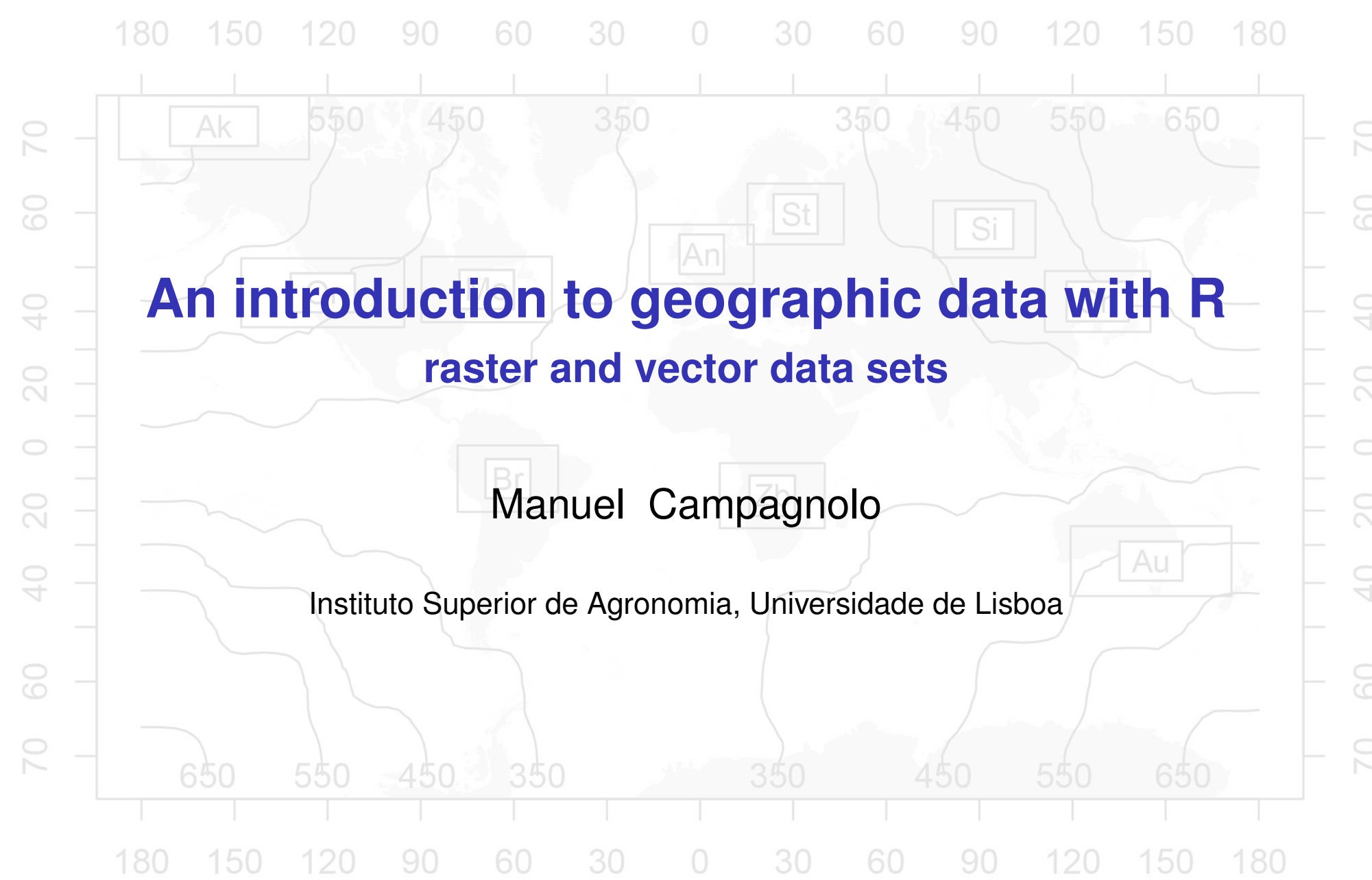



# An introduction to geographic data with R

## raster and vector data sets

Manuel Campagnolo

Instituto Superior de Agronomia, Universidade de Lisboa

## Processing geographic data with R

To do spatial analysis on geographic data, we first need to be able to:

1. read raster and vector data sets, e.g. **geotiff**, **shapefile**, or other file formats;
2. associate a coordinate reference system (CRS) to a data set and reproject it to a new CRS if necessary;
3. manipulate and explore data structures for geographic data in R;
4. if necessary, export the results as new geographic data sets.

Some useful web site links:

1. The Comprehensive R Archive Network's task view "Analysis of Spatial Data", by Roger Bivand
2. Edzer Pebesma and Roger Bivand, Classes and methods for spatial data in R, *R news*, 5/2 (2005) 9–14
3. Robert J. Hijmans, *Introduction to the raster package*, 2014

Reference:

1. Roger S. Bivand, Edzer Pebesma, Virgilio Gomez-Rubio, 2013. *Applied spatial data analysis with R*, 2nd edition. Springer, NY. www.asdar-book.org/

# R packages for geographic data

The following site updates regularly the many available packages for geographic and spatial processing:

> *The Comprehensive R Archive Network's task view "Analysis of Spatial Data", by Roger Bivand.*

The major packages we rely on are `sp`, `raster`, `rdal` and `rgeos`. These are necessary to read and write different file formats, convert them into R objects, and manipulate data structures for geographic data in R.

```
1 library(raster) # raster data sets
2 library(sp) # vector data sets
3 library(rgdal) # import/export data sets
4 library(rgeos) # spatial analysis of geographic data sets
```

# Geographic data sets

Geographic data sets encompass:

- geographic location, defined over some coordinate reference system (CRS), which can be *geographic* (latitude/longitude) or cartographic ($x$, $y$ coordinates on a plane);
- values associated to each location which can be continuous or categorical;
- an appropriate data format:
  - raster data sets tend to be used to represent variables which value is known over a continuous extension of space, like surface reflectance or temperature;
  - vector data sets tend to be used to represent features at selected locations.

# Raster data sets

# Read raster data sets

Package `raster` provides many classes and functions for raster data sets. Class `RasterLayer` supports single band images, and classes `RasterBrick` and `RasterStack` support multi band images, where each band has the same extension and spatial resolution. `RasterStack` is more flexible, and `RasterBrick` is more efficient.

In this example, we use function `stack` to read satellite Landsat 7 images over a 30 $\times$ 30 km$^2$ area in Ribatejo, an intensive agricultural region of Portugal. Each band is available as a distinct GeoTIFF file, so it needs to be first grouped as a `RasterStack` object, and then it can be converted into a `RasterBrick` for further processing in R.

```
1 fichs <- list.files(pattern="band[1-7]") # list of file names
2 s <- stack(as.list(fichs)) # read raster data sets
3 s # data set summary
4 b <- brick(s) # create RasterBrick object
5 b # idem
6 nlayers(b) # number of layer in the RasterBrick object
```

## Access pixel values with `values`

Function `values` returns either values of a `RasterLayer` (single band) as a vector or values of a `RasterBrick` (multiple bands) as a `matrix` object. In this last case, each row represents one pixel, and each column represents a band. "No data" values are represented by `NA`.

In the following example, one plots the histograms for all bands. To plot each histogram one can either use `hist(b[[i]])` or `hist(values(b)[,i])` since both represent all values in the *i*th band.

However, to determine the range of values for all bands one has to do `range(values(b))` since `range(b)` return a new `RasterBrick` object with the minimum and maximum values for each pixel.

```
1 par(mfrow=c(3,2),mar=c(4,4,2,2));
2 for (i in 1:nlayers(b))
3   hist(b[[i]], xlim=range(values(b),na.rm=TRUE),breaks=seq(0,1,
      length.out=20),xlab="reflectance")
```

# Color composite with `plotRGB`

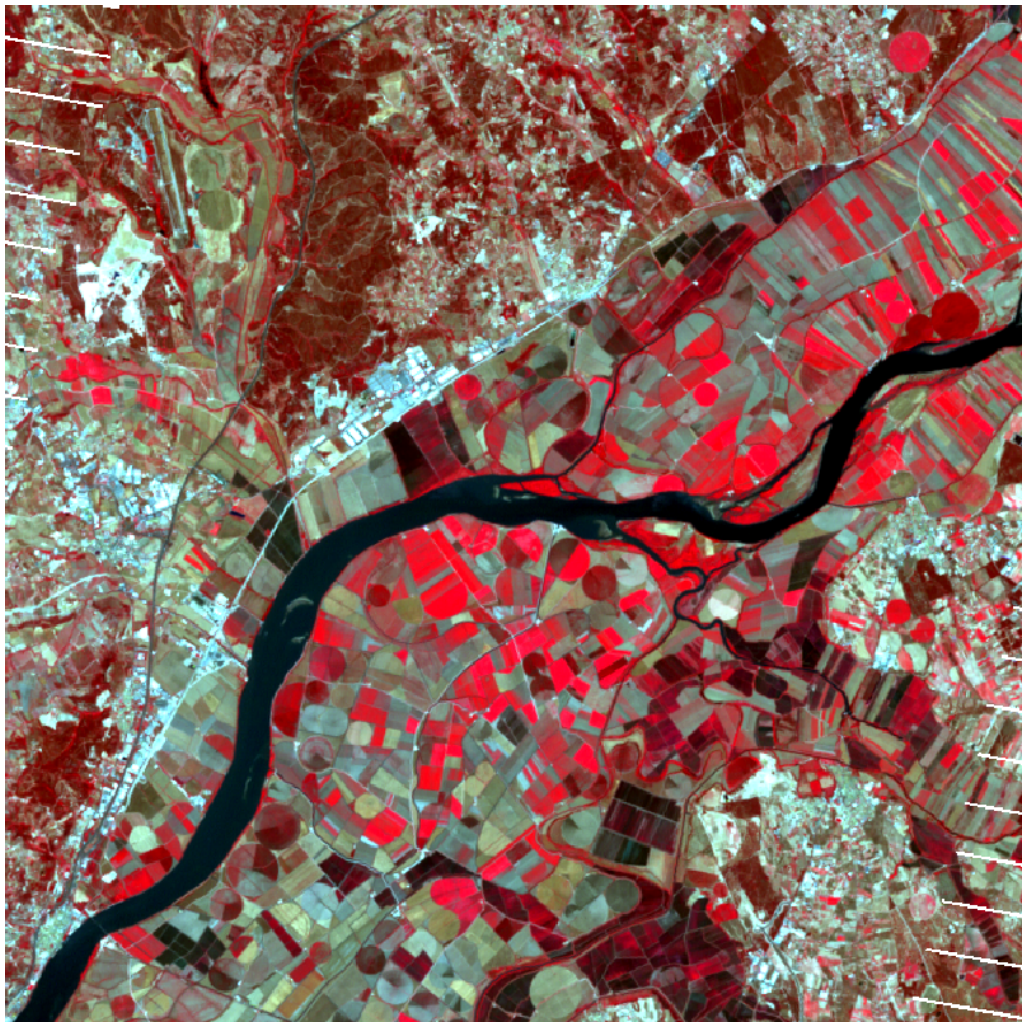
```
box <-extent(c(500000,  520000, 4310000, 4330000))
```

Function `plotRGB` defines color composites of multilayer images. Here, we define a `RGB=432` composite.

```
par(mfrow=c(1,1),mar=
    rep(0,4))
plotRGB(b,r=4,g=3,b=2,
    stretch="lin",ext=
    box)
```

The option `stretch="lin"` enhances constrast.

The plot's extent is defined by argument `ext`.



## Raster algebra and manipulation of raster values

The Normalized Difference Vegetation Index (NDVI) can be computed with arithmetic operations over the multilayer Landsat image:

```
ndvi <- (b$band4 - b$band3)/(b$band4 + b$band3)
```

Many R operations for matrices can be applied to raster layers. In this example, one computes the proportion of `NA` values in `ndvi`:

```
ncell(ndvi[is.na(ndvi)])/ncell(ndvi) # length could be used
    instead of ncell
```

The values of the pixel can be modified as in the following example:

```
ndvi[ndvi<0] <- 0 # replace negative ndvi values by 0
```

Applying a function to a `RasterBrick` or `RasterStack` returns a new `raster` object:

```
min(b, na.rm=TRUE) # RasterLayer with the minimum value at each
    pixel
range(crop(b,box), na.rm=TRUE) # returns RasterBrick with two
    layers
```

# Raster manipulations

The function `cellStats` returns statistics for each layer of a `RasterBrick` or `RasterStack` object:

```
cellStats(b,"mean") # returns a vector of length 6, with the
    averages for each band:
```

Functions `calc` and `overlay` can also be used to apply some function to a multilayer image: `fun` has to match the layers of the `RasterBrick` or `RasterStack` inputs.

```
median(crop(b,box)) # error
b.median<-overlay(crop(b,box),fun=median) # it works, but it is
     slow...
```

To obtain the coordinates as a 2-column matrix, and add a 3rd column with the pixel values:

```
xyz<-cbind(coordinates(ndvi),values(ndvi))
```

Exercise: what is the location with the highest NDVI value?

# Access very high resolution imagery through R

Several functions are available to download very high resolution imagery:

- `GetMap` from package `RgoogleMaps`; only to create a background image;
- `gmap` from package `dismo`; this function returns a `raster` object, either a `RasterLayer` or a `RasterBrick` that can be processed in R.

Example:

```
1 library(RgoogleMaps)
2 TA <- GetMap(center="Tapada da Ajuda", zoom=13, maptype="
      satellite") #read dsatellite image
3 par(mfrow=c(1,1),mar=c(1,1,1,1))
4 PlotOnStaticMap(TA) # plot
5 bb <- TA$BBOX # extension (lat/long)
6 lat<-(bb$ll[1]+bb$ur[1])/2; long<-(bb$ll[2]+bb$ur[2])/2
7 text(long,lat,"Welcome to ISA",cex=4,col="yellow") # add text
```

## Interactive tools in R

In the following example, we use function `gmap` from package `dismo` to access very high resolution (VHR) imagery. Then, we superimpose the high resolution imagery with the earlier Landsat 7 color composite.

The first thing is to determine the location of the VHR image we want to download. This can be done selecting a location with `locator`:

```
plotRGB(b,r=4,g=3,b=2,stretch="lin",ext=box)
xy<-locator(n=1) # to select n points over the current plot
```

Alternatively, one can use `drawExtent()` to determine an `extent` object:

```
e<-drawExtent() # drawExtent returns an extent object
```

The other possibility is to define the point or the extension explicitly:

```
e<-extent(c(513614,514211,4317780,4318205))
plotRGB(b,r=4,g=3,b=2,stretch="lin",ext=e)
```

Those coordinates are in the coordinate reference system of `b`.

## Coordinate reference system (CRS)

Check that `b` has CRS UTM, zone 29:

```
b@crs # or crs(b)
```

The string

```
"+proj=utm +zone=29 +datum=WGS84 +units=m
+no_defs +ellps=WGS84 +towgs84=0,0,0"
```

is called a proj.4 string and describes the CRS: the projection is universal transverse mercator at zone 29, and the reference datum is WGS84. The coordinate units are meters. Finally, `+ellps` indicates the ellipsoid, and `+towgs84` contains parameters for datum transformation.

This CRS can also be referred to by its `epsg` code:

```
utm.29<-"+init=epsg:32629" # string that can be interpreted as
    a CRS
```

CRS can be searched at https://epsg-registry.org/ or at http://spatialreference.org/.

## Coordinate reference system (CRS)

The difficulty here is that we need to know the latitude and longitude of the location where we want to obtain a VHR image. Towards that end, we need to convert our UTM, zone 29, coordinates into latitude and longitude.

This can de done by reprojecting the Landsat image to a geographic (*i.e.* lat/long) CRS:

```
cb<-crop(b,e) # crops b to the extent defined by e
wgs84<-"+proj=longlat +ellps=WGS84 +datum=WGS84" # define new lat/long CRS
b.wgs<-projectRaster(cb,crs=CRS(wgs84)) # returns raster with values and lat/long coordinates
# or
b.wgs<-projectExtent(cb,crs=CRS(wgs84)) # returns raster with no values and lat/long coordinates
e.wgs<-extent(b.wgs) # or b.wgs@extent
```

As a result, `e.wgs` are the coordinates (in lat/long) of the extent `e` originally in UTM, zone 29.

# Download very high resolution imagery with `gmap`

Now that we know the extent (in latitude/longitude) for the VHR image we want to download, we can use `gmap` from package `dismo`. The option `scale=2` is to download a raster with the highest possible resolution. Note that the resolution of the image returned by `gmap` also depends on the extent: for a smaller extent the resolution is better.

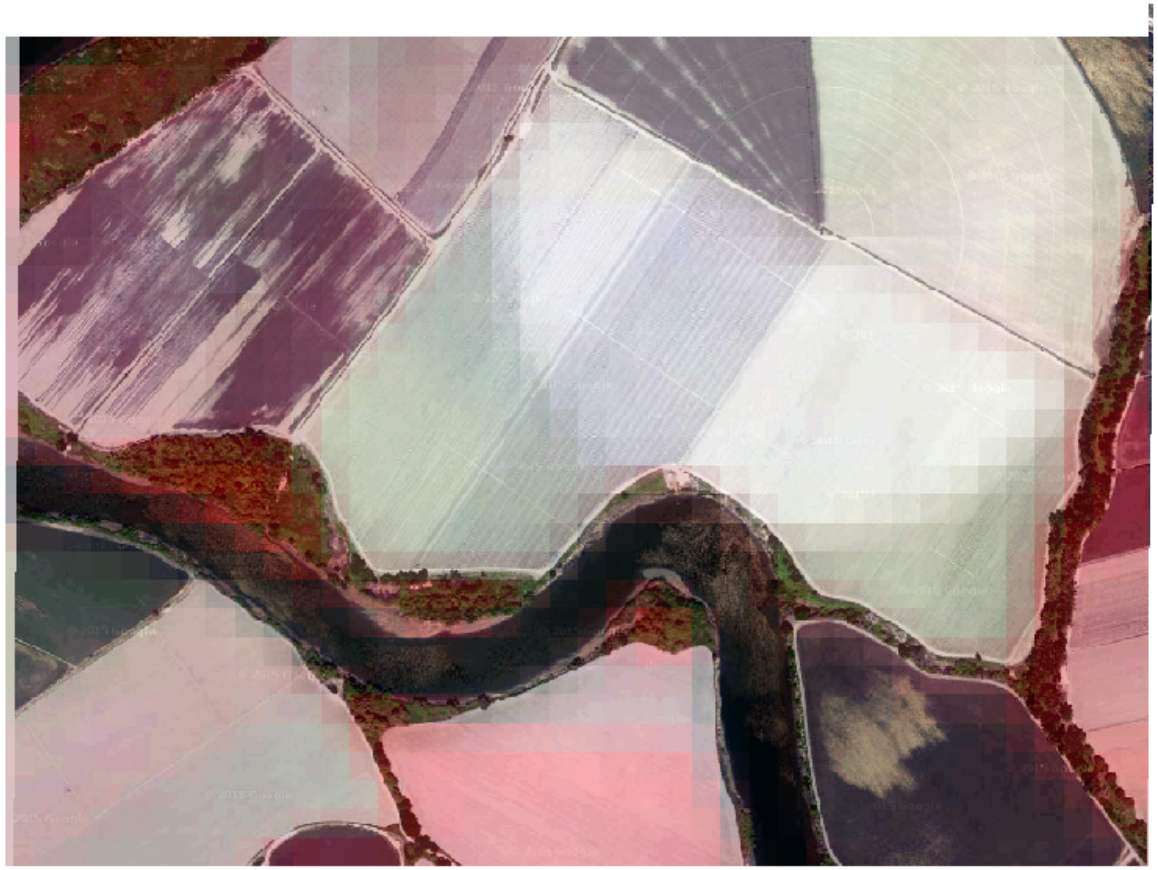
```
1 library(dismo)
2 gm.wgs<-gmap(e.wgs, type="satellite", lonlat=TRUE, rgb=TRUE, scale
      =2)
```

Finally, we convert `gm.wgs` back to UTM, zone 29, so it can be superimposed over the Landsat original image:

```
1 gm.utm<-projectRaster(gm.wgs, crs=utm.29)
```

# Superimpose two images

```
1 par(mfrow=c(1,1))
2 # VHR image
3 plotRGB(gm.utm, r=3,g
      =2,b=1)
4 # Landsat RGB=432
      composite
5 plotRGB(b, r=4,g=3,b=2,
      stretch="lin",ext=
      e,alpha=100,add=
      TRUE)
```



The `alpha` argument controls transparency: it ranges from 0 to 255.

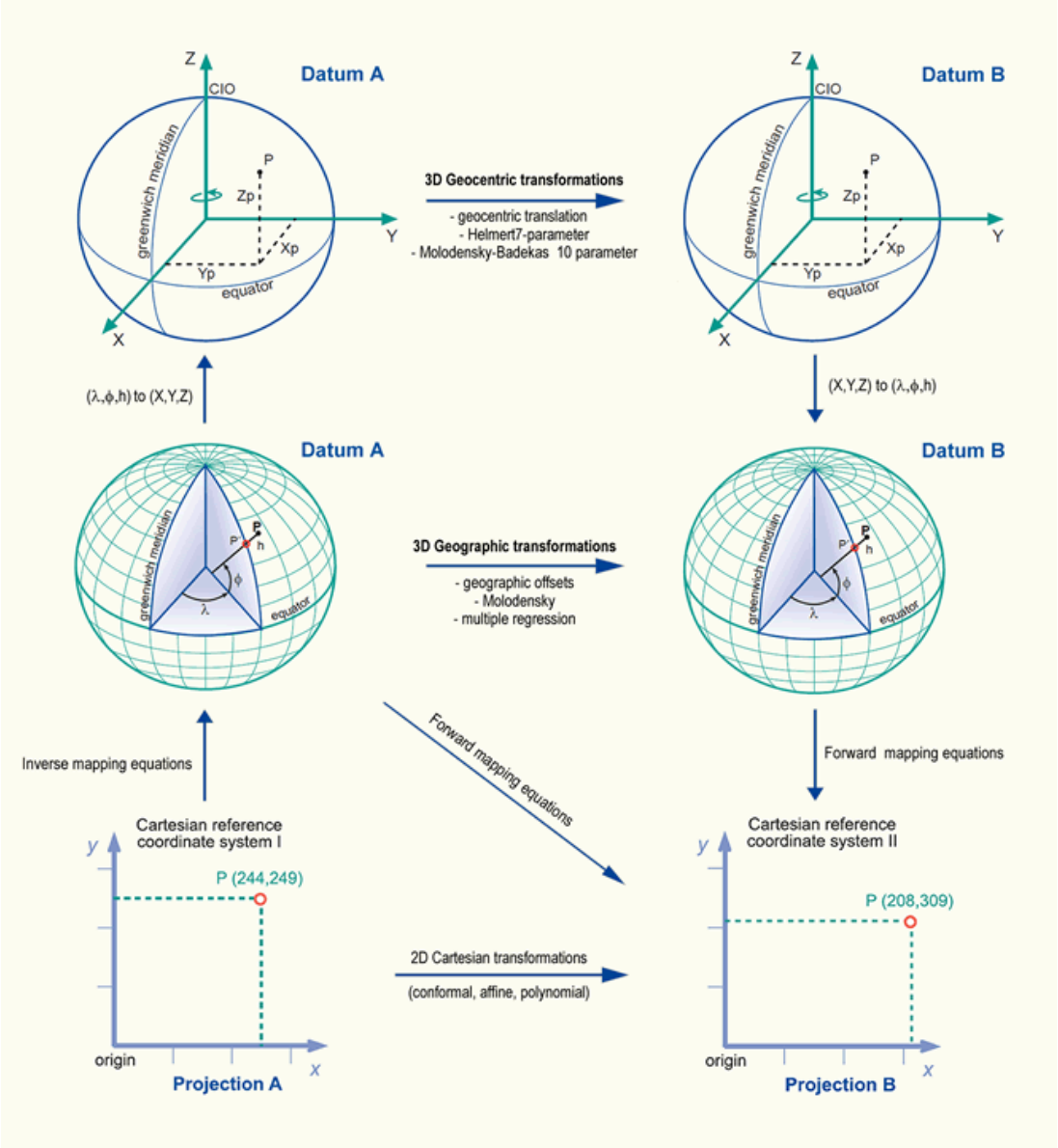
Exercise: execute the above commands but replace `ext=e` by `ext=2*e`. Recall that `e` is an object of class `extent` that was used to define the extension of the downloaded VHR image.

# Some brief notes on coordinate reference systems

# A few notions on coordinate reference systems and projections

There are three basic systems of coordinates used in geography:

1. 3D Cartesian coordinates: used for geodesy;
2. Geographic coordinates (lat/long): reference coordinates to store information; these are the coordinates gathered by GPS devices;
3. Cartographic planar coordinates $(x, y)$.

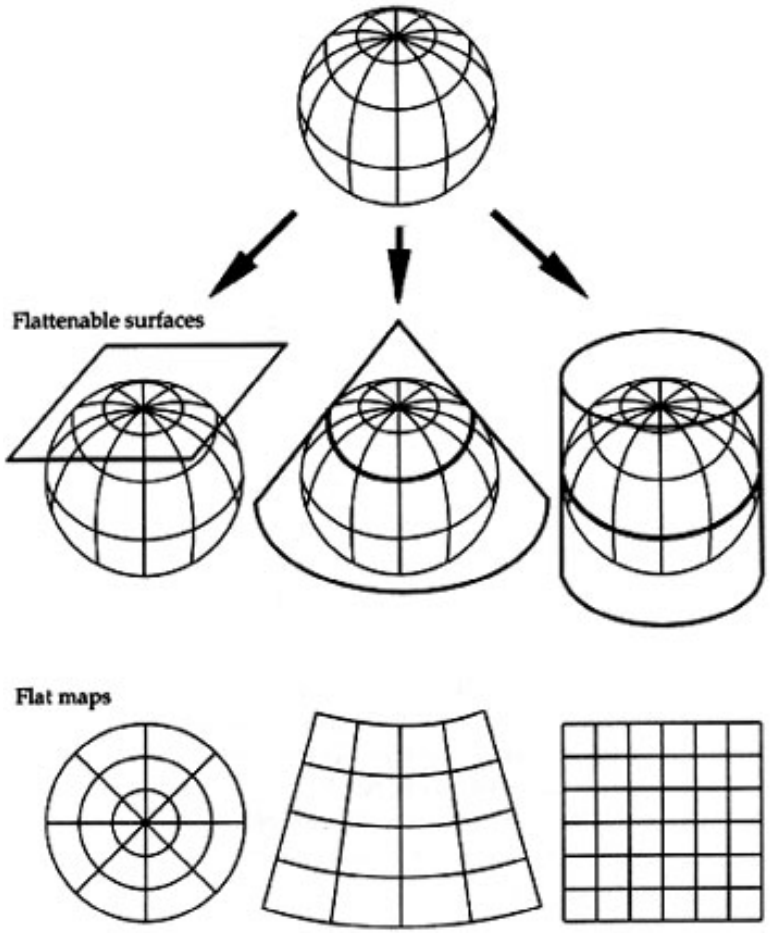


# Map projections

A map projection is a method to produce all or part of a spheroid (ellipsoid of revolution) on a flat surface.

Even if map projections are not in general perfect geometric projections, it is convenient to classify them according to the most similar geometric projection, which can be:

- azimuthal,
- conic,
- cylindrical



## Decrypting PROJ.4 descriptions

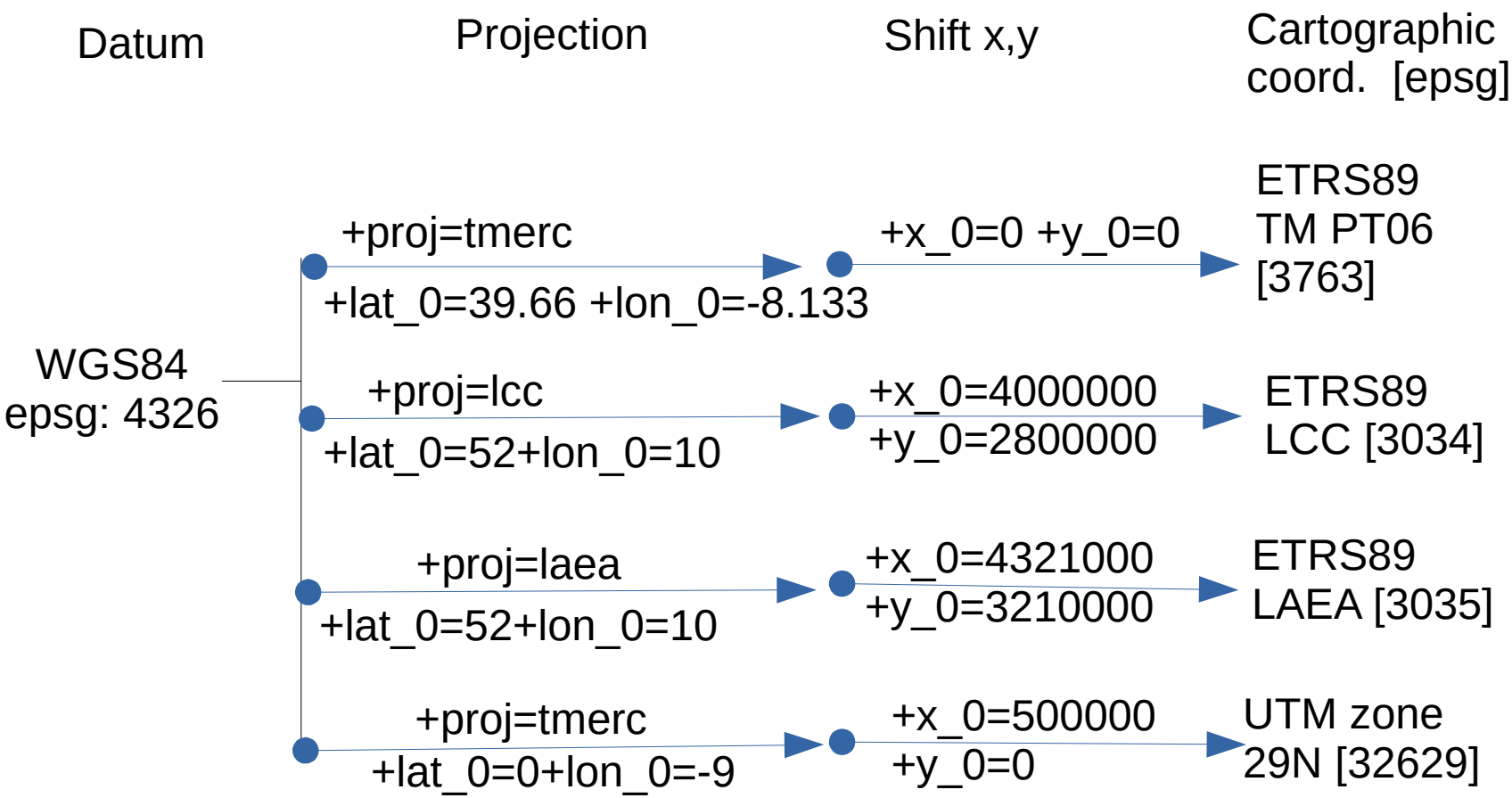
Consider the following proj.4 description of a CRS:

```
+proj=laea +lat_0=52 +lon_0=10 +x_0=4321000
+y_0=3210000 +ellps=GRS80 +towgs84=0,0,0,0,0,0,0
+units=m +no_defs
```

The parameters of the CRS are:

1. `+proj=laea`: Lambert azimuthal equal-area map projection;
2. `+lat_0=52 +lon_0=10`: geographic coordinates of the origin of the projection;
3. `+x_0=4321000 +y_0=3210000`: false easting and false northing, i.e. distances (m) from the origin (x=0,y=0) of the cartographic coordinates to the origin of the projection;
4. `+ellps=GRS80`: ellipsoid name;
5. `+towgs84=0,0,0,0,0,0,0` : datum transformation parameters to WGS84;
6. `+units=m`: units of projection coordinates

# Some other CRS examples




A few CRS that are used in modern cartography for the EU and for Portugal in particular (ETRS89-TM-PT06). LCC stands for Lambert conformal conic, and LAEA stands for Lambert azimuthal equal-area.

# Some other CRS examples (cont')

The CRS in the previous slide have the following proj.4 descriptions:

**epsg:3763**
```
+proj=tmerc +lat_0=39.66825833333333
+lon_0=-8.133108333333334
+k=1 +x_0=0 +y_0=0
+ellps=GRS80 +units=m +no_defs
```

**epsg:3034**
```
+proj=lcc +lat_1=35 +lat_2=65
+x_0=4000000 +y_0=2800000
+ellps=GRS80 +units=m +no_defs;
```

**epsg:3035**
```
+proj=laea +lat_0=52 +lon_0=10
+x_0=4321000 +y_0=3210000
+ellps=GRS80 +units=m +no_defs;
```

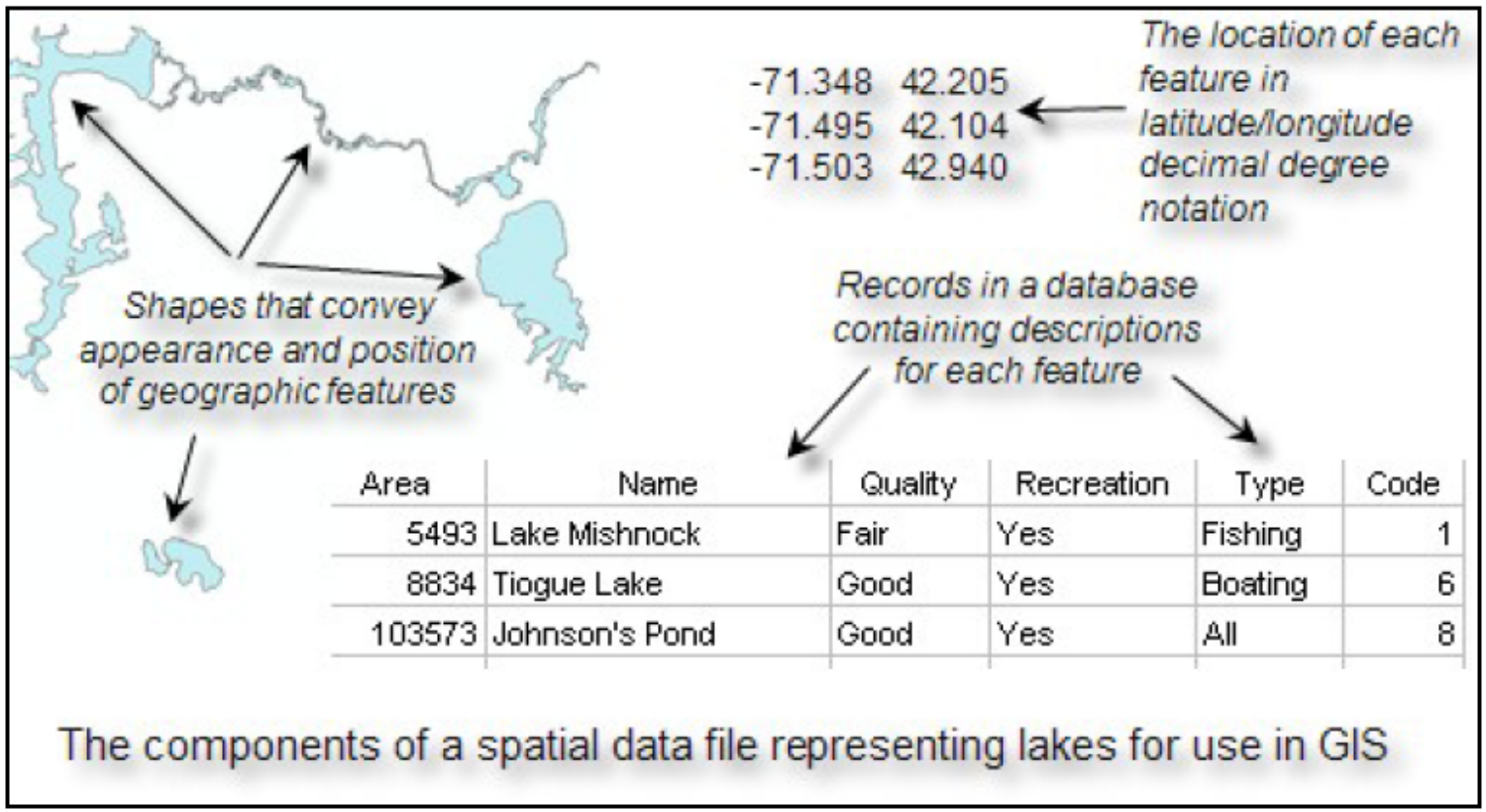
**epsg:32629**
```
+proj=utm +zone=29 +ellps=WGS84
+datum=WGS84 +units=m +no_defs.
```

# Vector data sets

# Features and attributes

A geographic feature is a digital representation of a real world phenomenon. A feature class is a set of features with a common structure and theme. In general, feature classes include

*georeferenced geometric objects + attributes.*




Source: `http://gisatbrown.typepad.com/gis/files/spatialdatafiles.pdf`

## Standards for analysis of georeferenced geometric objects

The OGC OpenGIS Implementation Standard for Geographic Information / ISO 19125 defines:

**Geometric objects** which can be of type point, line, polygon, multi-point, etc, and are associated to a given Coordinate Reference System;

**Methods on geometric objects** return properties like dimension, boundary, area, centroid, etc;

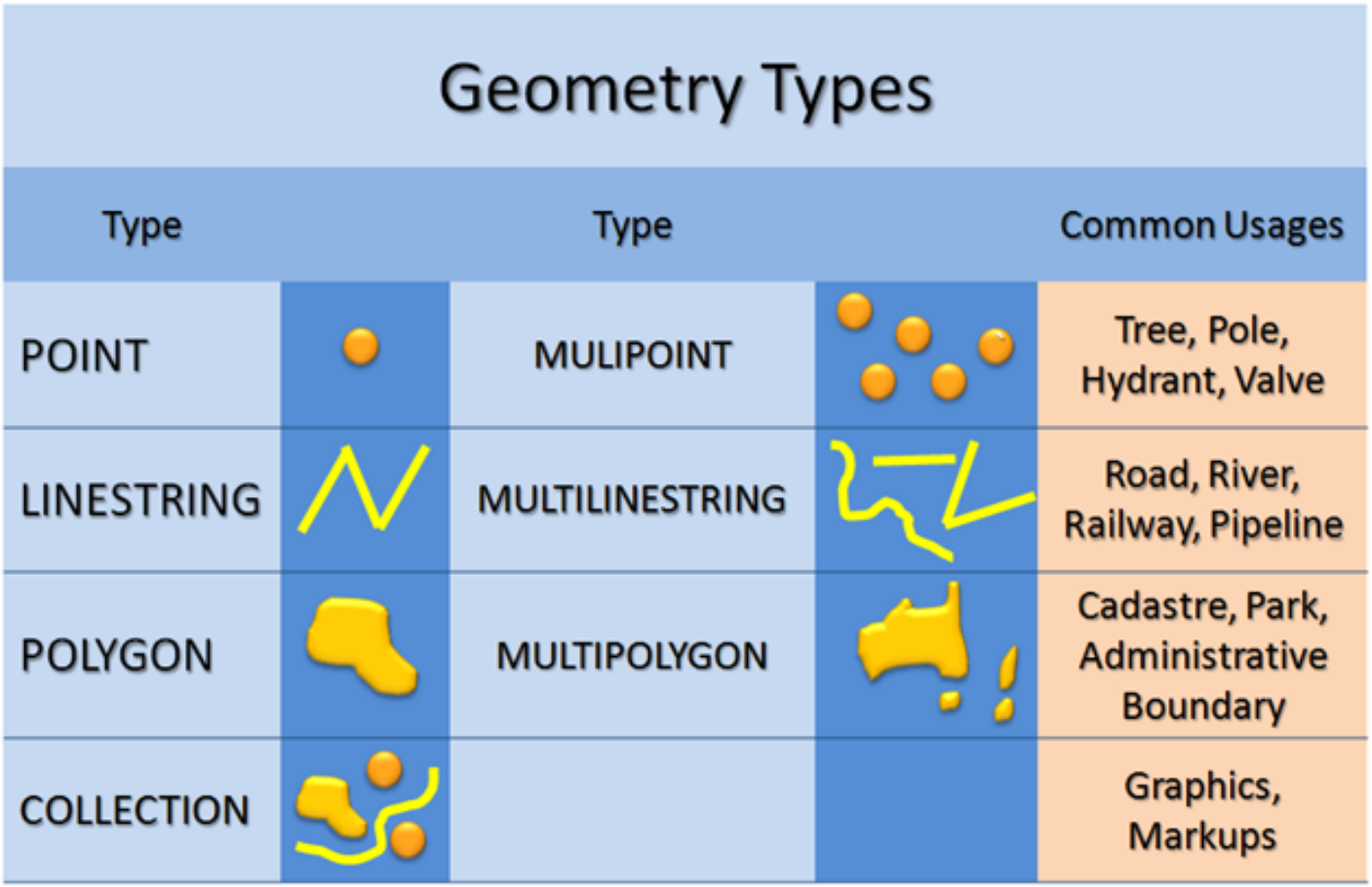
**Methods for testing spatial relations between geometric objects** equals, disjoint, intersects, touches, crosses, within, contains, overlaps and relate, which returns **TRUE** or **FALSE** (see Slide 50);

**Methods that support spatial analysis** distance, which returns a **distance**, and buffer, convex hull, intersection, union, difference, and symmetric difference, which returns **new geometric objects** (see Slide 59).

Source: `www.opengeospatial.org/standards/sfa`

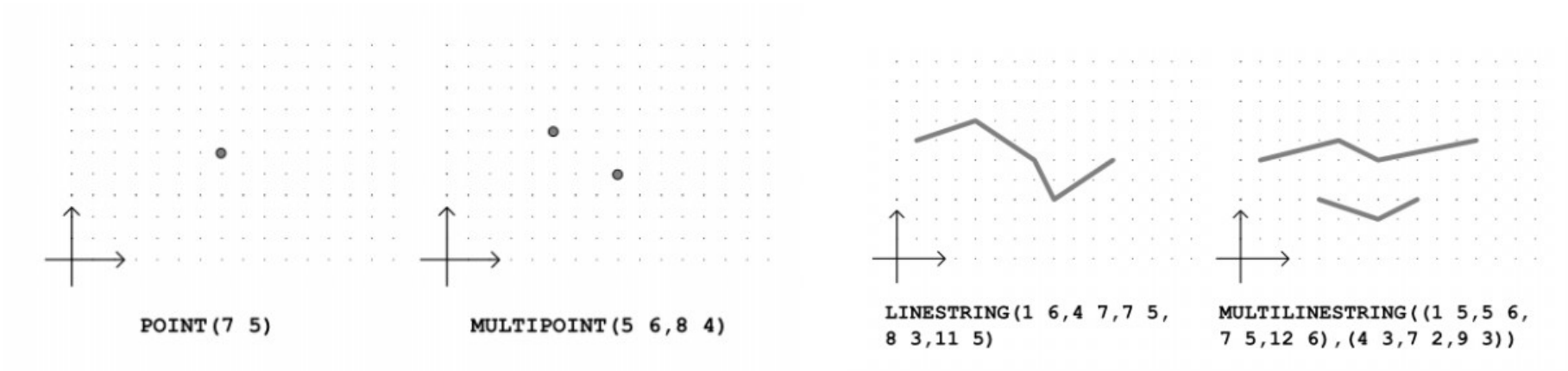
# Geometric objects

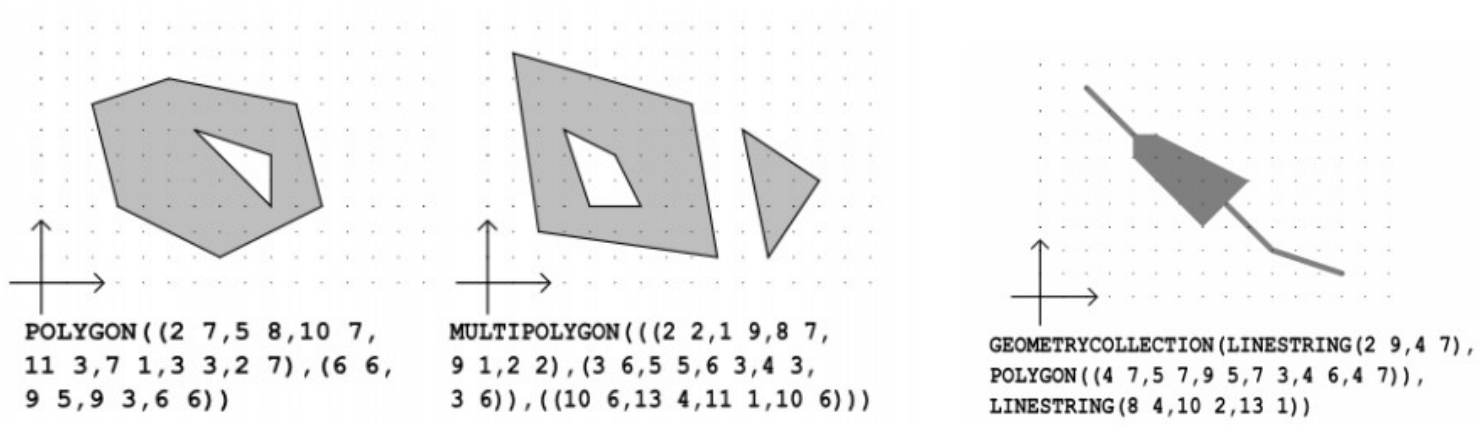
A geometric object is a spatial object representing locations with respect to a given CRS. A collection of geometric objects is called a geometry. Therefore, a **feature class is spatially represented by a geometry**.



# Geometric objects (cont')

The basic and composed geometry types are described by their coordinates in the CRS: represented below are geometric objects of dimension 0 (points), 1 (linestrings), and 2 (polygons). A *geometry collection* can contain objects of of different dimensions.



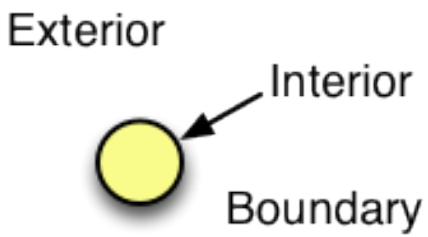



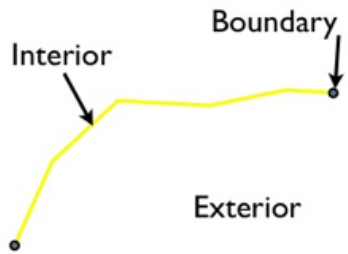
# Some basic properties of geometric objects: dimension, interior, boundary, exterior

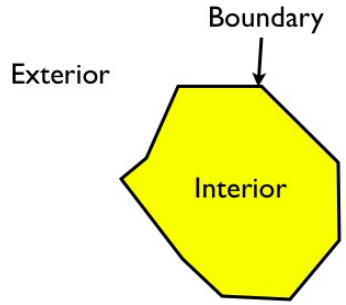
Points and Multipoints have dimension 0,
Curves like LineStrings and MultiLineStings have dimension 1,
Surfaces like Polygons and MultiPolygons have dimension 2.

Geometric objects have an **I**nterior, a **B**oundary and an **E**xterior:

1. If **p** is a point, then $\mathbf{I}(\mathbf{p}) = \mathbf{p}$, $\mathbf{B}(\mathbf{p})$ is empty, and the exterior $\mathbf{E}(\mathbf{p})$ are all the points not in **p**;
2. If **L** is a curve, then $\mathbf{B}(\mathbf{L})$ are the ends of the curve, $\mathbf{I}(\mathbf{L})$ are all the points of the curve except the ends, and the exterior $\mathbf{E}(\mathbf{L})$ are the remaining points;
3. If **P** is a surface, $\mathbf{B}(\mathbf{P})$ is the boundary, $\mathbf{I}(S)$ is the set of points of **P** which are not on the boundary and $\mathbf{E}(\mathbf{P})$ are the remaining points.



## *singlepart* and *multipart* geometric objects

A geometric object can be either *singlepart* or *multipart*.

1. point : one single feature can be represented as
   - a single point (*singlepart*);
   - more than one point (*multipart*).
2. curve : one single feature can be represented as
   - one single linestring (*singlepart*); or
   - a set of linestrings (*multipart*).
3. surface : each feature may have or not holes. Each feature is represented by:
   - one single polygon (*singlepart*) with no holes in its interior;
   - one single polygon with one or more holes in its interior (*singlepart*);
   - more than one polygon with or without holes in their interiors (*multipart*).

   Each hole is also represented by a polygon. A *singlepart* surface, with or without holes, is always spatially connected, i.e. any two points in its interior can be connected. Hence, any non-connected feature must be represented by a *multipart* surface.

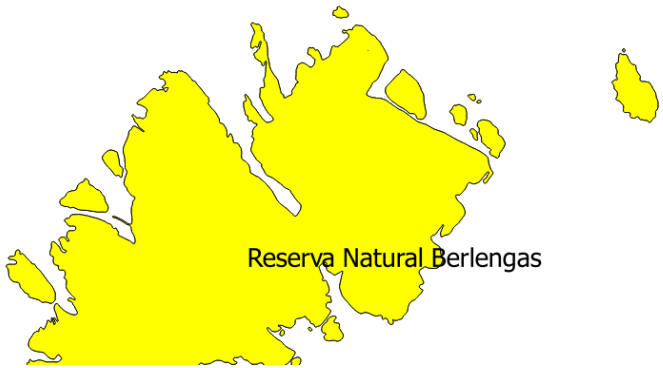
# *singlepart* e *multipart* geometric objects: examples

The following geometric objects represent protected areas in Portugal (ICNF, 2014):

*singlepart* geometric object with one part and no holes


*singlepart* geometric object with one part and 5 holes. Note that it is **spatially connected**.


*multipart* object represented by several polygons, which is not spatially connected.



## Feature classes in R: the package `sp`

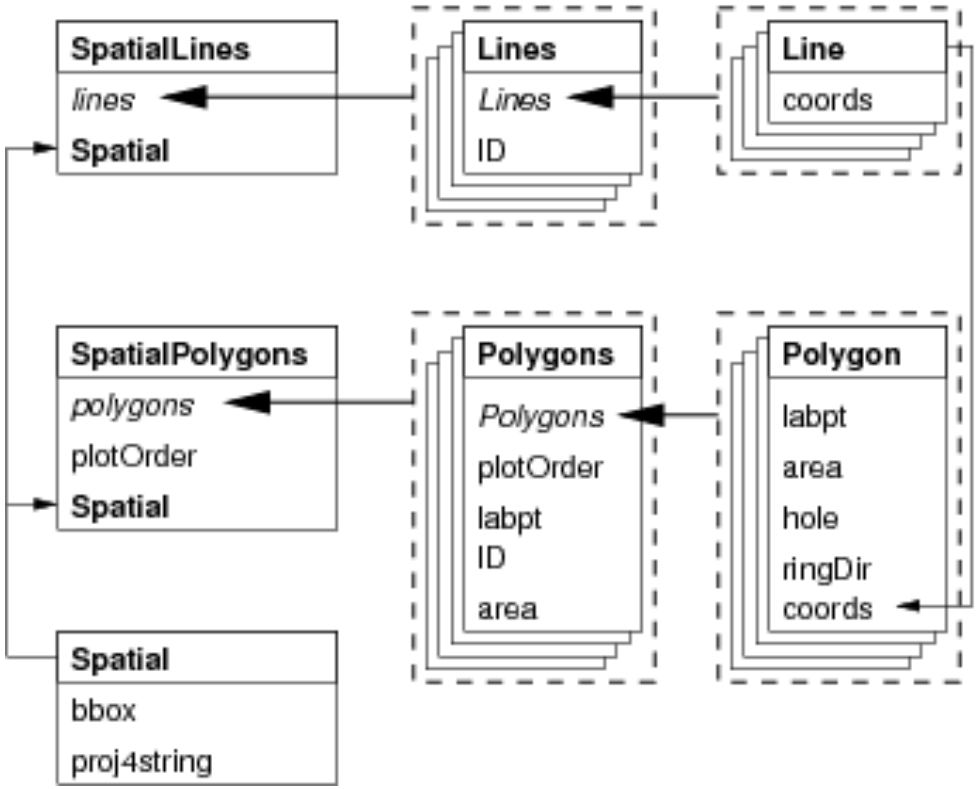
The `sp` package (Pebesma and Bivand, 2005) provides classes and methods for dealing with spatial data in R. The spatial data structures implemented include points, lines, polygons, and grids; each of them with or without attribute data. This package offers a uniform interface for handling spatial data and makes R more coherent for analyzing different types of spatial data.

Some data classes in `sp`:

| type | class | attributes | contains |
|---|---|---|---|
| points | SpatialPoints | | Spatial* |
| | SpatialPointsDataFrame | data.frame | SpatialPoints* |
| line | Line | | |
| lines | Lines | | Line list |
| | SpatialLines | | Spatial*, Lines list |
| | SpatialLinesDataFrame | data.frame | SpatialLines* |
| rings | Polygon | | Line * |
| | Polygons | | Polygon list |
| | SpatialPolygons | | Spatial*, Polygons list |
| | SpatialPolygonsDataFrame | data.frame | SpatialPolygons |

# Data structure for `sp` classes

*Slots* @bbox (extension) and @proj4string are common to all classes. Particular classes have their own slots like @data for attributes, and @lines or @polygons, which are lists of geometric objects.




For instance, for `SpatialPolygons`, each element of list `@polygons` represents a *feature*; each element of list `@Polygons` represents a part (or a hole) of a *feature*.

## Read a polygon feature class

The following data set in `shapefile` format contains agriculture parcels in Ribatejo, not far from Lisbon:

```
parcels<-readOGR(dsn=getwd(),layer="parcels3763")
parcels # summary
head(parcels) # first rows of attribute table
length(parcels) # number of features
```

One can see that `parcels` has 7687 features and it is a `SpatialPolygonsDataFrame`. Hence, `parcels` has an attribute table with 7687 rows: each row contains the attribute values of the corresponding feature.

To access the attribute table, one uses the `@data` slot of the object:

```
slotNames(parcels) # displays "data" "polygons" "plotOrder"
    "bbox" "proj4string"
head(parcels@data)
```

The coordinate reference system is returned by:

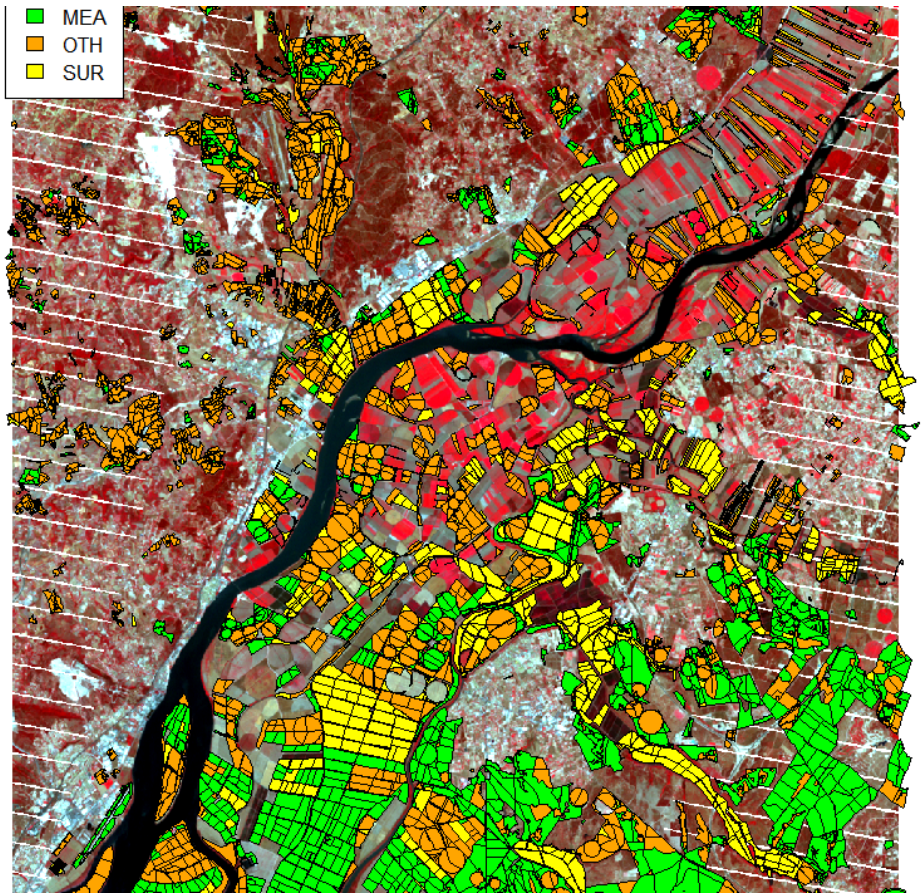
```
parcels@proj4string # object of class "CRS"
```

# Reproject vector data set

Reproject `parcels` to match Landsat color composite (`b`):

```
1 parcels.utm<-spTransform(parcels,CRSobj=b@crs)
```

```
1 par(mfrow=c(1,1),mar=rep(0,4))
2 plotRGB(b,r=4,g=3,b=2,stretch="
      lin")
3 plot(parcels.utm,col=c("green",
      "orange", "yellow")[
      parcels.utm$CODSUBSIDY],add
      =TRUE)
4 # add legend
5 legend(x=parcels.utm@bbox["x","
      min"],y=parcels.utm@bbox["y
      ","max"],legend=levels(
      parcels.utm$CODSUBSIDY),
      fill=c("green", "orange", "
      yellow"))
```




`c("green", "orange", "yellow")[parcels.utm$CODSUBSIDY]` provides the colors for the 7687 features.

# Join attribute tables in R

The attribute table of `parcels.utm` is missing the crop names. This can be fixed by performing a *join* with a new table that has crops' codes and names.

Firstly, we read the new table as a `data.frame`:

```
1 crops<-read.table("crop-codes-and-names.txt",sep=";",header=
    TRUE)
2 head(crops)
```

```
  code                  name
1   88       not cultivated
2  143 permanent grazeland
3   24                  rice
4  142                meadow
5    1                 wheat
6   89           fallow land
...
```

## Join attribute tables in R (cont')

This new `data.frame` has a column `code` that matches the attribute `CODCROP` from the attribute table of `parcels.utm`. Both tables can be joined using the R base function `merge` for `data.frame`:

```
1 newdf<-merge(x=parcels.utm@data, y=crops, by.x="CODCROP", by.y="
      code", all.x=TRUE)
```

However, package `sp` provides an alternative `merge` function, where the first argument is a `sp` object, and returns a `SpatialPolygonsDataFrame` with additional attributes from `crops`:

```
1 parcels.merge<-merge(x=parcels.utm, y=crops, by.x="CODCROP", by.y=
      "code", all.x=TRUE)
2 head(parcels.merge[,-2])
```

```
     CODCROP NPARCEL NSUBPARCEL CODSUBSIDY                 name
2925      88     301          00        OTH not cultivated
2926      88     401          02        OTH not cultivated
```

If `all.x=FALSE` only features that match a `code` value in `crops` are preserved.

# Select features based on attribute values

To select character strings, one can use regular expressions with `grep` or `grepl`.

Select a subset of features using attributes:

```
parcels.merge@data[grep(x=parcels.merge$name,pattern="table"),]
    # selects strings containing 'table'; returns data.frame
parcels.merge[grep(x=parcels.merge$name,pattern="table"),] #
    returns SpatialPolygonsDataFrame
parcels.merge[grep(x=parcels.merge$name,pattern="^table"),] #
    selects strings that start by 'table' and returns SPDF
```

Create map for one single crop:

```
sunflower<-parcels.merge[grep(x
    =parcels.merge$name,pattern
    ="sunflower"),]
plotRGB(b,r=4,g=3,b=2,stretch="
    lin", ext=sunflower@bbox)
plot(sunflower,add=TRUE,col="
    yellow")
```

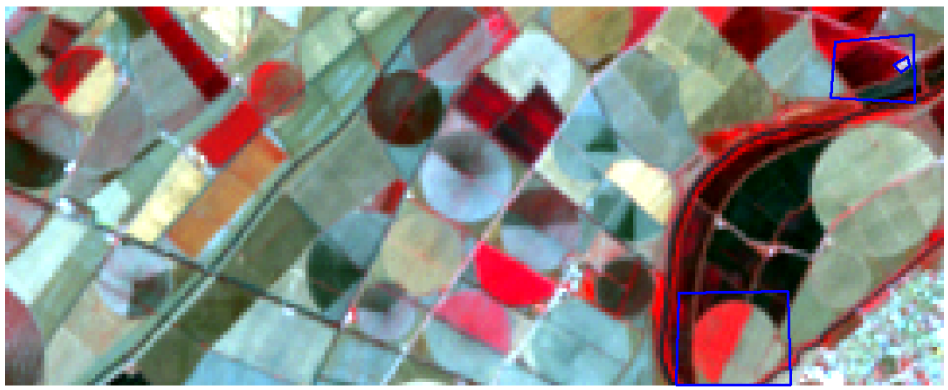
## Create a `sp` object from scratch

Let's define some monitoring plots over our study area. You can define your own coordinates with `locator()` clicking over the image, or you can use the following coordinates for three polygons:

```
pol1<-cbind(c(511113,511103,510498,510463),
c(4311271,4311788,4311734,4311306))
pol2<-cbind(c(510951,511005,511069,511054),
c(4311557,4311498,4311547,4311621)) # pol2 defines a hole
pol3<-cbind(c(509276,509296,510136,510156),
c(4309100,4309806,4309815,4309100))
```

Visualize the location of the plots:

```
plotRGB(b,r=4,g=3,b=2,stretch="
    lin", ext=sunflower@bbox)
polygon(pol1, border="blue",lwd
    =3)
polygon(pol2, border="blue",lwd
    =3)
polygon(pol3, border="blue",lwd
    =3)
```



# Create a `sp` object from scratch (cont')

We define a `SpatialPolygons` object named `myplots` with the three polygons defined earlier. First, we create three parts, indicating which parts are holes:

```
part1<-Polygon(pol1,hole=FALSE)
hole2<-Polygon(pol2,hole=TRUE)
part3<-Polygon(pol3,hole=FALSE)
```

Then, we create features that group one or more parts; each feature has a distinct ID of character type (in this example, the first feature has a "positive" part and a hole):

```
feat1<-Polygons(list(part1,hole2),ID="0")
feat2<-Polygons(list(part3),ID="1")
```

Finally, we group the features and associate the CRS:

```
myplots<-SpatialPolygons(list(feat1,feat2),proj4string=CRS(utm
    .29))
```

# Manipulate the structure of `sp` objects

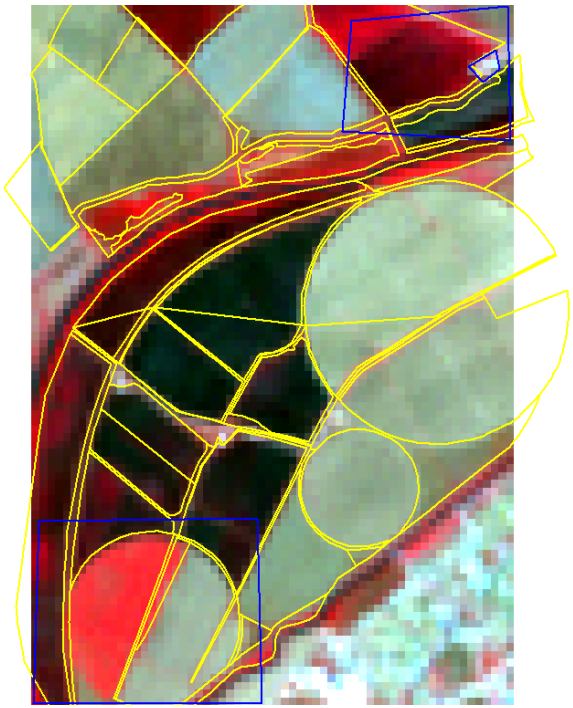
We can use `sp` methods to determine the extent of of `myplots` and select from `parcels.utm` a subset of parcels which centroids fall within the extent of `myplots`.

```
1 ext<-extent(myplots) # object of class extent
```

Select parcels which coordinates are in `ext`:

```
1 xy<-coordinates(parcels.utm) # return the
      centroids of the 7687 features
2 condition<- xy[,1] > ext@xmin & xy[,1] <
      ext@xmax & xy[,2] > ext@ymin & xy[,2]
      < ext@ymax # TRUE if xy falls into ext
3 myparcels<-parcels.utm[condition,]
```

```
1 plotRGB(b,r=4,g=3,b=2,stretch="lin", ext=
      ext)
2 plot(myplots,add=TRUE,border="blue",lwd=2)
3 plot(myparcels,border="yellow",add=TRUE,
      lwd=2)
```



## Overview of methods in package `sp`

1. Standard methods: "[" to select rows (features) or columns of the attribute table, "[[" extract column, "[['<-' assing new value to column, plot, summary, print, dim and names to operate on the attribute table, as.data.frame, as.matrix and image for gridded data, lines, points, subset, stack, over for spatial joins, spplot, length for number of features.
2. Spatial methods: dimensions returns number of spatial dimensions, spTransform (requires rgdal), bbox, coordinates, gridded for SpatialPixels and SpatialPoints, spplot, over, spsample samples point coordinates, geometry strips the data.frame and returns just the geometry.

To manipulate `sp` objects further one needs to explore the structure of each class, namely Points, Lines, and Polygons classes.

Note: there are also Grid classes but that data format has been explored earlier with package `raster`.

## Extract pixel values within spatial features

Remote sensing imagery is an increasingly available source of data. With R there are many ways of extracting pixel values from imagery at given locations. This can be done with the high-level `extract` {raster} function, which returns a list of matrices of pixel values, one for each feature of the input. When `weights=TRUE`, the output includes the proportion of each pixel that is contained in the feature.

Let's apply `extract` to extract pixel values for 'table grape' features, with `cellnumbers=TRUE` to return the indices of the pixels in `b`.

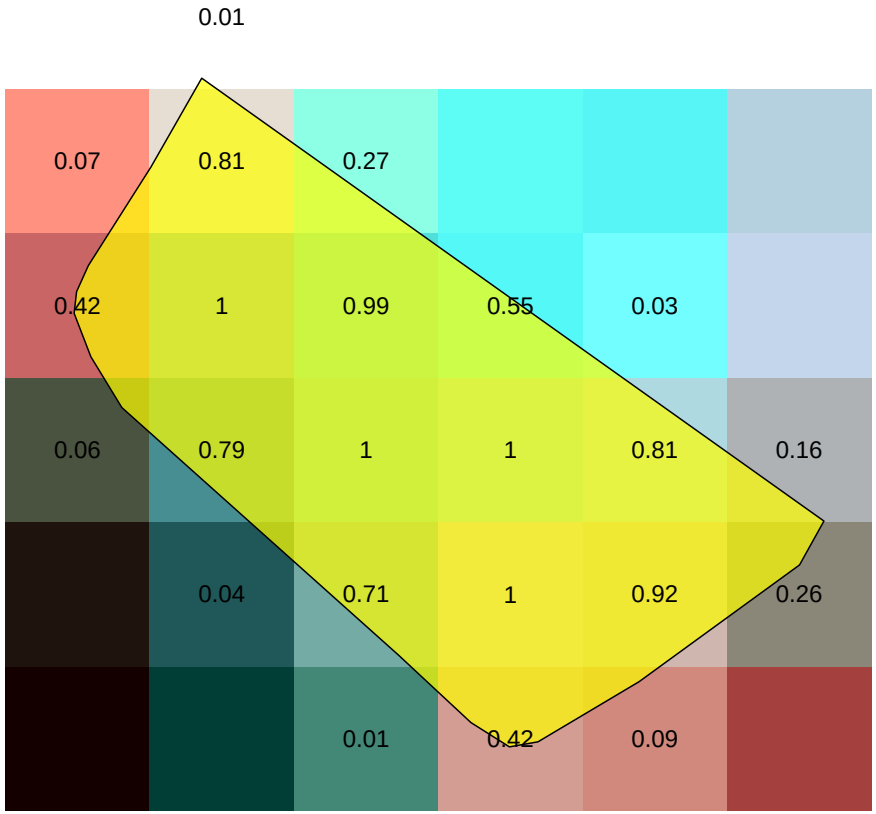
```
1 tablegrape<-parcels.merge[parcels.merge$name=="table grape",]
2 out<-raster::extract(b,tablegrape,weights=TRUE,normalizeWeights
    =FALSE,cellnumbers=TRUE)
3 round(out[[20]],3) # pixel values for the 20-th feature
```

```
      cell band1 band2 band3 band4 band5 band7 weight
[1,] 695121 0.103 0.140 0.156 0.267    NA    NA   0.27
[2,] 695122 0.108 0.140 0.165 0.258    NA    NA   0.01
[3,] 696120 0.088 0.110 0.117 0.267 0.259 0.168   0.19
[4,] 696121 0.097 0.130 0.144 0.258    NA    NA   0.99
...
```

# Extract pixel values within spatial features

We can superimpose the feature over the image and add the weights:

```
1 par(mfrow=c(1,1),mar=rep(0,4))
2 idxfeat<-20 # display the 20-th
      feature
3 plotRGB(b,r=4,g=3,b=2,stretch="
      lin",ext=tablegrape[idxfeat
      ,]@bbox)
4 plot(tablegrape[idxfeat,],col=
      rgb(r=1,b=0,g=1,alpha=.7),
      add=TRUE)
5 xy<-coordinates(b)[out[[idxfeat
      ]][,"cell"],]
6 text(xy[,1],xy[,2],out[[idxfeat
      ]][,"weight"])
```




Indices `out[[index]][,"cell"]` are used to obtain the pixel's coordinates. Option `ext=tablegrape[idxfeat,]@bbox` draws raster just over the feature's extent and `col=rgb(r=1,b=0,g=1, alpha=.7)` defines "yellow" with 70% opacity.

# Geographic data analysis with package `rgeos`

## Package `rgeos`

Package `rgeos` implements the methods of the OGC standard (see Slide 26).

However, the major spatial analysis functions from package `rgeos` apply to *valid* geometric objects. Hence, it can be important to use function `gIsvalid` to select valid objects and obtain some information about non-valid ones.

Here, we check if the 7687 features of `parcels.utm` are valid:

```
valid<-gIsValid(parcels.utm, byid=TRUE, reason=TRUE)
idx<-which(valid!="Valid Geometry")
idx # feature 4129
```

In the next slide we will see that the feature self intersects, and its vertices 24 and 33 have the same coordinates:

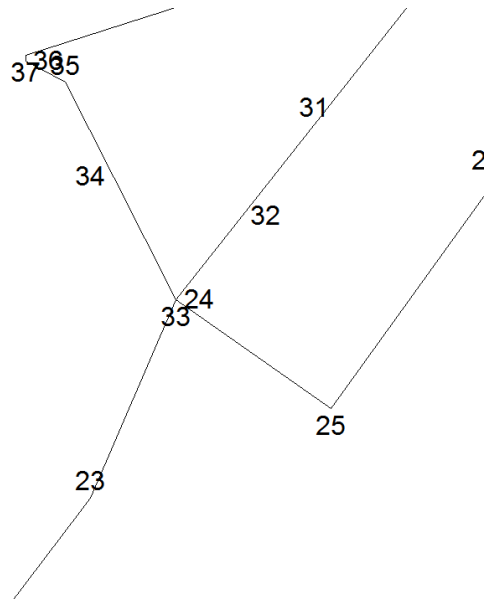
```
parcels.utm[idx,]@polygons[[1]]@Polygons[[1]]@coords[c(24,33),]
```

```
          [,1]    [,2]
[1,] 494784.5 4324416
[2,] 494784.5 4324416
```

# Validity of geometries

The output of `valid` helps us to understand where the problem lies:

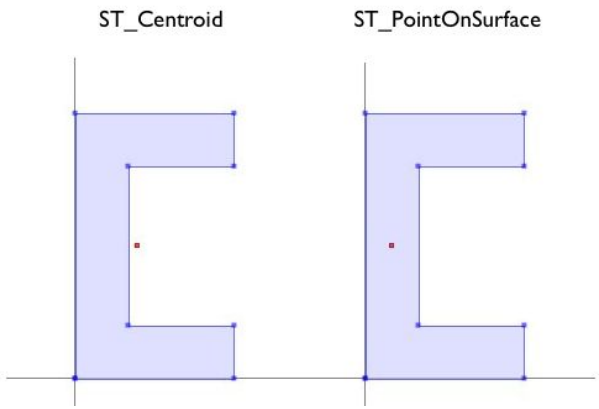
```
valid[idx]  # 4128  "Ring Self-
    intersection[494784.53235936
    4324416.33533211]"
plot(parcels.utm[idx,]) #
D <- 25 # define some neighborhood
plot(parcels.utm[idx,], xlim=c(494784-D,
    494784+D), ylim=c(4324416-D, 4324416+D
    ))
# print coordinates of vertices
xy<-parcels.utm[idx,]@polygons[[1]]
    @Polygons[[1]]@coords
text(xy,apply(round(xy),1,paste,collapse="
    ,"))
# print indices of the vertices
plot(parcels.utm[idx,], xlim=c(494784-D,
    494784+D), ylim=c(4324416-D, 4324416+D
    ))
text(xy,labels=1:nrow(xy),pos=c(1,2,3,4),
    cex=.8) # vertices 24 and 33 coincide
```



# Some basic properties of geometric objects

All geometric objects have basic properties which depend on their dimension. For instance,

1. (`length`) A curve has a length, which is the length of the linestring. If the curve is *multipart*, its length is the sum of the lengths of all parts;
2. (`area`) area is a property of surface objects; If a surface object is *multipart*, its area is the sum of the areas of each part;
3. (`centroid`) is the point of the center of mass of the object, which might lie or not on the object;
4. (`PointOnSurface`) returns a point that lies on the object.



## Some basic properties of geometric objects returned by `rgeos`

Recall that `myplots` has a hole in the 1st feature. Let's compare the areas returned by the slot `@area` and `gArea(myplots)`:

```
myplots@polygons[[1]]@area # 1st feature: 296375 (m2)
myplots@polygons[[2]]@area # 2nd feature: 611030 (m2)
gArea(myplots, byid=TRUE)  # both features: 288873 611030 (m2)
```

While the slot `@area` of a `sp` object returns the area of "positive" polygons but doesn't take into account holes, `gArea` returns the true area of the feature, after holes are removed.

Note that you can get the `@area` values for all features using `sapply` since `myplots@polygons` is a list of features.

```
sapply(myplots@polygons, function(feat) feat@area) # 296375
    611030
```

Some properties are returned as new `sp` objects:

```
gPointOnSurface(myplots, byid=TRUE) # returns SpatialPoints
```

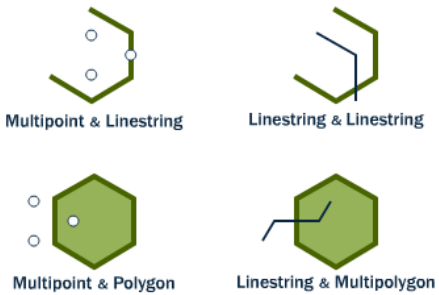
## Methods for testing spatial relations

Given geometric objects *a* and *b* of dimension 0 (**p**), 1 (**L**) or 2 (**P**):

1. Equal: the objects coincide;
2. Disjoint: the objects do not intersect;
3. Touches applies to two objects as long as at least one has dimension larger than 0: *a* Touches *b* if they intersect at their boundaries;
4. For *a*/*b* of type **p**/**L**, **p**/**P**, **L**/**L** or **L**/**P**, *a* Crosses *b* if the geometries overlap but *a* is not within *b* nor *b* is within *a*;
5. *a* Within *b* if *a* doesn't intersect the exterior of *b*;
6. Overlaps applies to objects of the same dimension: It is true is *a* and *b* intersect but *a* is not within *b* nor *b* is within *a*;
7. *a* Contains *b* if *b* Within *a*;
8. *a* Intersects *b* if *a* and *b* are not Disjoint;
9. Relates allows us to define any kind of spatial relation between *a* and *b* by testing the intersections between the interior, boundary, and exterior of both objects.

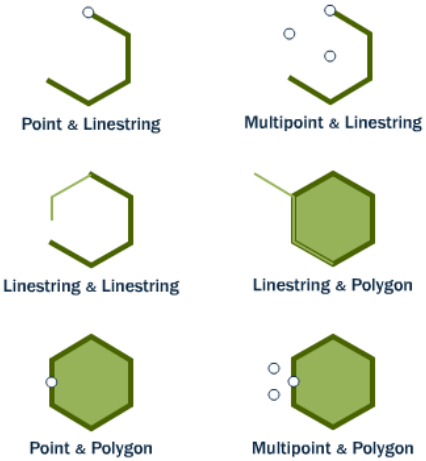
# Methods for testing spatial relations (cont')

## Cross

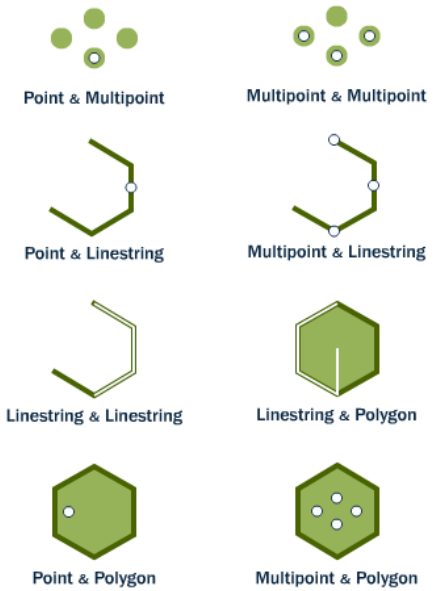



## Overlap

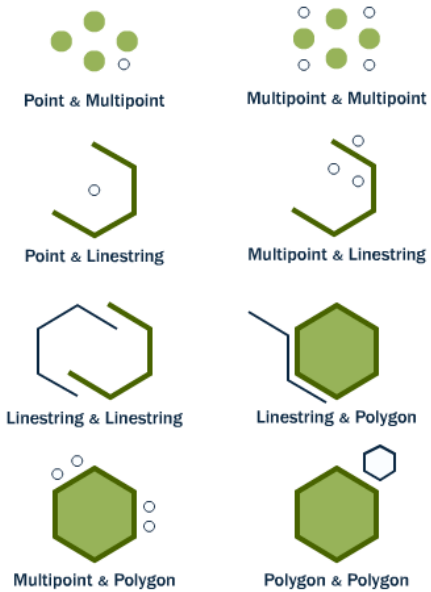

## Touch



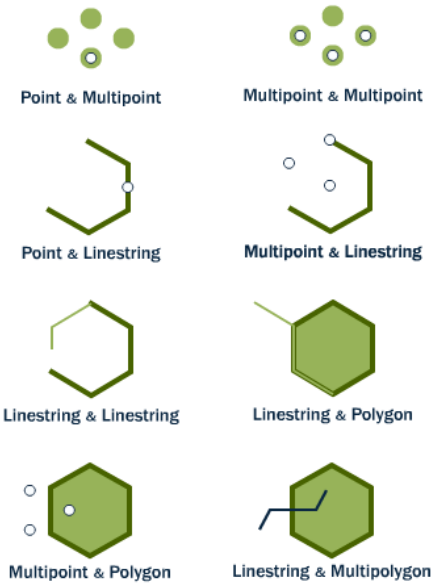

## Within/Contains




## Disjoint




## Intersects



# Geometries and spatial relations: further details

The formal definition of spatial relations is based on the Dimensionally Extended 9-Intersection model DE9IM (Egenhofer, Clementini *et al.*) which uses the 9 intersections one can define between the interior, boundary and exterior of two geometric objects.

In particular, the spatial relation Relates can be defined in any possible way allowed by the DE9IM model.

The details can be found in the above mentioned source `www.opengeospatial.org/standards/sfa` (Simple Feature Access - Part 1: Common Architecture).

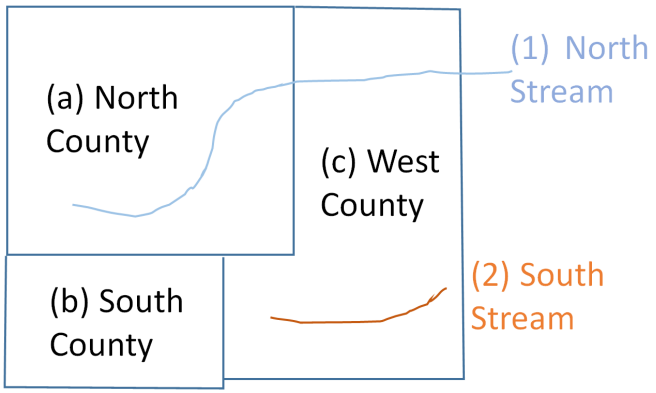
A description of DE9IM and the derived definitions of spatial relations are also available at `en.wikipedia.org/wiki/DE-9IM`.

# Methods for testing spatial relations: logical matrices

Consider the two following feature classes (fc):

1. fc `streams` of type curve with two *singlepart* objects (1 and 2) where each feature represents a stream;
2. fc `counties` of type surface with three *singlepart* objects (a, b, c) where each feature represents a county.

The result of the method Intersects: "`streams` intersects `counties`" is the following:

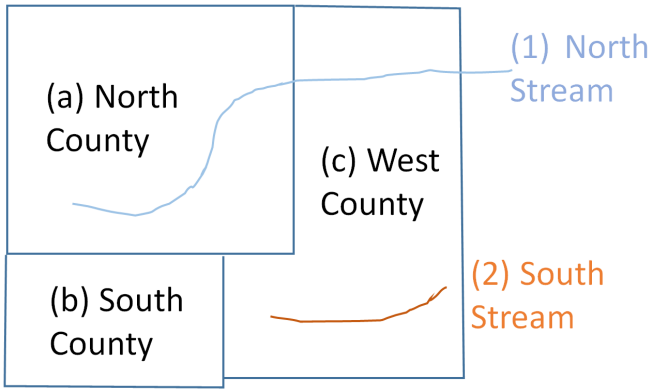



`streams` × `counties`

| intersects | a | b | c |
|---|---|---|---|
| 1 | **T** | **F** | **T** |
| 2 | **F** | **F** | **T** |

# Methods for testing spatial relations: logical matrices (cont')

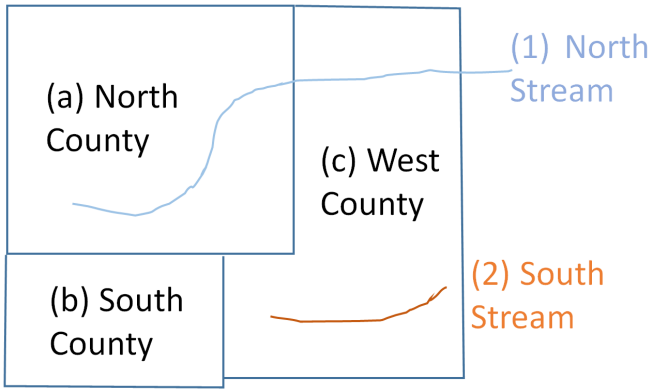
- Within relations "`streams` within `counties`"




streams \ counties

| within | a | b | c |
|---|---|---|---|
| 1 | **F** | **F** | **F** |
| 2 | **F** | **F** | **T** |

- The above example is equivalent to using the relation Contains if *inputs* are switched around: "`counties` contains `streams`"




counties \ streams

| contains | 1 | 2 |
|---|---|---|
| a | **F** | **F** |
| b | **F** | **F** |
| c | **F** | **T** |

## Methods for testing spatial relations with `rgeos`

The package `rgeos` includes functions that implement OGC methods for testing spatial relations: `gIntersects`, `gWithin`, `gContains`, `gContainsProperly`, `gCovers`, `gCoveredBy`, `gCrosses`, `gEquals`, `gEqualsExact`, `gOverlaps`, `gRelate`, `gTouches`.

Before we start using methods that apply to more than one data set, and to prevent meaningless error messages, let's first make CRS strings equal, since we know that both data sets have the same CRS:

```
1 myplots@proj4string<-myparcels@proj4string
```

Let's determine which parcels intersect the features in `myplots`:

```
1 gits<-gIntersects(myplots,myparcels, byid=
    TRUE,checkValidity=TRUE) # matrix of
    TRUE and FALSE
2 dim(gits) # the matrix is 62*2 because
    myparcels has 62 features and myplots
    has 2 features
3 tail(gits)
```

```
          0     1
609    TRUE FALSE
611   FALSE FALSE
612   FALSE FALSE
613   FALSE FALSE
614    TRUE FALSE
617    TRUE FALSE
```

# Methods for testing spatial relations with `rgeos` (cont')

Instead of `gIntersects`, we use `gContains` to determine which parcels are totally contained in some feature of `myplots`:

```
1 gcts<-gContains(myplots,myparcels, byid=TRUE,checkValidity=TRUE
      ) # matrix of TRUE and FALSE
2 tail(gcts) # last rows of the matrix
3 which(apply(gcts,1,max)==1) # returns 5 feature indices
```

```
> which(apply(gcts,1,max)==1)
307 327 601 608 609
  9  22  49  56  57
> tail(gcts )
        0     1
609  TRUE FALSE
611 FALSE FALSE
612 FALSE FALSE
613 FALSE FALSE
614 FALSE FALSE
617 FALSE FALSE
```

Looking at `gcts`, one concludes that features named 608 and 609 of `myparcels` are contained in the 1st feature of `myplots`, and features named 307, 327, and 601 of `myparcels` are contained in the 2nd feature of `myplots`.

# Spatial query

The output of `gIntersects`, `gWithin`, `gContains`, etc, can be used to perform a spatial query, also called a "selection by location."

Earlier, we used the "[" method of `sp`, to perform selection of features by attributes. A selection requires some condition which is **TRUE** or **FALSE** for each row of the table, i.e. for each feature.

Using the logical matrices produced by e.g. `gIntersects`, we can define a logical vector to perform a selection. For example, the vector

```
1 apply(gcts,1,max)==1 # logical vector of length 62; it is TRUE
    if the corresponding row of gcts=gContains(myplots,
    myparcels, byid=TRUE) has at least one TRUE value
```

in the previous slide indicates which parcels from `myparcels` are contained in at least one feature of `myplots`. To obtain the result of the corresponding spatial query, we just have to write

```
1 myparcels[apply(gcts,1,max)==1,] # new SpatialPolygonsDataFrame
```

# Spatial analysis: computing distances and new geometric objects

Up to this point, spatial analysis was only used to do queries on the data and to relate (join) information from spatial location.

The OGC/ISO standard (see Slide 26) also defines a set of operators which return the distance between geometric objects or return new geometric objects from existing ones according to some spatial relation. The standard operators are: Distance, Buffer, ConvexHull, Intersection, Union, Difference, and Symmetric Difference.

The results of some of these methods, namely Distance and Buffer, depend on the distance between any two points and therefore vary with the Coordinate Reference System associated to the geometry.

For instance, the distance measured on a geographic CRS (latitude/longitude) is in general quite different from the distance measured on projected coordinates for large distances.

# Spatial analysis: computing distances and new geometric objects (cont')

Given two geometric objects *a* and *b*:

1. Distance between *a* and *b* returns the shortest **distance** between points of *a* and points of *b*;
2. Buffer of *a* with distance *d* returns a geometric object that represents all points whose **distance** to *a* is less or equal to *d*;
3. ConvexHull of *a* returns a geometric object that represents the convex hull of all points in *a*;
4. Intersection of *a* and *b* returns the geometric object that represents all the points that belong to *a* and *b*;
5. Union of *a* and *b* returns the geometric object that represents all the points that belong either to *a* or to *b*;
6. Difference between *a* and *b* returns the geometric object that represents all the points that belong to *a* but not to *b*;
7. Symmetric Difference of *a* and *b* returns the geometric object that represents all the points that either (1) belong to *a* but not to *b*, or (2) belong to *b* but not to *a*.

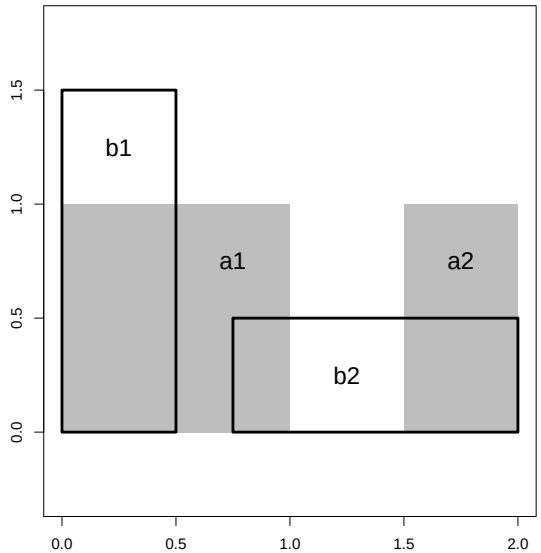
# Spatial analysis: computing distances

The method Distance returns distances between geometric objects *a* of feature class A and geometric objects *b* of feature class B. The output is a distance matrix.

Let A and B be surface features classes (i.e. of polygon type) with the same CRS:

1. fc `A` with objects singlepart `a1` and `a2` (gray shape);
2. fc `B`, with objects singlepart `b1` and `b2` (black border).

The distance matrix is on the right hand side. It indicates that all distances are 0 except for the pair (`b1`, `a2`).




| distance | b1 | b2 |
|---|---|---|
| a1 | 0.00 | 0.00 |
| a2 | 1.00 | 0.00 |

## Computing distances with `gDistance`

R has many packages that can be used to compute distances between locations like `rdist` {fields}, `nn2` {RANN}, `distGeo` {geosphere}, and `gDistance` {rgeos} to apply method Distance to `sp` objects.

The following command returns a 62 $\times$ 2 matrix of distances between the features. It uses the vertices of each feature: the value [i,j] is the shortest distance between all pairs of vertices from the i-th feature of `myparcels` and the j-th feature of `myplots`. Hence, all pairs of features that intersect (see `gIntersects`) have distance 0.

```
1 gDistance(myplots, myparcels, byid=TRUE)
```

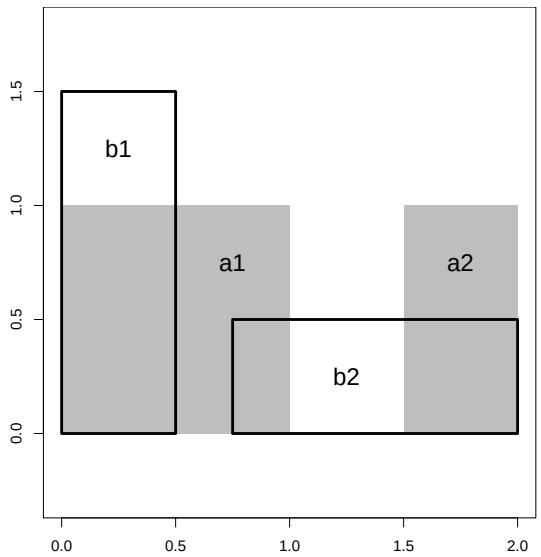
As a result, function `gDistance` is slow for large data sets. As an alternative, one can "simplify" one or both feature classes:

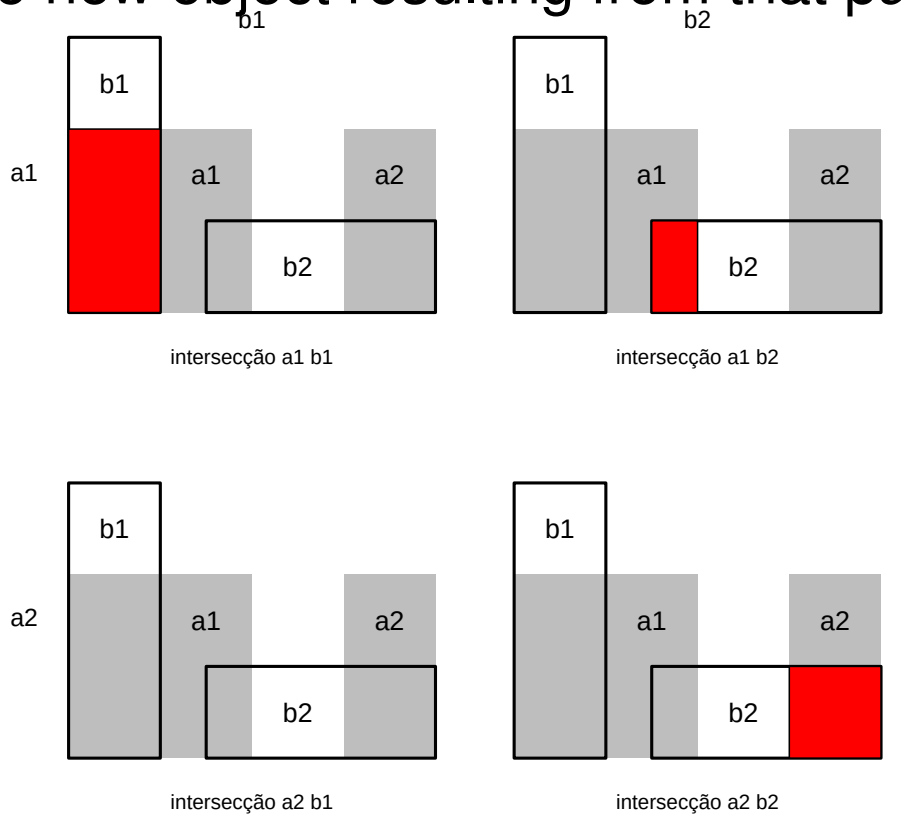
```
1 gDistance(gSimplify(myplots, tol=10), gSimplify(myparcels, tol=10)
      , byid=TRUE) # or replace feature by centroid:
2 gDistance(myplots, gCentroid(myparcels, byid=TRUE), byid=TRUE)
```

# Spatial analysis: Intersection method

The Intersection method returns new geometric objects derived from the objects in the inputs. It is applied to every pair (`a`,`b`) , where `a` belongs to feature class A and `b` belongs to feature class B.

In this example, there are 4 pairs of objects, all singlepart, and the output of Intersection has the 3 red objects depicted below. Since `a2` and `b1` do not intersect, there is no new object resulting from that pair.



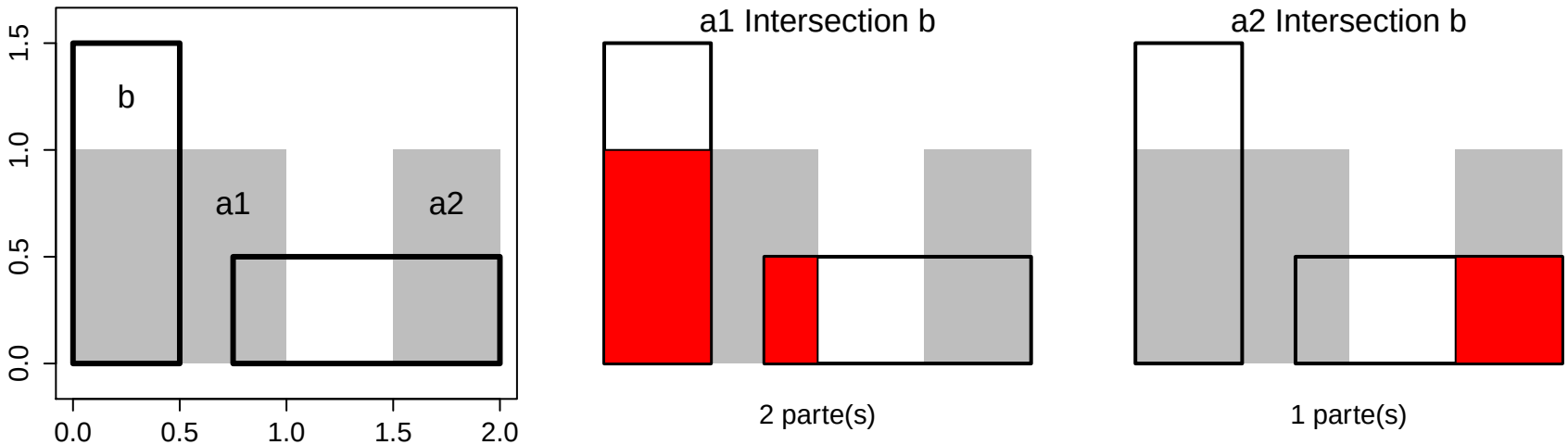



# Spatial analysis: Intersection method (multipart objects)

In the example below,

- feature class A has two single part geometric objects `a1` and `a2` (i.e. two features), depicted in gray, and
- feature class B has a single multipart geometric object `b` (i.e. a single feature), depicted by a black border.

As before, the operation is applied to all pairs (`a`,`b`), where `a` belongs to A and `b` belongs to B. In this example, there are two pairs (`a1`,`b`) and (`a2`,`b`). The output has two objects depicted in red; the one on the left is multipart; the one on the right is singlepart.

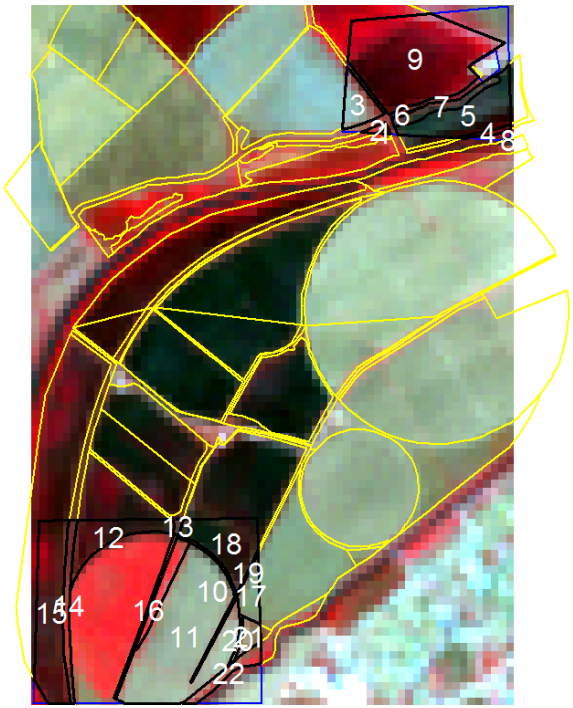


## Computing intersections with `gIntersection`

There are at most 62 $\times$ 2 possible intersections betweens features of `myparcels` and `myplots`, but in most cases the intersection is empty. The following command returns an object of class `SpatialPolygons` (with no attribute table) and all non empty features defined by all 62 $\times$ 2 intersections:

```
gits<-gIntersection(myplots,myparcels,byid=TRUE) # returns
    SpatialPolygons with 22 features
```

```
plotRGB(b,r=4,g=3,b=2,stretch="lin", ext=
    ext)
plot(myplots,add=TRUE,border="blue",lwd=2)
plot(myparcels,border="yellow",add=TRUE,
    lwd=2)
plot(gits,border="black",add=TRUE,lwd=3)
xy<-coordinates(gits)
text(xy[,1],xy[,2],1:length(gits),cex=2,
    col="white")
```

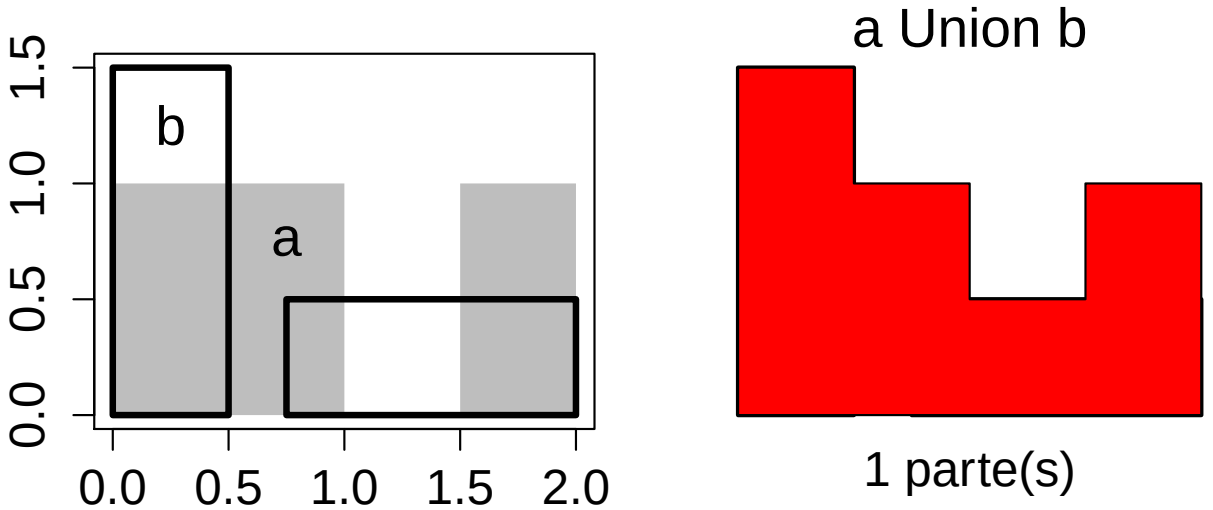


# Spatial analysis: Union method

This method is also applied to all pairs of objects (`a`,`b`), where (`a` belongs to A and `b` belongs to B. In the example below,

- feature class A has a single multipart geometric object `a` (i.e. a single feature with two parts), and
- feature class B has also a single multipart geometric object `b` (i.e. a single feature with two parts).

The operation is applied to the single pair (`a`,`b`). In this case the output has a unique singlepart object which is, therefore, spatially connected.

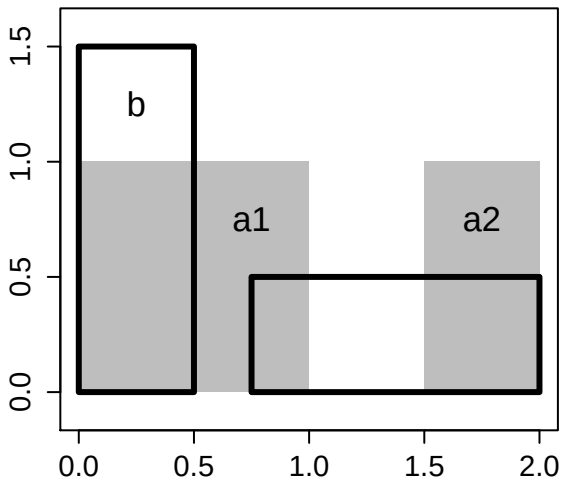


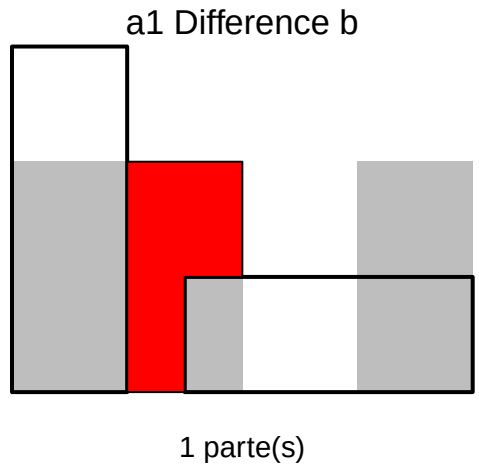
# Spatial analysis: Difference method

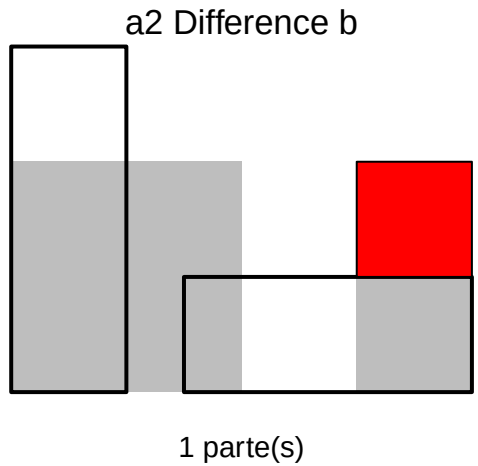
This method is also applied to all pairs of objects (`a`,`b`), where (`a` belongs to A and `b` belongs to B, but depends on the order of the inputs. In the example below,

- feature class A has two single part geometric objects `a1` and `a2` (i.e. two features), and
- feature class B has a single multipart geometric object `b` (a single feature with two parts).

In this example, there are two pairs (`a1`,`b`) and (`a2`,`b`). The output has two objects depicted in red, both singlepart.



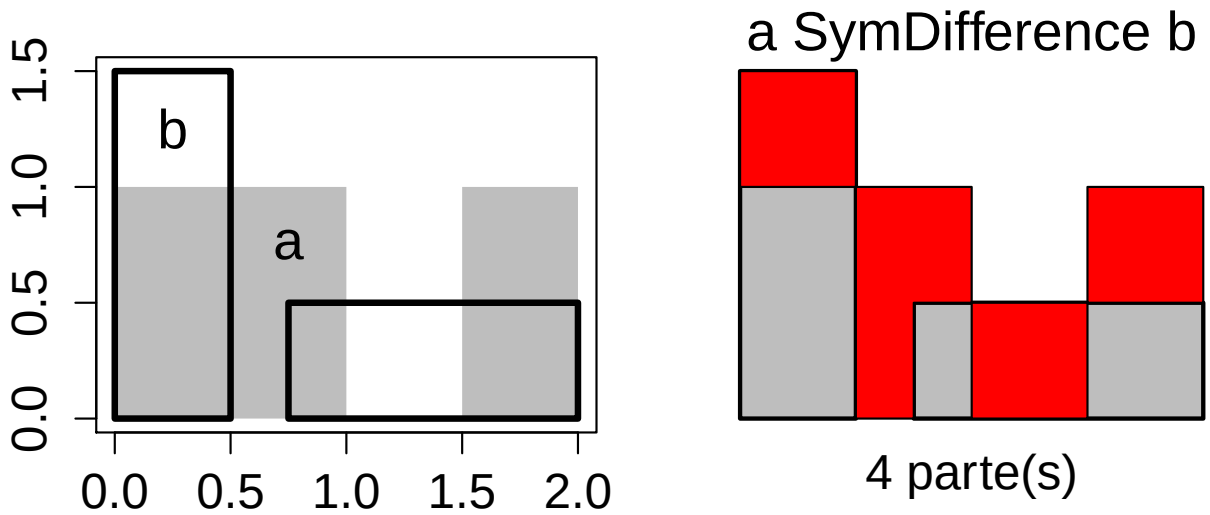



# Spatial analysis: Symmetric Difference method

Unlike Difference, the result of Symmetric Difference does not depend on the order of the inputs. In the example below,

- feature class A has a single multipart geometric object `a` (i.e. a single feature), and
- feature class B has also a single multipart geometric object `b` (i.e. a single feature).

The operation is applied to the single pair (`a`,`b`). In this case the output has a unique multipart geometric object, with 4 parts: areas in `a` but not in `b`, and areas in `b` but not in `a`.



## Aggregating features with `gUnaryUnion`

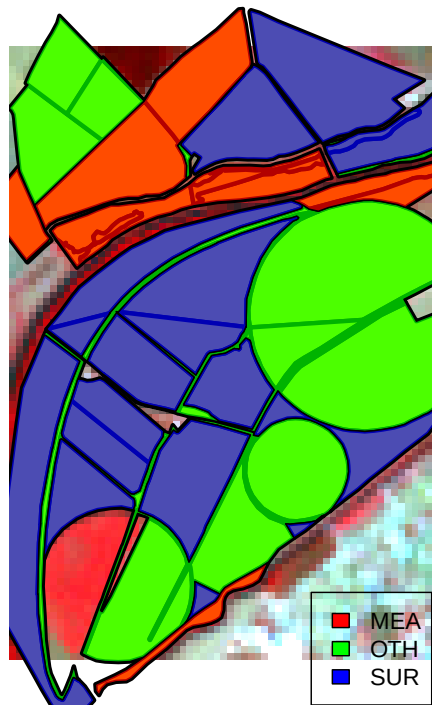
Often, one needs to "dissolve" features, i.e. return the feature class with intersecting geometries merged. To do this, it is just necessary to define the new membership of the features in the output.

Suppose we want to merge features in `myparcels` which have the same subsidy code, defined by `myparcels$COSSUBSIDY`:

```
1 umy<-gUnaryUnion(myparcels, id=myparcels$CODSUBSIDY)
```

Visualize new features over the original ones; the faded boundaries have been removed.

```
1 plotRGB(b,r=4,g=3,b=2,stretch="lin", ext=
      ext=extent(myparcels))
2 plot(myparcels,add=TRUE, col="yellow",lwd
      =3)
3 L<-levels(myparcels$CODSUBSIDY)
4 plot(umy, col=rainbow(length(L),alpha=0.7)
      ,add=TRUE)
5 legend("bottomright",legend=L, fill=rainbow
      (length(L)))
```



# Computing differences with `gDifference`

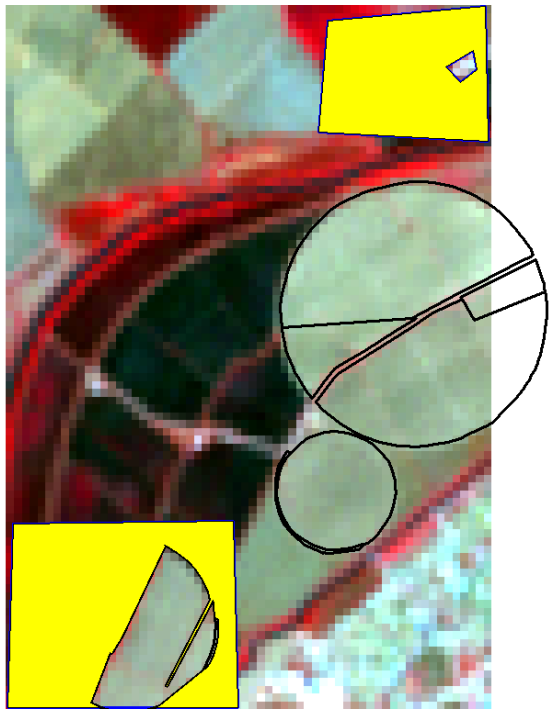
Let's use `gDifference` to determine the parts of `myplots` (2 features) that are not covered by features in `sunflower` (6 features).

```
gdiff<-gDifference(myplots, myparcels, byid=TRUE) #returns 6*2 features
```

returns all 12 possible combinations. But if we apply `gUnaryUnion` to `sunflower` first, we only get the two differences between `myplots` and the aggregate area of `sunflower`:

```
gdiff<-gDifference(myplots, gUnaryUnion(sunflower), byid=TRUE)
```

```
plotRGB(b, r=4, g=3, b=2, stretch="lin", ext=extent(myplots))
plot(myplots, add=TRUE, border="blue", lwd=3)
plot(sunflower, add=TRUE, border="black", lwd=3)
plot(gdiff, col="yellow", add=TRUE)
```



## Computing buffers with `gBuffer`

Determining buffers is a typical GIS operation that is performed on spatial data. Buffers can be positive or negative.

Suppose that we want to determine the sunflower parcels which are totally contained inside `myplots`:

```
gWithin(sunflower,myplots,byid=TRUE) # TRUE for one feature
```

If we expand `myplots` by 20 meters with `gBuffer`:

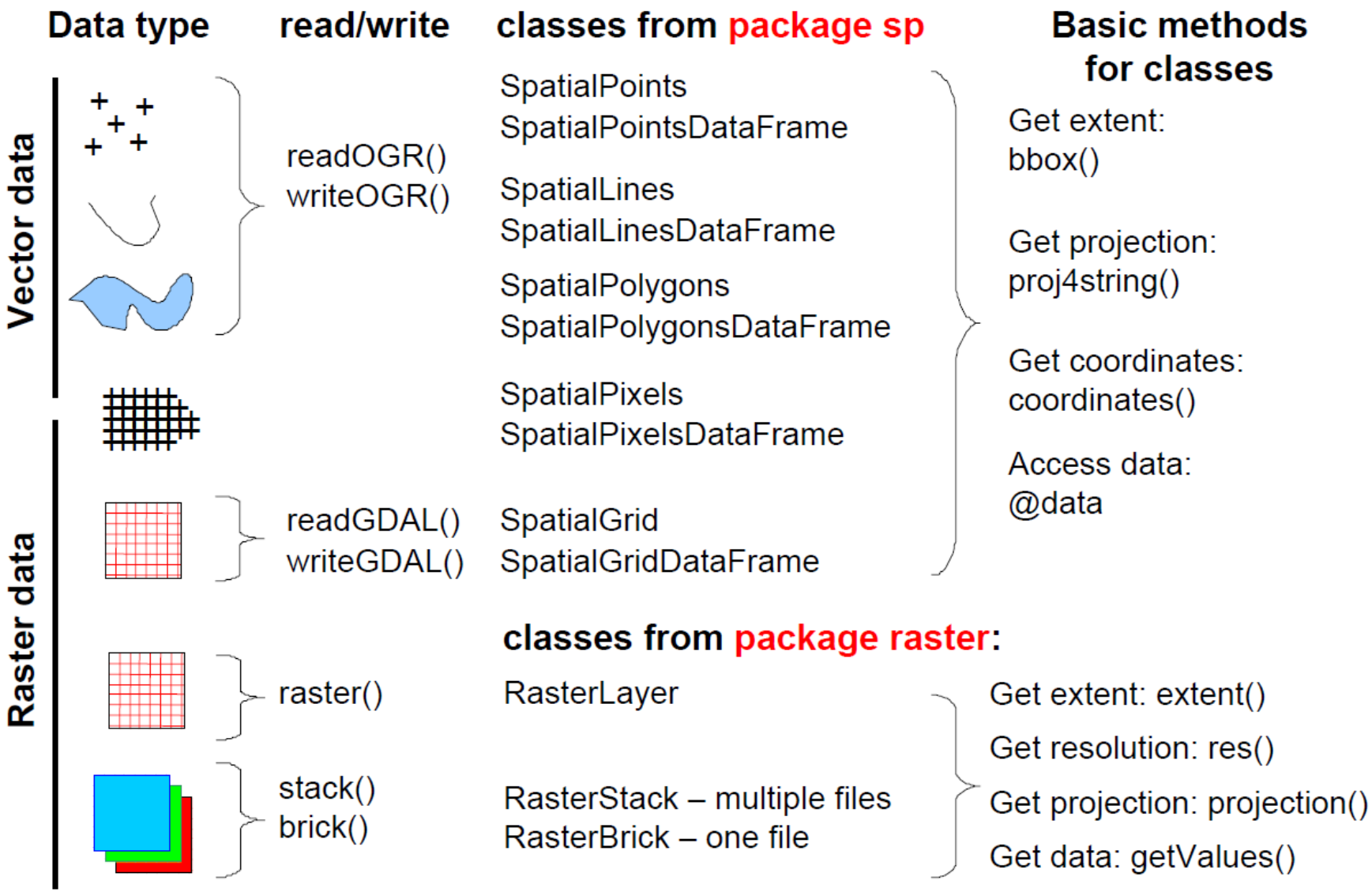
```
gWithin(sunflower,gBuffer(myplots,byid=
    TRUE,width=20),byid=TRUE) # TRUE for
    two features
```

```
plotRGB(b,r=4,g=3,b=2,stretch="lin", ext
    =1.1*extent(myplots))
plot(sunflower,add=TRUE, border="yellow",
    lwd=2)
plot(myplots,add=TRUE,border="blue",lwd=2)
plot(gBuffer(myplots,byid=TRUE,width=20),
    add=TRUE,border="red",lwd=2)
```

# Overview

# General view of packages `raster` e `sp`

## Classes for spatial data in R



## Main methods from `raster`

1. Read/write: `raster()`, `brick()` `stack()` (multilayer images) and `writeRaster()`
2. Coordinate reference systems: `projection()` or `@crs`
3. Spatial resolution: `res()`
4. Extension: `extent()` or `@extent`
5. Range of values: `@data`
6. Reprojection: `projectRaster()`
7. Pixel coordinates: `coordinates()`
8. Pixel values: `values()`
9. Extract pixel values at given locations: `extract()`
10. Crop: `crop()`
11. Mosaics: `merge()` and `mosaic()`
12. Slope, aspect and hillshading for DEM : `terrain()`, `hillShade()`
13. Linear and non linear filters: `focal()`

# Main methods from `sp`

1. Read/write: `readOGR()` and `writeOGR()`
2. Coordinate reference systems: `proj4string()` or `@proj4string`
3. Extension: `bbox()` or `@bbox`
4. Attribute table: `@data`
5. Reprojection: `spTransform()`
6. Coordinates of feature's centroids: `coordinates()`
7. List of features: `@polygons` or `@lines`
8. List of parts of the i-th feature: `@polygons[[i]]@Polygons`
9. List of parts of the i-th feature: `@lines[[i]]@Lines`
10. Join tables: `merge()`
11. Convert to raster: `rasterize()`

# File formats for read/write

1. To list file formats that can be accessed through `readOGR()` and `writeOGR()`:

```
ogrDrivers()
```

2. To list raster formats that can be accessed through `raster()` and `writeRaster()`:

```
writeFormats()
```

# Index