Introduction

Today we are focusing on a completely different spatial paradigm: raster images. Raster maps usually represent continuous phenomena such as elevation, temperature, population density or spectral data, but also discrete data, such as land cover types.

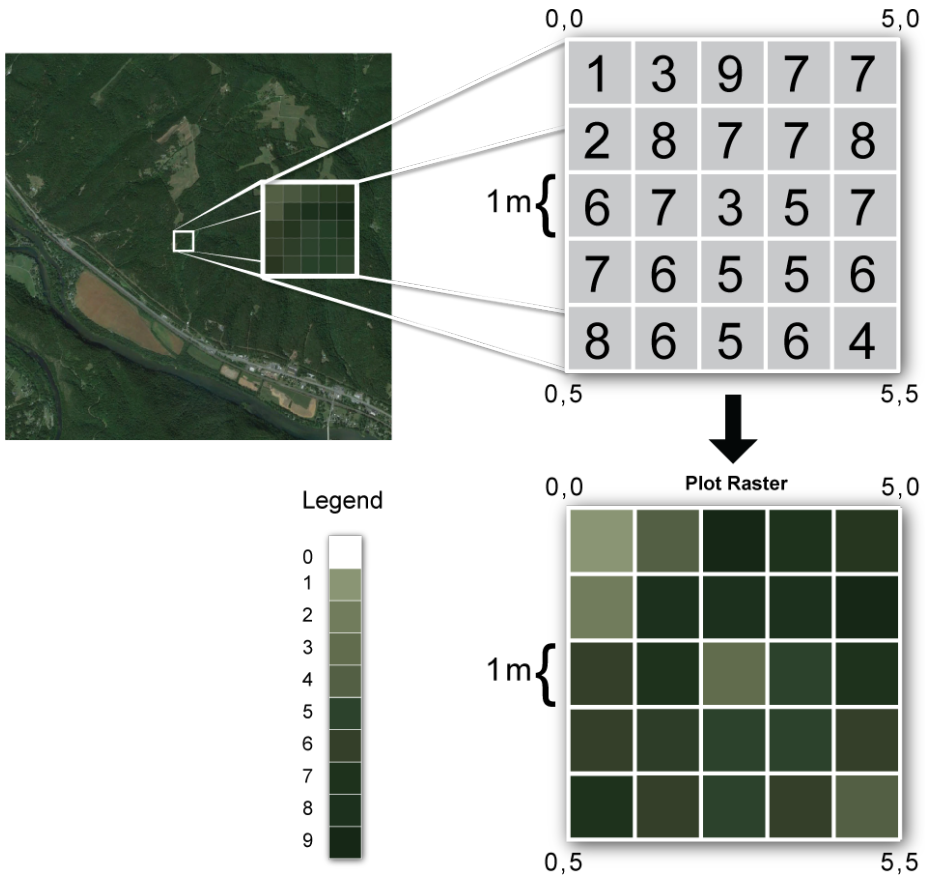
R handles these data using the eponymous `raster` package. The package is still part of the old `sp` world and does not play well with the `tidyverse`. It will be replaced over the next years by the new `stars` package (which is not mature yet).

Raster data are huge matrices. Keeping the study area constant, the finer the resolution, the more computing resources are needed.

Large rasters will quickly bring your laptop to its knees, even if they are mostly kept on disk. Processing high-resolution or global raster data efficiently will require dedicated computers or HPC clusters.

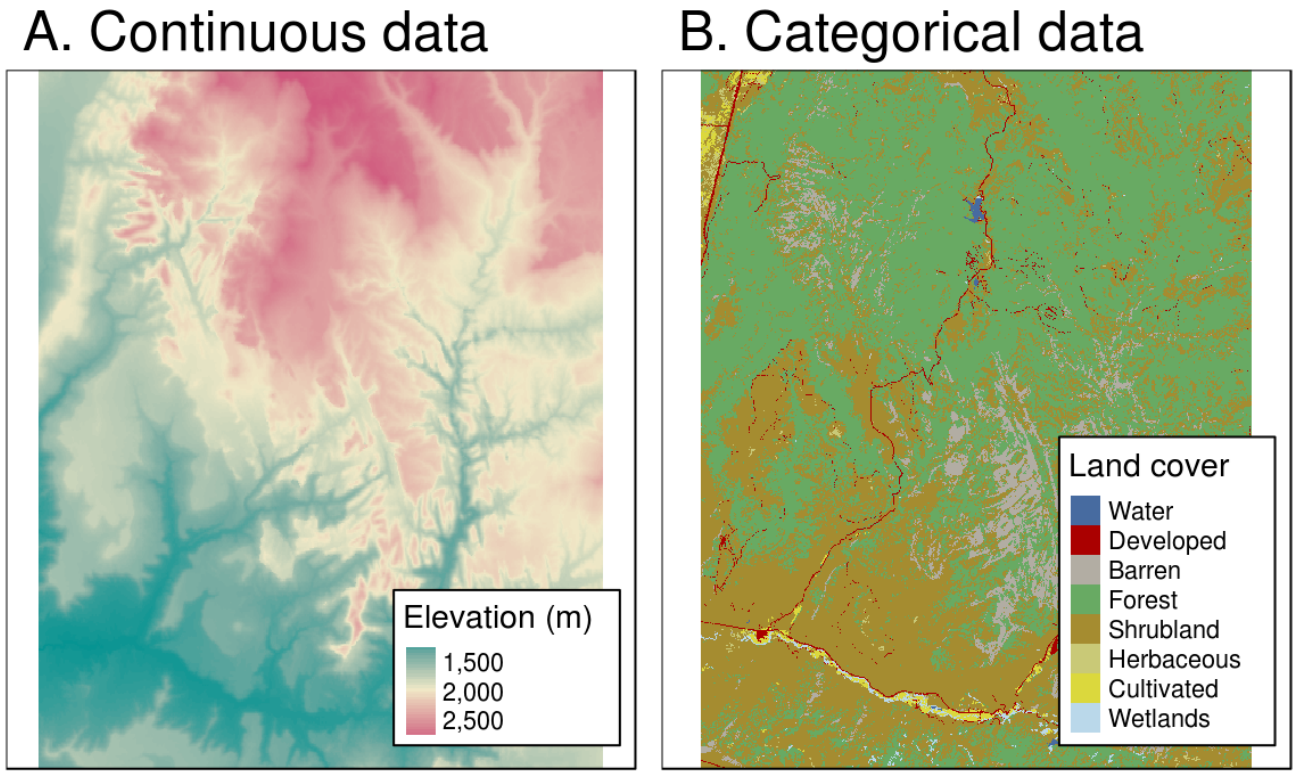
A different spatial paradigm

Raster data are stored as a grid of values which are rendered on a map as pixels. Each pixel represents an area on the Earth's surface.



Continuous vs. categorical rasters

Rasters store data in floats, integers or factors, although the latter is rare. Typically you have to know if they have a qualitative meaning.



GeoTiffs

Raster data come in a variety of formats. The most common is GeoTiff, but you will also come across ESRI grid rasters, ASCII text rasters or NetCDF raster time series.

A GeoTIFF is a standard .tif image with additional spatial (georeferencing) information embedded in the file as tags.

These tags can include the following raster metadata:

- A Coordinate Reference System (CRS)
- Spatial extent (bounding box)
- Values that represent missing data (the `NoData` value)
- The resolution of the data (e.g. 30 arc seconds)

Some of this information is also provided in a `.hdr` and `.tfw` file, but GeoTiffs open without these extras.

The raster package

All raster input-output and computation are done in *R* using the `raster` package. Install it using

```
install.packages("raster")
```

you may also want to install `rgdal` and `rgeos` while you are at it.

Put this in the preamble of each script:

```
require(raster)
require(tidyverse)
require(sf)
```

but be careful to load the `tidyverse` and `sf` *after* `raster`, otherwise you might get problems with `dplyr::select()`. Two raster functions are masked and have to be called using `raster::function_name`.

Reading and writing rasters

The raster function directly reads a `RasterLayer` from most known formats. Small rasters are loaded into RAM, the rest stays on disk.

```
r.ken <- raster("./data/F182010.tif")
inMemory(r.ken)

## [1] FALSE
```

You can write a GeoTiff using

```
writeRaster(r.ken, "./data/F182010new.tif")
```

but you should consider setting the data type to save storage space (e.g. `INT1S` for integers from -127 to 127).

Raster properties

```
res(r.ken) # resolution
```

```
## [1] 0.008333333 0.008333333
```

```
extent(r.ken) %>% as.vector() # bounding box
```

```
## [1] 33.912499 41.895832 -4.679166  4.629167
```

```
dim(r.ken) # rows and columns
```

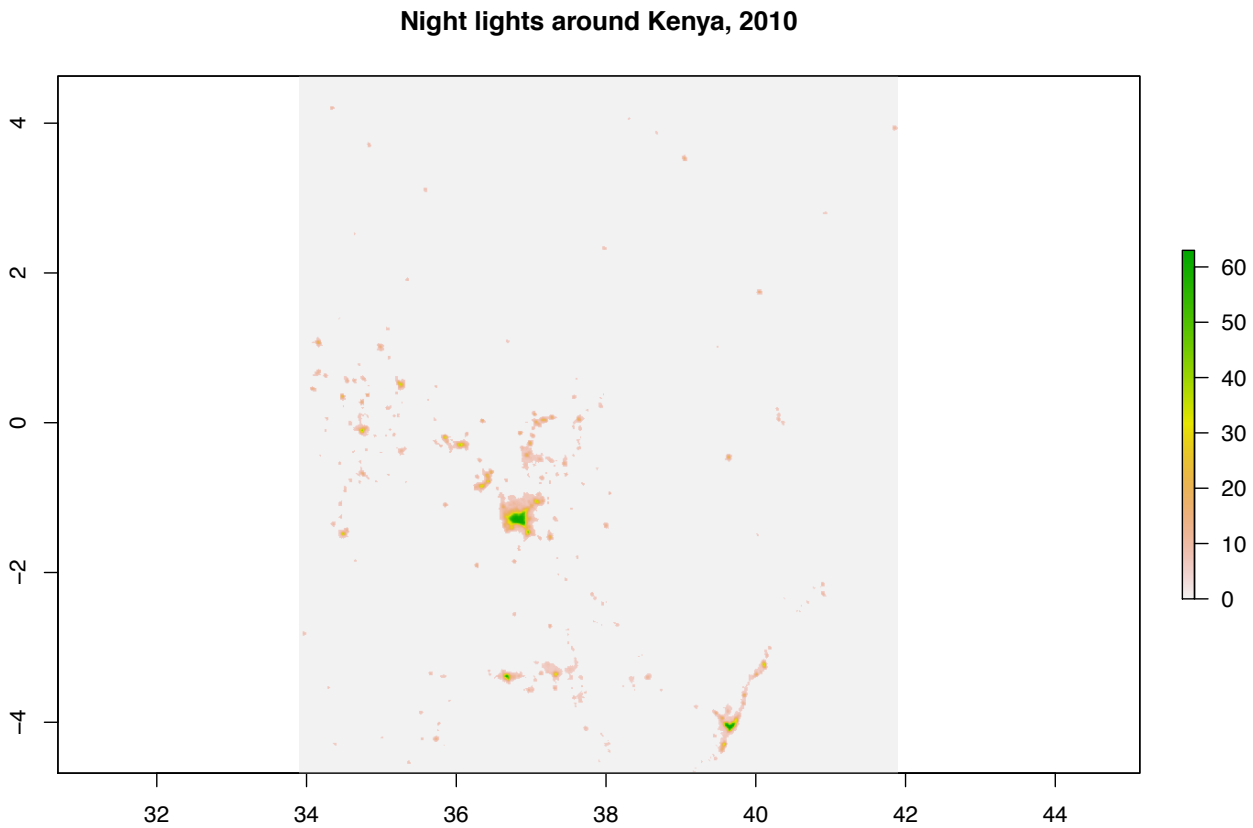
```
## [1] 1117  958    1
```

```
ncell(r.ken) == nrow(r.ken)*ncol(r.ken) # rectangle!
```

```
## [1] TRUE
```

Plotting rasters

```
plot(r.ken, main = "Night lights around Kenya, 2010")
```



Raster classes

Raster data often have more than one layer or band (think RGB).

- A `RasterLayer` consists of only one layer.
- A `RasterBrick` consists of multiple layers, which typically correspond to a single multispectral satellite file or a single multilayer object in memory.
- A `RasterStack` is similar to a RasterBrick but allows you to connect several raster objects stored in different files or multiple objects in memory. It is a list of `RasterLayer` objects with the same extent and resolution.

Note that operations on `RasterBrick` and `RasterStack` objects will typically return a `RasterBrick`.

Raster options

Processing rasters takes time, memory and hard disk. By default you do not see the progress of computations or know where intermediate files are saved. `rasterOptions()` allows you to change this.

Your preamble should include

```
rasterOptions(tmpdir = "./mytemp/",
              progress = "text",
              timer = TRUE)
tmpDir()
```

but you may also want to set `maxmemory`, i.e. the maximum number of cells to read into memory.

Raster subsetting

Raster subsetting is done with the base R operator `[`, which accepts a variety of inputs:

- Row-column indexing (e.g. `x[1,1]`)
- Cell IDs (e.g. `x[1]`)
- Coordinates
- Another raster or extent object

These will all return a vector or matrix of raster values. Use `x[..., drop = FALSE]` if you would like another raster.

There are also specialized functions that read rows (`getValues()`), blocks (`getValuesBlock()`), or entire neighborhoods (`getValuesFocal()`).

Selecting by coordinates or extent

Selecting by coordinates

```
id <- cellFromXY(r.ken, xy = c(37, -1))
r.ken[id]

##
## 7
```

Selecting by extent (could also create a `raster` object)

```
r.box <- extent(36, 38, -2, 0) #xmin, xmax, ymin, ymax
r.ken[r.box] %>% glimpse()

##  int [1:57600] 0 0 0 0 0 0 0 0 0 0 ...
```

Summarizing raster objects

These functions run "fast" even on disk:

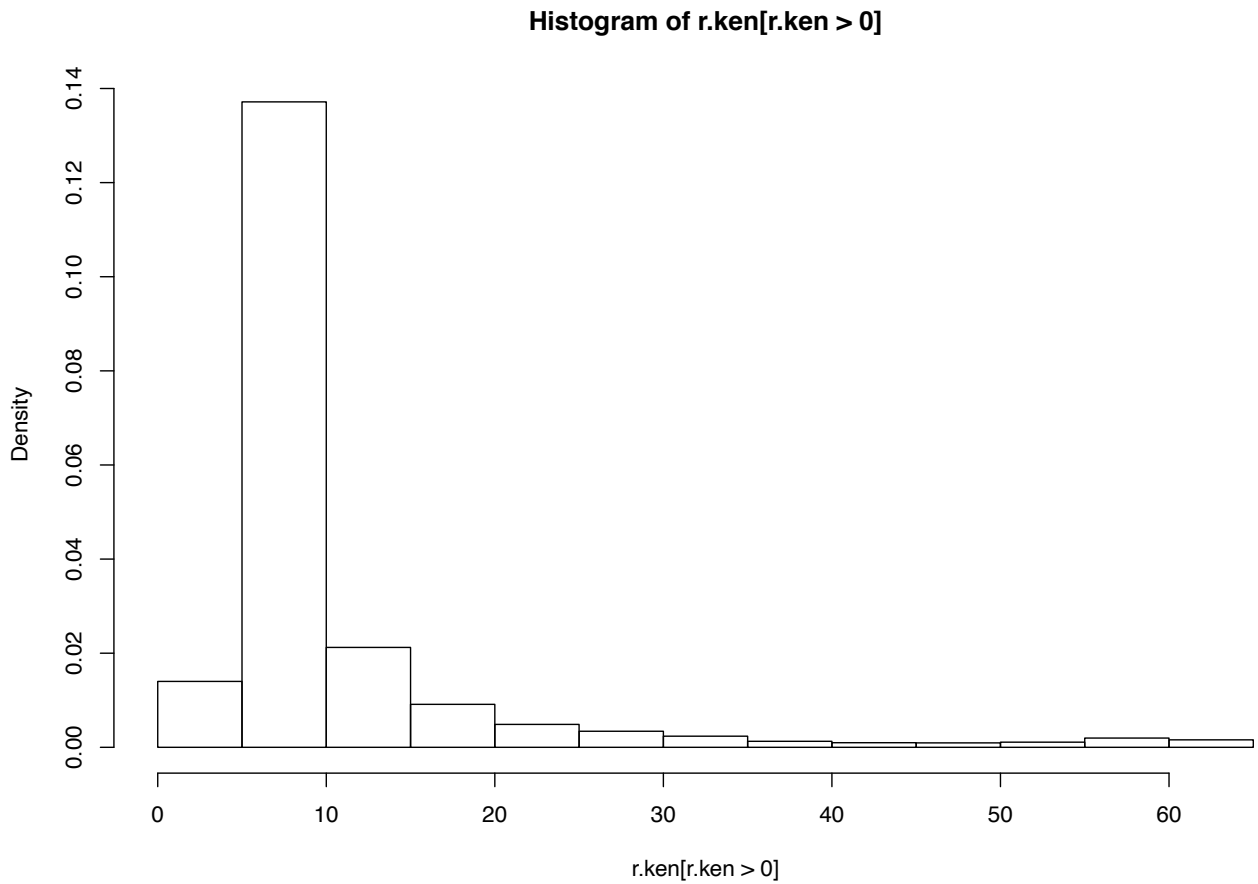
```
cellStats(r.ken, 'mean')

## [1] 0.1756261
```

```
quantile(r.ken) # all data

##   0%  25%  50%  75% 100%
##    0    0    0    0   63
```

```
quantile(r.ken[r.ken>0]) # bigger 0

##   0%  25%  50%  75% 100%
##    4    6    7   10   63
```

Histograms

```
hist(r.ken[r.ken>0], freq=F)
```



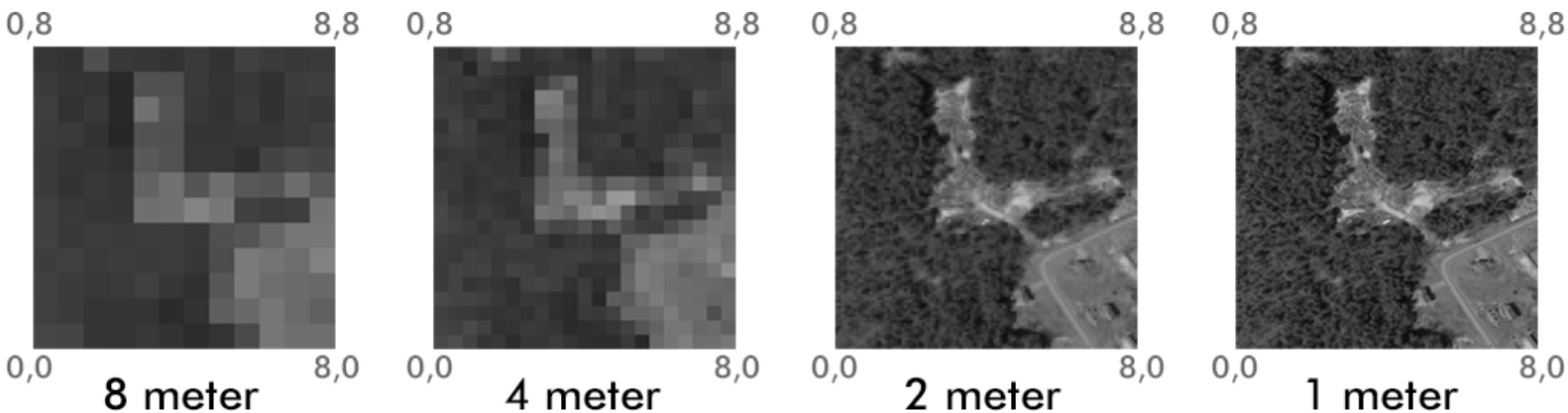
Aggregating and disaggregating

Changing the raster resolution can save disk space and computational time, or allow for the calculation of smaller areas.

To change the resolution of a continuous raster by any factor use:

```
aggregate(x, fact=2, fun=mean) # or sum etc
disaggregate(x, fact=2) # keeps values at lower res
```

This example aggregates from right to left:



Projections and transformations

Just like vector data, rasters have a CRS which can be set or transformed. Contrary to vector data, projecting raster data backwards and forwards is *not* lossless.

```
projection(r.ken)

## [1] "+proj=longlat +datum=WGS84 +no_defs ...

# projecting involves resampling to a new resolution
r.ken.moll <- projectRaster(r.ken,
                            crs = "+proj=moll",
                            res = 1000) # in km
projection(r.ken.moll)

## [1] "+proj=moll +ellps=WGS84"
```

Resampling

Re-projecting the data and aligning two rasters always involves resampling.

Resampling is the process of transferring values from one raster to another. These rasters may have

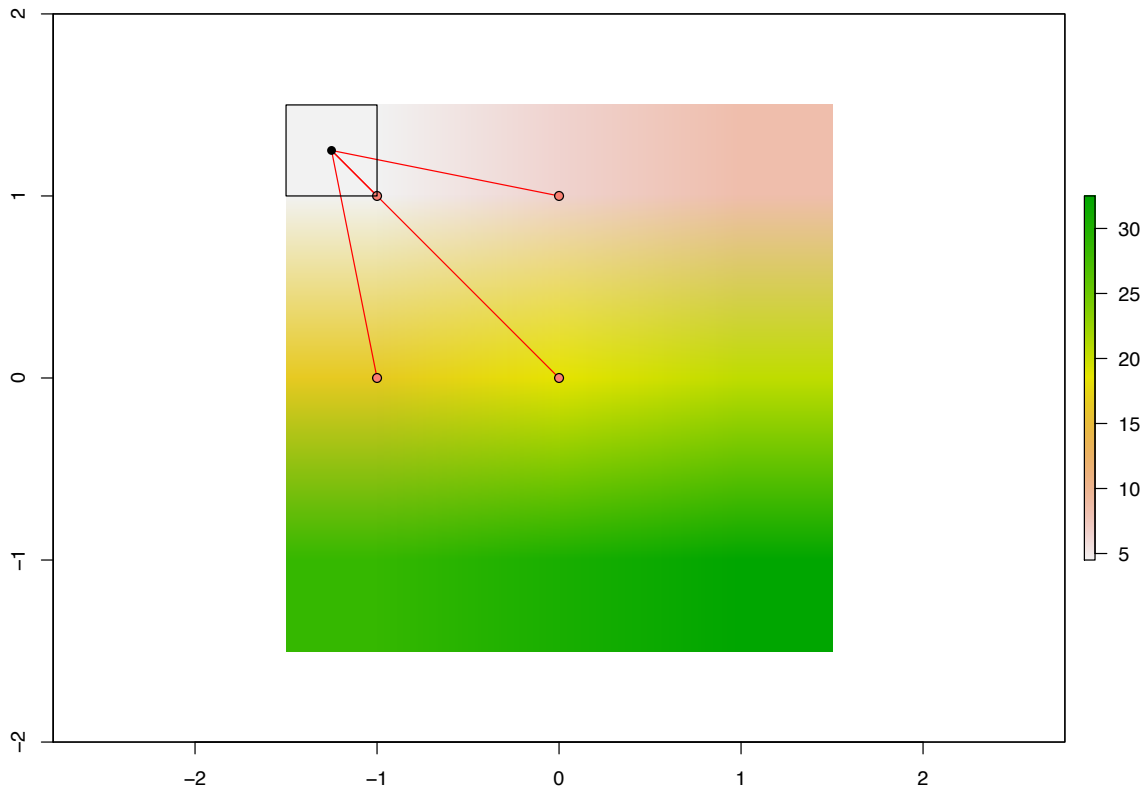
- a different origin (use `resample()`), or
- a different resolution (use `resample()`), or
- a different CRS (use `projectRaster()`).

There are two resampling algorithms:

1. Nearest neighbor resampling is used for categorical data and minor shifts in the underlying grid (differences in origin).
2. Bilinear resampling is used for continuous data.

Bilinear interpolation

Bilinear interpolation determines the new value of a cell by computing the *distance-weighted average of the values of the four nearest input cell centers*.



Map algebra

Map algebra is not matrix algebra. It uses a one-to-one locational correspondence but ignores the coordinate information. Rasters do not change their dimensions during mathematical operations. This is why you standardize your rasters first.

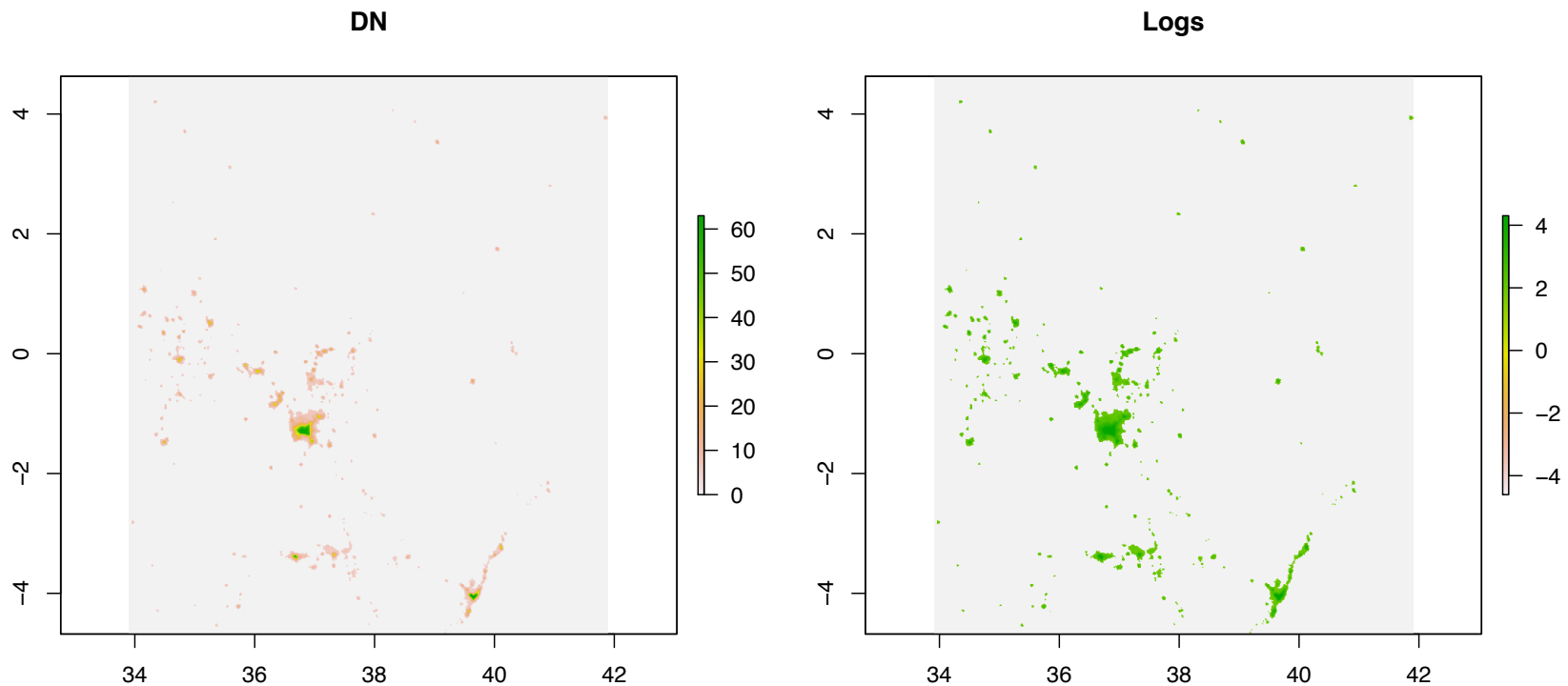
Map algebra divides raster operations into four subclasses:

- Local or per-cell operations (e.g. dividing two rasters cell-by-cell).
- Focal or neighborhood operations. Most often the output cell value is the result of a 3 x 3 input cell block.
- Zonal operations are similar to focal operations but any irregular size or shape is the basis of the calculations.
- Global or per-raster operations are essentially summary stats. We have covered them already.

Local operations

Raster algebra operations (+, −, /, etc) are local operations.

```
r.ken.ld <- log(r.ken/area(r.ken) + 0.01)
par(mfrow=c(1,2))
plot(r.ken, main="DN"); plot(r.ken.ld, main="Logs")
```



More raster algebra

All of these work as well (not run):

```
r <- r.ken
s <- r + 10
s <- sqrt(s)
s <- s * r + 5
r[] <- runif(ncell(r))
r <- round(r)
r <- r == 1
```

Subset and replace (not run):

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

Reclassification

Reclassification of multiple values can be done at once using the `reclassify()` function.

```
# (-Inf,0] to 0, (0,20] to 1, and (20,Inf] to 2
rc.ken <- reclassify(r.ken,
                     c(-Inf,0,0, 0,20,1, 20,Inf,2))

# take a look at the new distribution
freq(rc.ken)

##      value   count
## [1,]     0 1052674
## [2,]     1   15803
## [3,]     2    1609
```

Calc and overlay

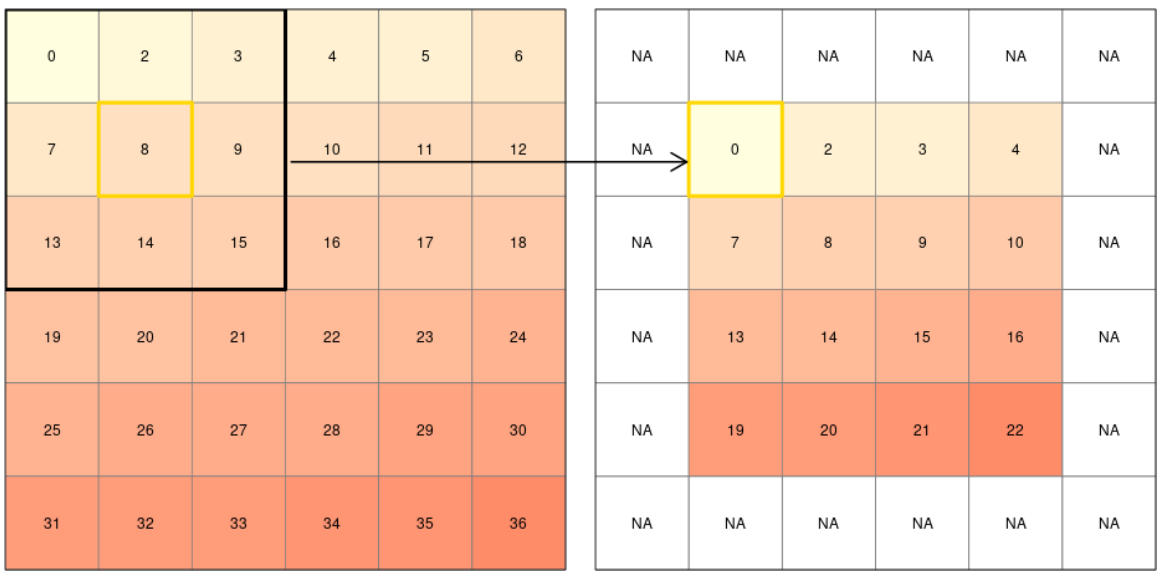
Using map algebra directly is inefficient when large rasters are stored on disk. The function `calc()` does block-wise processing for single rasters and `overlay()` does the same for multiple rasters.

```
# single layer: log of DN + 0.01
r1 <- calc(r.ken, fun = function(x) { log(x + 0.01) })

# two layers: log of DN/area + 0.01
r2 <- overlay(r.ken, area(r.ken),
              fun = function(x,y) { log(x/y + 0.01) })

# equal to previous result?
cellStats(r2 == r.ken.ld, "min")

## [1] 1
```

Focal operations

Focal operations are neighborhood statistics. You must supply a function (the default is sum) and a weighting matrix.

```
r.foc <- focal(r.ken, fun = min,
                    w = matrix(1, nrow = 3, ncol = 3))
```



Zonal operations

Zonal statistics summarize the values of one raster layer by values ("zones") of another raster of the same extent.

For large rasters, the function only takes predefined functions in characters: "mean", "sd", "min", "max", "sum" or "count".

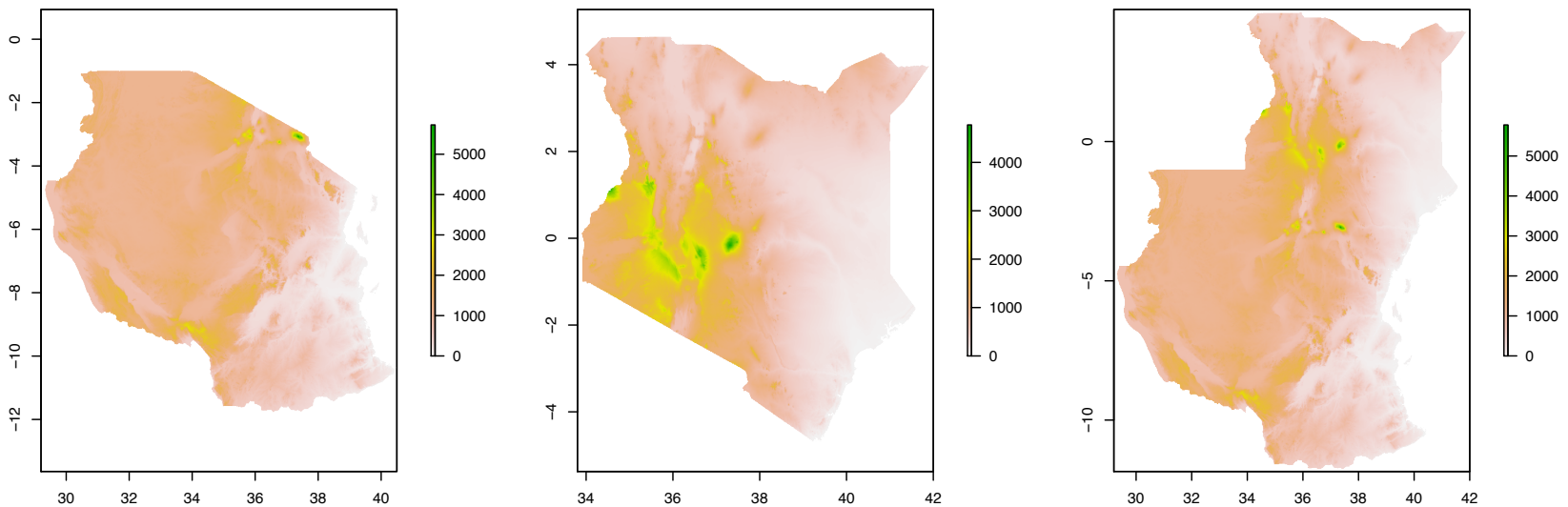
```
r.grd <- raster(ext = extent(r.ken), res = res(r.ken),
                vals = sample(1:3, ncell(r.ken),
                              replace = T))
# now run zonal with the random grid
zonal(r.ken, r.grd, fun='mean')
```

```
##      zone      mean
## [1,]    1 0.1761907
## [2,]    2 0.1709343
## [3,]    3 0.1797593
```

Merging rasters

The merge function lets you merge two or more rasters into a single new object. They must have the same resolution and origin.

```
r.tza.alt <- getData("alt", country = "TZA")
r.ken.alt <- getData("alt", country = "KEN")
r.both <- merge(r.tza.alt, r.ken.alt)
```



Vector-raster interactions

There are three main techniques of raster-vector interactions:

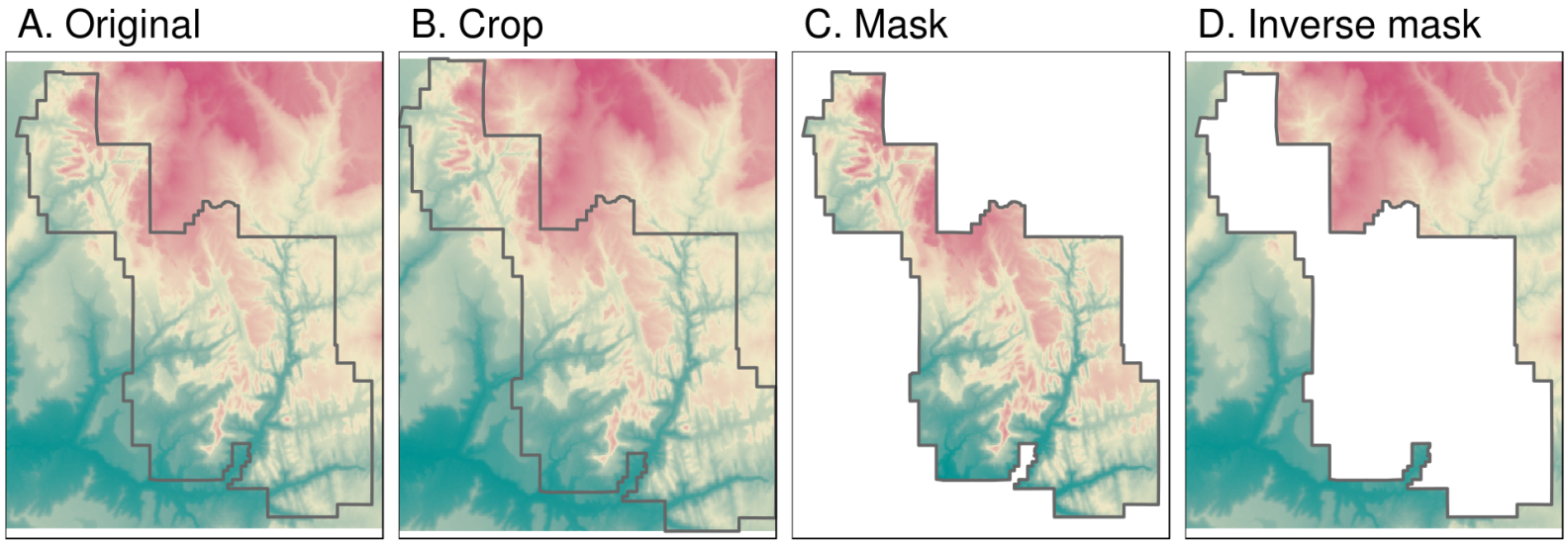
1. Raster cropping and masking using vector data
2. Extracting raster values using vector data
3. Raster-vector conversion

The `raster` package only deals with sp-objects and does not understand simple features. You can easily coerce one to the other:

```
# read the county boundaries
ken.l1.sf <- st_read("./data/kenya_counties.shp",
                     quiet = T)
# convert back and forth sf <--> sp
ken.l1.sp <- as(ken.l1.sf, "Spatial")
ken.l1.sf <- st_as_sf(ken.l1.sp)
```

Getting rasters into shape

Rasters are big rectangles which often well exceed our study area of interest. Cropping by a bounding box, `crop()`, or masking by vector data, `mask()`, deletes data outside the region of interest.



Masking is slow and often not necessary. Changing the settings of `mask()` yields in different results: `maskvalue = 0` sets all pixels that are outside to 0 and `inverse = T` masks everything inside.

Extracting

Raster extraction is the process of identifying and returning the values of a raster at specific locations of vector data.

Other GIS software often also call this type of extraction zonal statistics. It is probably the single most useful operation.

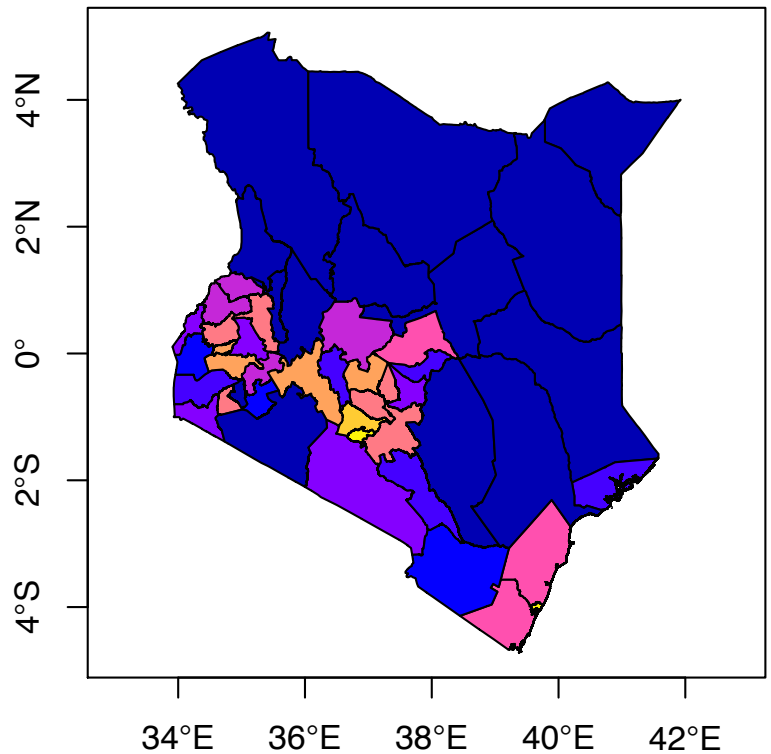
Let's examine the `extract()` function and its options:

```
extract(x, y, fun=NULL, na.rm=FALSE, weights=FALSE,
   normalizeWeights=TRUE, cellnumbers=FALSE,
   small=TRUE, df=FALSE, layer, nl,
   factors=FALSE, sp=FALSE, ...)
```

here `x` is a raster and `y` is a vector If you specify a function, such as `mean`, then `extract` returns summary statistics, if not, it returns all cells within the object. Extract gets masked by the `tidyverse`, call it with `raster::extract()`.

Extracting a data frame via polygons

```
ken.l1.ex <- raster::extract(r.ken, ken.l1.sp,
                             fun=mean, na.rm=T, df=T)
ken.l1.sf <- ken.l1.sf %>% mutate(ID = 1:nrow(.)) %>%
  left_join(ken.l1.ex, by = "ID")
```



Extracting from a raster stack

If you have several rasters of the same type you should extract all data at once.

```
# create stack and get sum of cells
s.ken <- stack(r.ken, area(r.ken))
s.ext <- raster::extract(s.ken, ken.l1.sp,
                         fun=sum, na.rm=T, df=T) %>%
  rename(sumlight = 2, sumarea = 3) %>% glimpse(50)

## Observations: 47
## Variables: 3
## $ ID       <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 1...
## $ sumlight <dbl> 203, 115, 1440, 447, 42, 848...
## $ sumarea  <dbl> 10897.642, 2443.665, 3018.01...
```

Vectorization

Vectorization is the conversion of raster objects into their representation in vector form. Two common conversions are raster to points and raster to polygons. Here are two examples:

```
# kill all zeros
r.ken.na <- reclassify(r.ken, c(-Inf,0,NA))

# to points
v.ken.pts <- rasterToPoints(
  r.ken.na, spatial = T) %>% st_as_sf()

# to poly and group by values
v.ken.pol <- rasterToPolygons(r.ken.na) %>% st_as_sf()
v.ken.pol <- v.ken.pol %>%
  group_by(layer) %>% summarize()
```

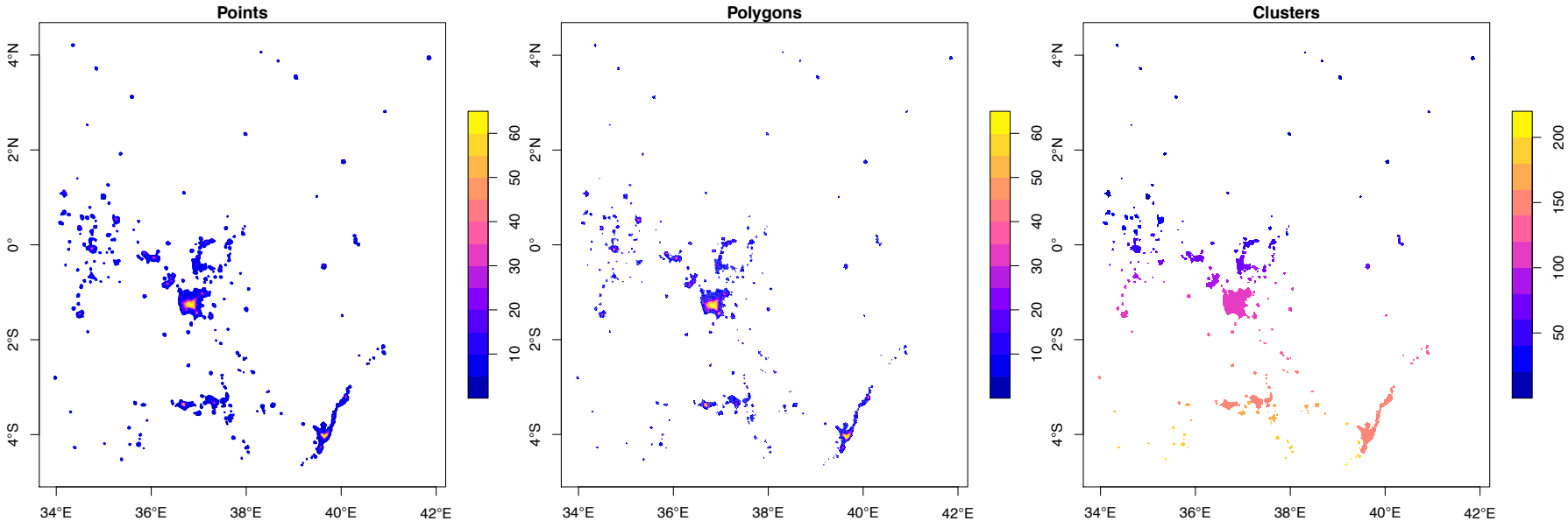
Detecting connected cells

Suppose that instead of converting all raster values into vector representations, we would like to detect clusters (or clumps) of connected cells.

For example, we might want to find all clusters of lit cells which might be cities, towns or villages.

```
# classify to binary raster
r.ken.bin <- reclassify(r.ken, rcl =
                          c(-Inf, 0, NA,
                            1, Inf, 1))

# find clusters and vectorize
v.ken.clu <- clump(r.ken.bin, gaps = F) %>%
  rasterToPolygons() %>% st_as_sf()
```

Three vectorized results



Rasterization

Rasterization is the conversion of vector objects into their representation in raster objects. It is the reverse case of vectorization. In my experience, its use cases are a bit limited.

Rasterization requires a reference raster with some origin, resolution and extent. It can be an empty dummy raster that is overwritten during the rasterization.

We can obtain the original light raster with 0 set to NA via

```
r.ken.alt <- rasterize(v.ken.pts, r.ken,
                       field = "layer", fun = sum)
cellStats(r.ken.alt, 'mean') ==
  cellStats(r.ken[r.ken>0, drop=F], 'mean')

## [1] TRUE
```