct access to **classInt** class intervals.

```
library(tmap)
tmap_mode("plot")
```

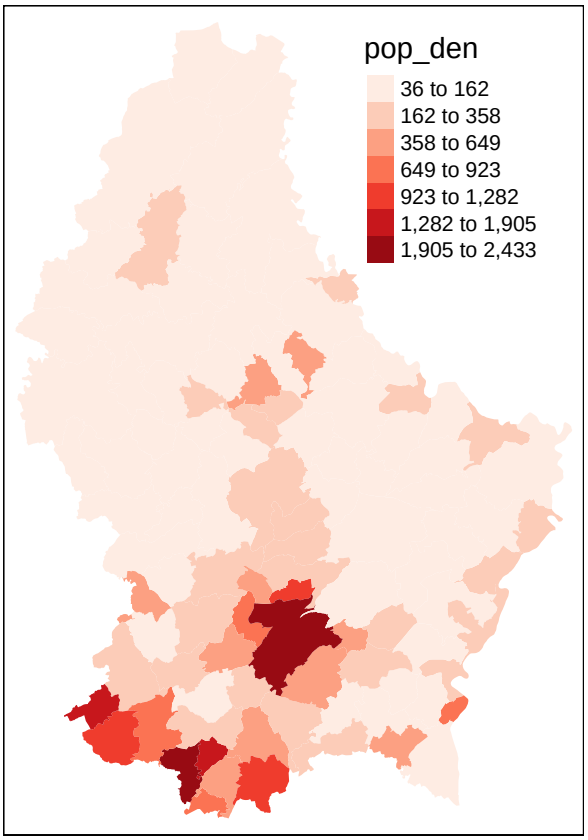
```
## tmap mode set to plotting
```

```
o <- tm_shape(lux_tmerc) + tm_fill("pop_den", style="fisher", n=7, palette="Reds")
class(o)
```

```
## [1] "tmap"
```

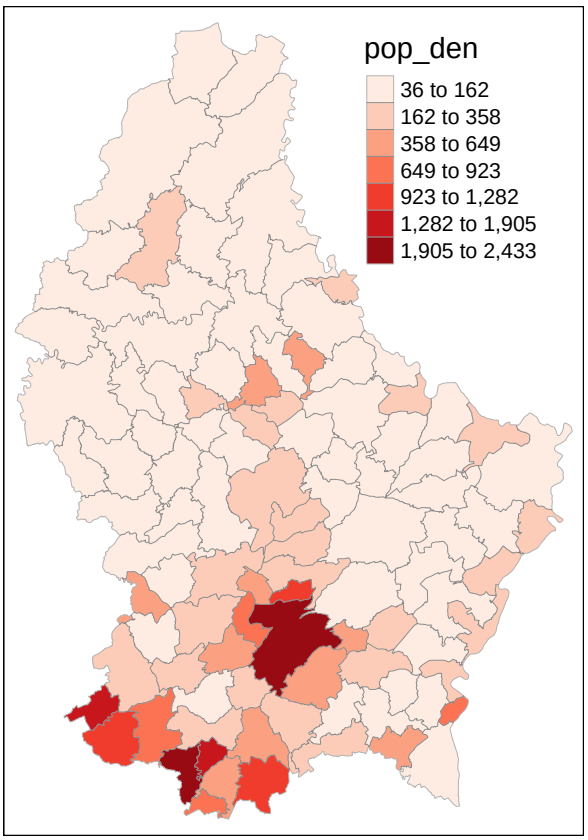
returns a "tmap" object, a **grid** GROB (graphics object), with print methods.

```
o
```




Since the objects are GROBs, they can be updated, as in **lattice** with **latticeExtra** or **ggplot2**:

```
o + tm_borders(alpha=0.5, lwd=0.5)
```

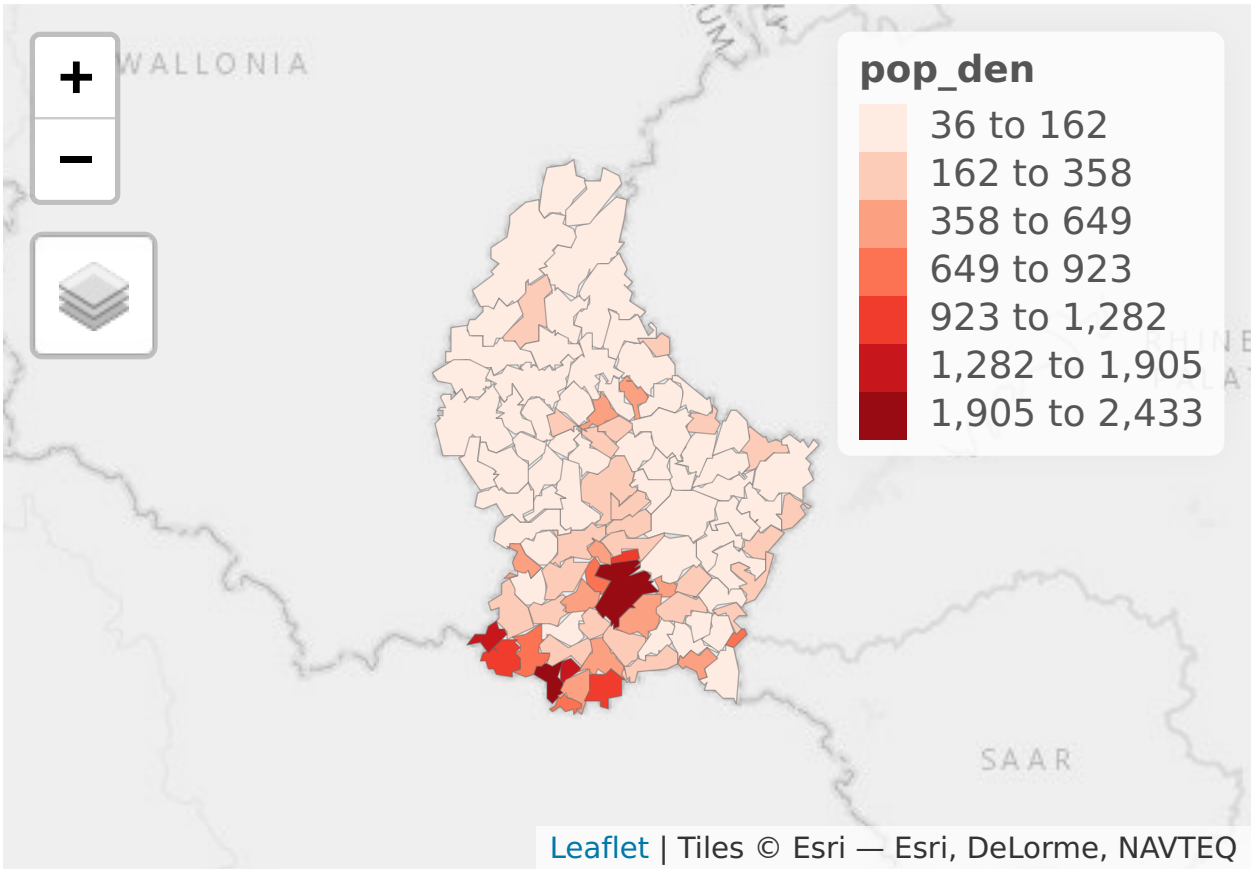



Using `tmap_mode()`, we can switch between presentation (`"plot"`) and interactive (`"view"`) plotting:

```
tmap_mode("view")
```

```
## tmap mode set to interactive viewing
```

```
o + tm_borders(alpha=0.5, lwd=0.5)
```




```
tmap_mode("plot")
```

```
## tmap mode set to plotting
```

There is also a Shiny tool for exploring palettes:

```
tmaptools::palette_explorer()
```

### The cartography package

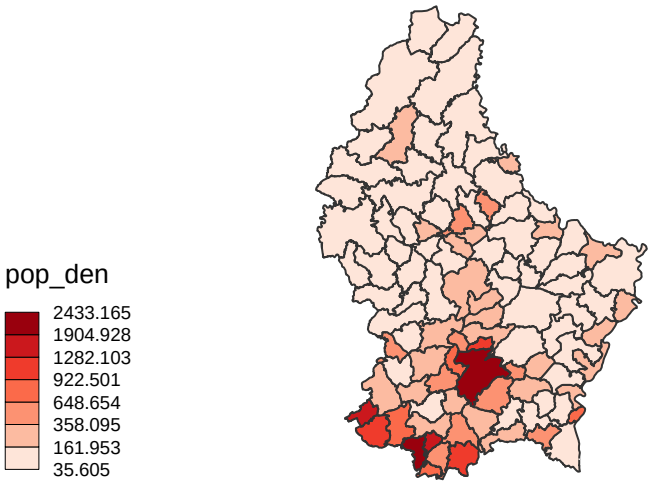
**cartography** helps to design cartographic representations such as proportional symbols, choropleth, typology, flows or discontinuities maps. It also offers several features that improve the graphic presentation of maps, for instance, map palettes, layout elements (scale, north arrow, title...), labels or legends. (Giraud and Lambert 2016, 2017), http://riatelab.github.io/cartography/vignettes/cheatsheet/cartography_cheatsheet.pdf. The package is associated with **rosm**: Download and plot Open Street Map http://www.openstreetmap.org/, Bing Maps http://www.bing.com/maps and other tiled map sources. Use to create basemaps quickly and add hillshade to vector-based maps. https://cran.r-project.org/web/packages/rosm/vignettes/rosm.html

The package organizes extra palettes:

```
library(cartography)
display.carto.all()
```


The plotting functions (mot methods) use base graphics:

```
choroLayer(lux_tmerc, var="pop_den", method="fisher-jenks", nclass=7, col=pal, legend.values.rnd=3)
```



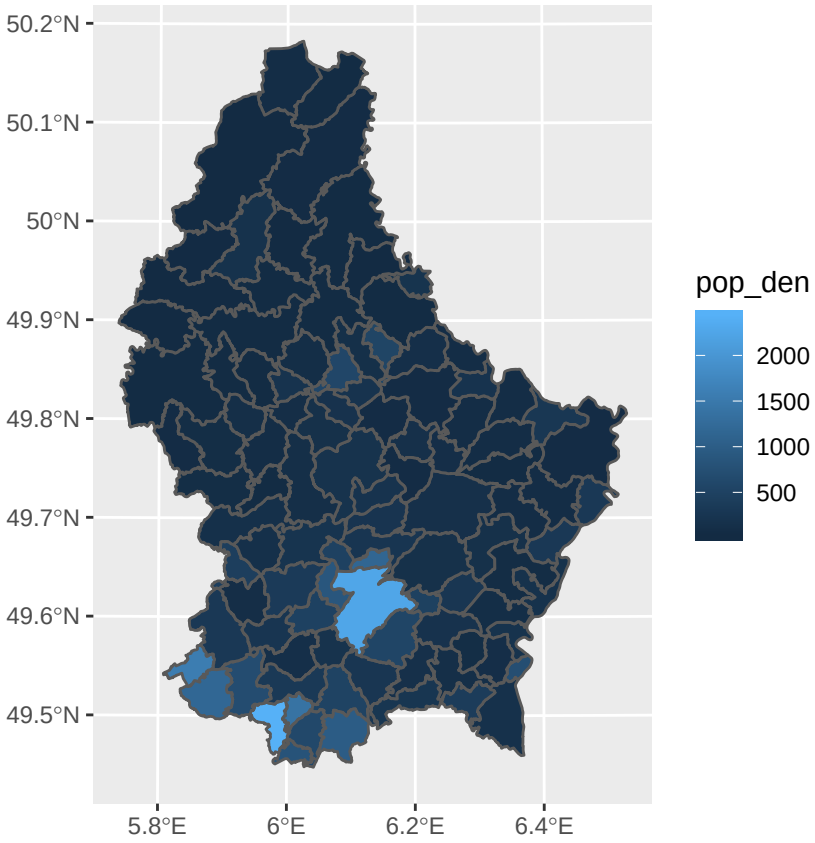

(returns NULL)

### The ggplot2 package

The **ggplot2** package provides the `geom_sf()` facility for mapping:

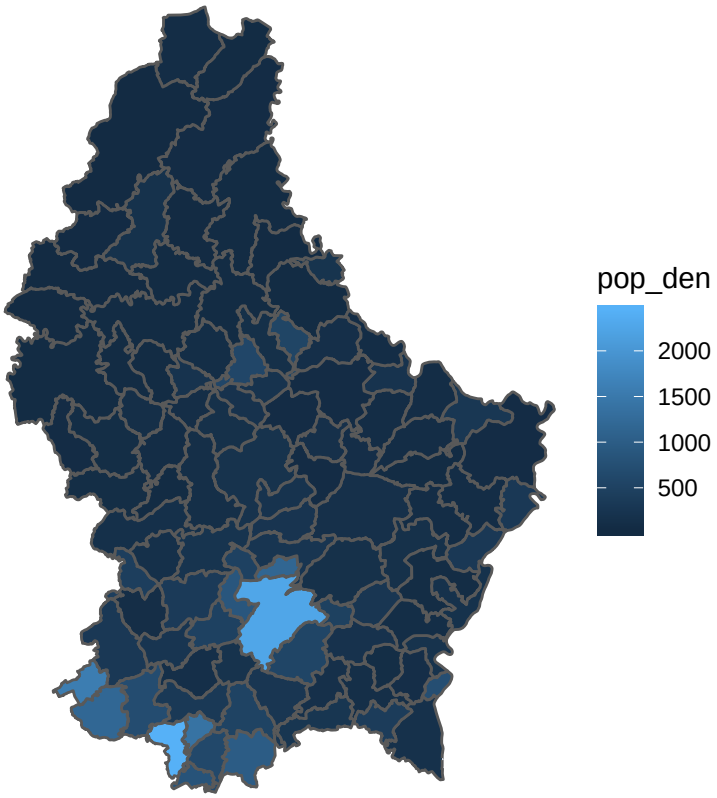
```
library(ggplot2)
```

```
g <- ggplot(lux_tmerc) + geom_sf(aes(fill=pop_den))
g
```

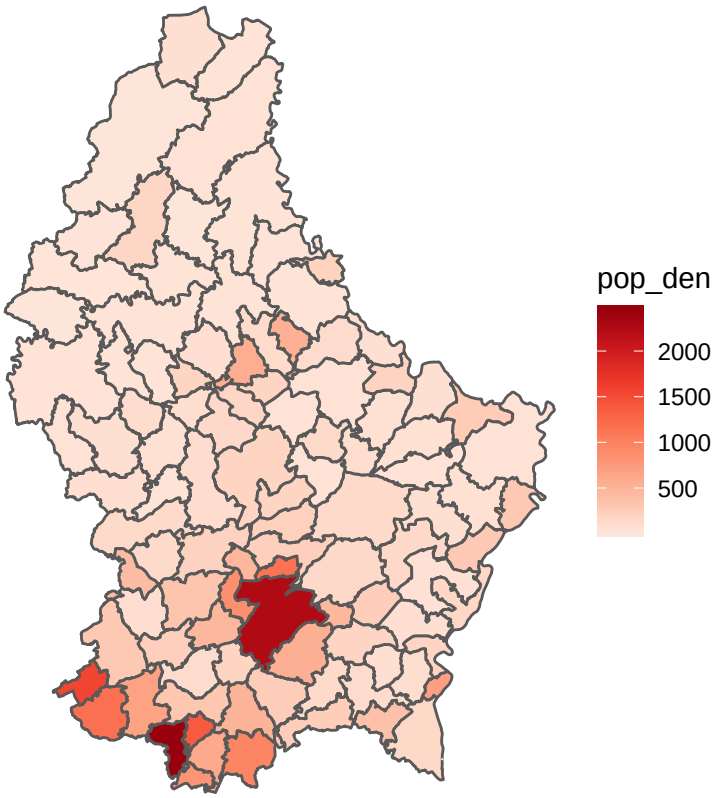



It is possible to set a theme that drops the arguably unnecessary graticule:

```
g + theme_void()
```




```
g + theme_void() + scale_fill_distiller(palette="Reds", direction=1)
```




but there is a lot of jumping through hoops to get a simple map. To get proper class intervals involves even

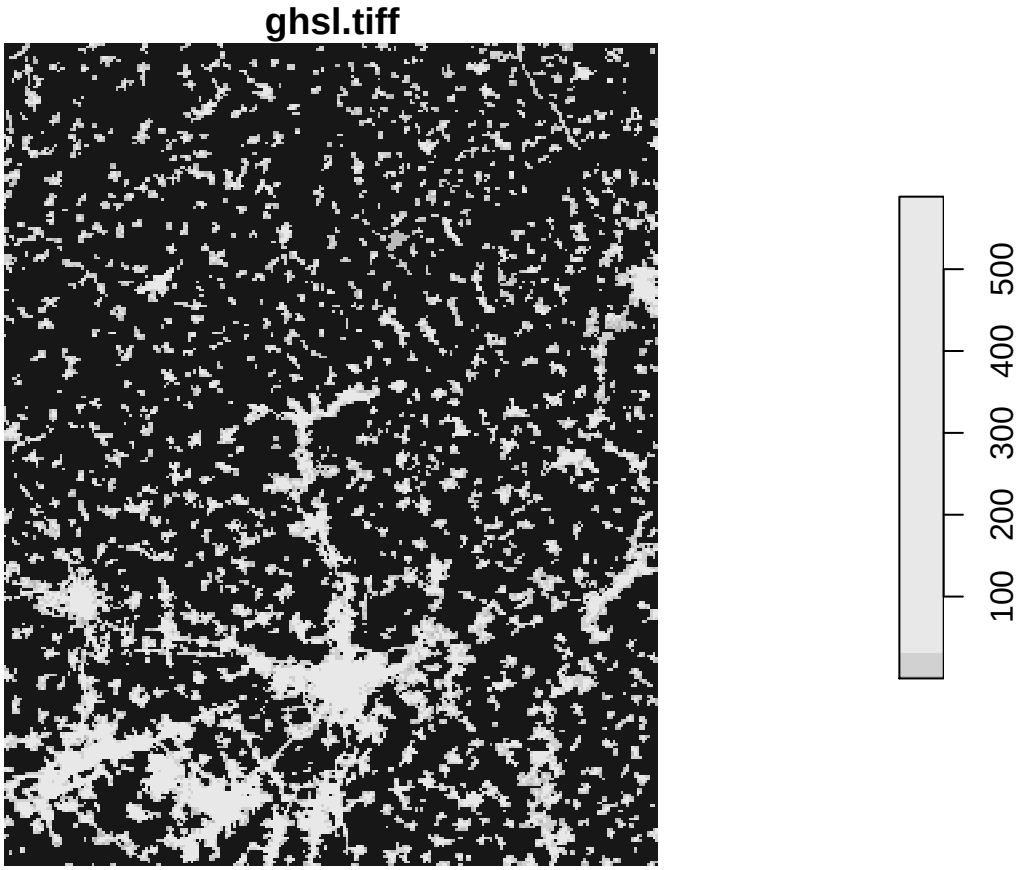
more work, because **ggplot2** takes specific, not general, positions on how graphics are observed. ColorBrewer eschews continuous colour scales based on cognitive research, but ggplot2 enforces them for continuous variables (similarly for graticules, which may make sense for data plots but not for maps).

### Raster visualization

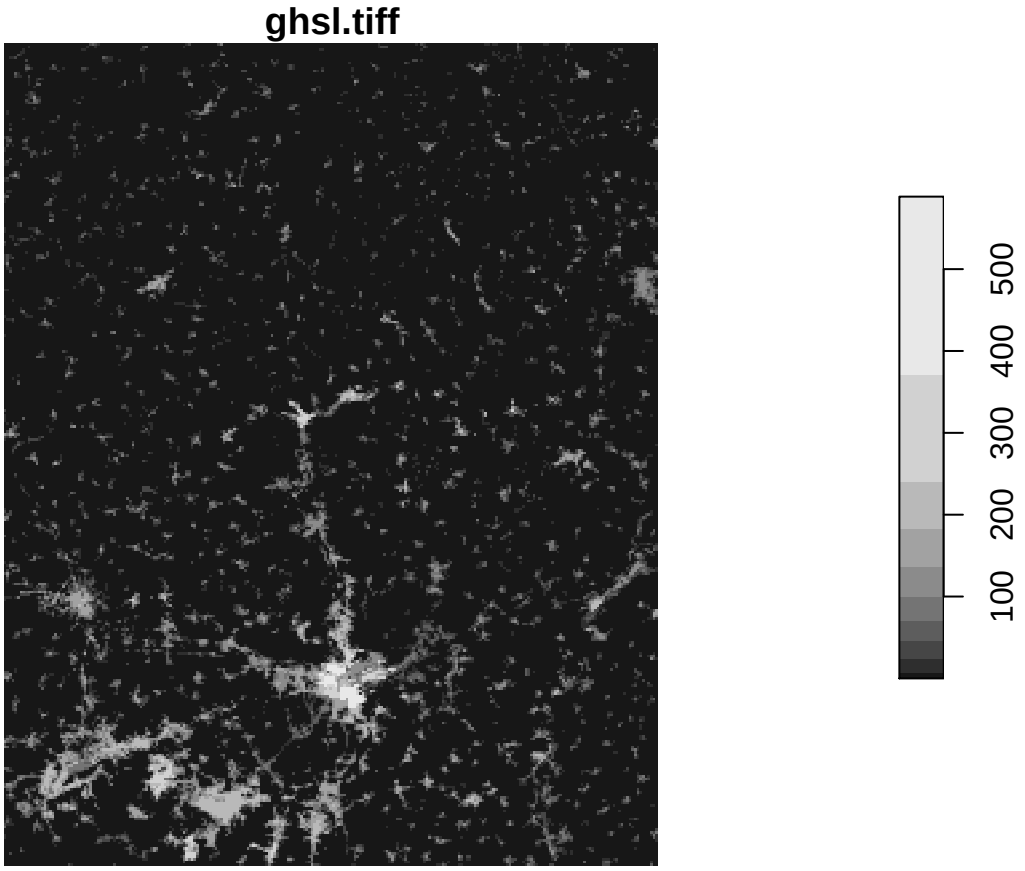
```
library(stars)
```

```
## Loading required package: abind
```

```
ghsl0 <- read_stars("../data/ghsl.tiff", proxy=FALSE)
plot(ghsl0["ghsl.tiff"])
```



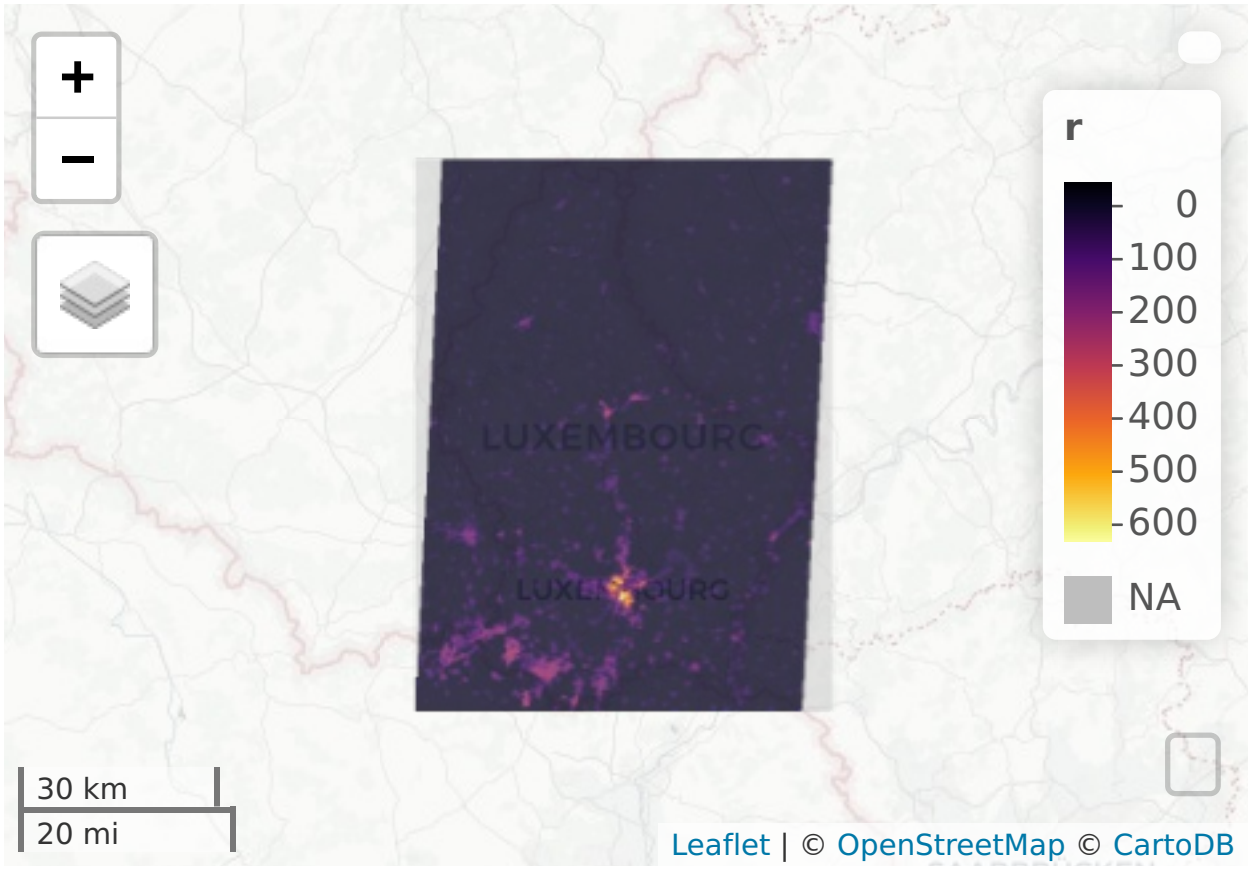

```
plot(ghsl0["ghsl.tiff"], breaks="fisher", nbreaks=11)
```




```
library(mapview)
library(raster)
```

```
## Loading required package: sp
```

```
r <- as(st_warp(ghsl0, crs=3857, cellsize=250), "Raster")
mapview(r)
```




**More on palettes**

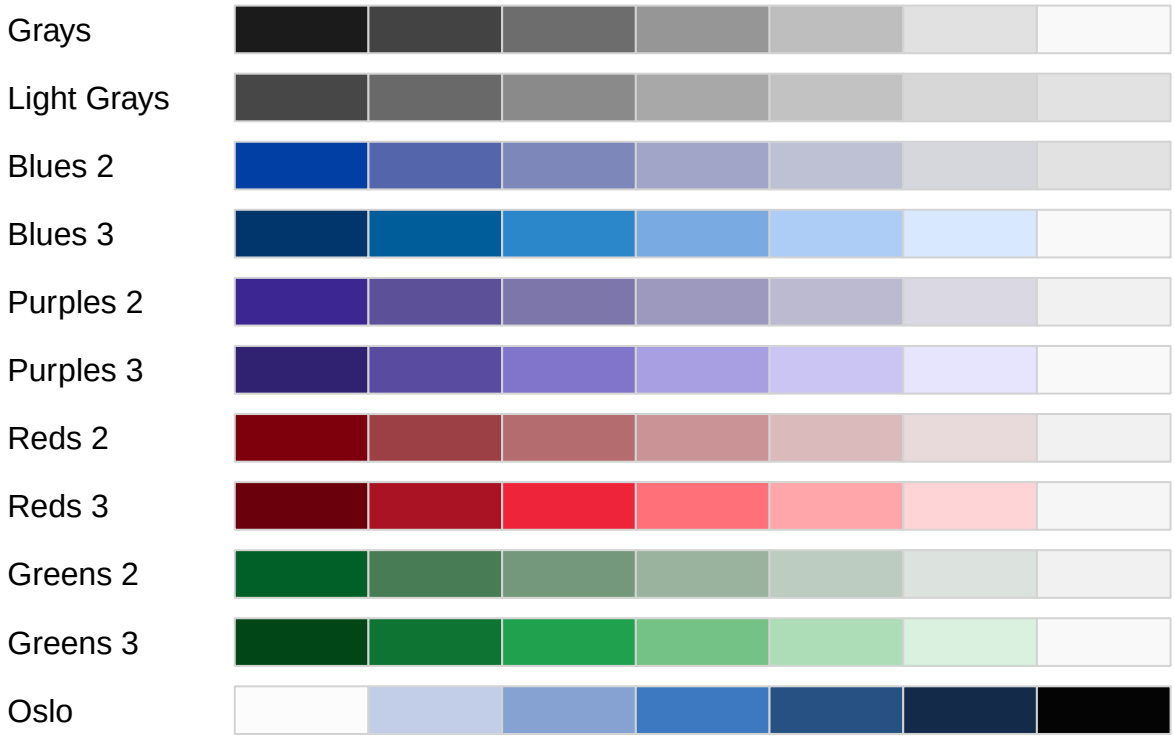
try exploring alternative class interval definitions and palettes, maybe also visiting http://hclwizard.org/ and its `hclwizard()` Shiny app, returning a palette generating function on clicking the "Return to R" button:

```
library(colorspace)
```

```
##
## Attaching package: 'colorspace'
```

```
## The following object is masked from 'package:raster':
##
##     RGB
```

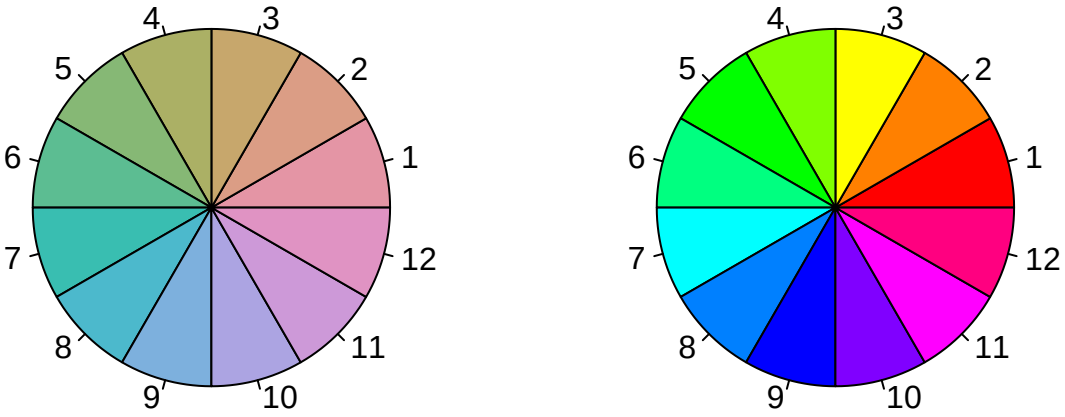
```
hcl_palettes("sequential (single-hue)", n = 7, plot = TRUE)
```




```
pal <- hclwizard()
pal(6)
```

The end of rainbow discussion is informative:

```
wheel <- function(col, radius = 1, ...)
  pie(rep(1, length(col)), col = col, radius = radius, ...)
opar <- par(mfrow=c(1,2))
wheel(rainbow_hcl(12))
wheel(rainbow(12))
```




```
par(opar)
```

Giraud, Timothée, and Nicolas Lambert. 2016. "Cartography: Create and Integrate Maps in Your R Workflow." *JOSS* 1 (4). The Open Journal. https://doi.org/10.21105/joss.00054.

———. 2017. "Reproducible Cartography." In *Advances in Cartography and Giscience. ICACI 2017. Lecture*

*Notes in Geoinformation and Cartography.*, edited by Michael Peterson, 173–83. Cham, Switzerland: Springer. https://doi.org/10.1007/978-3-319-57336-6_13.

Lovelace, Robin, Jakub Nowosad, and Jannes Muenchow. 2019. *Geocomputation with R*. Boca Raton, FL: Chapman and Hall/CRC. https://geocompr.robinlovelace.net/.

Tennekes, Martijn. 2018. "Tmap: Thematic Maps in R." *Journal of Statistical Software, Articles* 84 (6): 1–39. https://doi.org/10.18637/jss.v084.i06.