spatial stuff in R

May 2021

By: chloé

Session info:

- R version 4.0.1 (2020-06-06)
- sf v0.9.4
- raster v3.1.5
- ggplot2 v3.3.2

GIS in R

Sure, you can use specialized GIS software like ArcGIS or QGIS (free & open source btw) for your spatial analysis needs. But why learn a whole complicated new program when you already know and love R? Exactly. No reason. Hopefully this short guide will give you enough information to get going with spatial stuff in R: what the deal with the data is, how to load it, how to manipulate it, and of course how to plot it.

Of course, R isn't the same as a dedicated GIS software. In ArcGIS or QGIS, data loads as a map across which you can pan, zoom, and select features. It's easy to create and manipulate shapefiles manually. You can also load pictures of maps (like a jpeg) and transform them for georeferencing. R doesn't do this. So you might still need GIS sometimes.

Spatial data

Spatial data comes in a few forms:

1. Spreadsheets with coordinates
    - You probably already know how to load a spreadsheet into R. If it has coordinate data, it's easy to tell R that this is *spatial* data.
2. Raster maps (.tif, .asc, others)
    - A spatial raster (usually .tif, sometimes .asc, others) has pixels of different shades that encode specific information. It could be continuous data, like population density (e.g. dark colors mean no people, lighter colors mean more people), or categorical data where different colors correspond to categories (like types of land cover). Important to consider are the **resolution** (the size of the pixels) and the spatial **extent** (a city? a continent?). Raster files can be pretty big, so fine-scale data with large extents might be unwieldy. Often when you download raster data, you'll get a lot of other files (.aux; .vat.dbf; etc) in addition to the .tif. These hold the other spatial information like coordinate systems.
3. Shapefiles (.shp)
    - Shapefiles are vector data. There are 3 main **feature classes** in shapefiles: points, lines, and polygons. A shapefile stores the geographic data (the map) and can also be accompanied by an **attribute table**. If you had a shapefile with the boundaries of a few cities encoded as polygons, your attribute table could have a row for each city with a unique ID (the FID for "feature ID") and any other information you cared to store: the area, population density, median income, etc. Shapefiles (.shp) are also accompanied by other files (.shx, .dbf, ...) which store the attributes, geographic information, etc.

R packages

The main R packages for handling spatial data are **sf**, **sp**, and **raster**.

Vector data:

- **sp**, *spatial points*, works with its own class of data, e.g. 'SpatialPoints' or 'SpatialPointsDataFrame'. Not gonna talk more about this because sf is newer and easier to use.
- **sf**, *simple features*, is sort of a reworking of sp that works well with the tidyverse. This is because the sf data class is like a dataframe with an extra **geometry** column that contains the spatial information. Functions from sp and sf are sometimes similar, but ones from sf all start with 'st_'

Raster data:

- **raster** is for dealing with rasters.

There are other packages you'll come across regularly like **rgeos** & **rgdal**, that have functions for calculating distances & fancier geospatial analysis things.

Loading data

It's easy to read in shapefiles and rasters:

```
## Shapefile:
library(sf)
dat <- read_sf("data.shp")

## Raster:
library(raster)
dat1 <- raster("raster.tif") # for .adf and .tif
dat2 <- raster(read.asciigrid("raster.asc")) # for .asc
```

If you have a spreadsheet with coordinates on it:

```
## Read in trapping sites
sites <- read.csv("sites.csv", header = TRUE)

sites_sf <- st_as_sf(sites, coords = c("lon","lat")) # lon and lat are column names

# You can also use this function to turn objects of other classes into sf objects
```

Coooooool, it's spatial now:

```
## Check the class:
> class(sites_sf)
[1] "sf"     "data.frame"
```

Faking data

Let's make some fake data to play with for now:

```
library(sf)
library(tidyverse)
library(raster)
# Make some polygons: Utah+Colorado, cuz they're easy:
# lon and lat are coordinates of the vertices
statesdf <- data.frame(state = c(rep("Utah", 6), rep("Colorado", 4)),
                       lon = c(-114,-111, -111, -109, -109, -114, -109, -102, -102, -109),
                       lat = c(42, 42, 41, 41, 37, 37, 41, 41, 37, 37))
states <- statesdf %>%
        st_as_sf(coords = c("lon", "lat"), crs = 4326) %>% # dataframe to spatial data
        group_by(state) %>%
        summarise(geometry = st_combine(geometry)) %>% # mashes all the points into 1
        st_cast("POLYGON") # tell R this is 2 polygon, not 10 points

# Sprinkle some random sites over them:
sitesdf <- data.frame(ID = c("A", "B", "C", "D", "E"),
                      lon = c(-111.9, -111.0, -109.6, -113.4, -108.6),
                      lat = c(40.8, 41.0, 38.5, 38.0, 39.0))
sites_sf <- st_as_sf(sitesdf, coords = c("lon", "lat"), crs = NA)# no coordinate system yet
```

Coordinate systems

The **coordinate reference system** (**crs**) is the way the map is drawn. Geographic coordinate systems work on a sphere, while projected coordinate systems project the earth onto a flat surface.

Here's a good resource for using & specifying coordinate reference systems in R. This site is also super helpful for finding codes & converting formats.

To check the coordinate system of your data:

```
> st_crs(sites_sf)

## Coordinate Reference System: NA
```

A lot of the time raster and shapefile data come with coordinate systems specified in those extra files. But sometimes they don't. And if you're bringing in point data from longitude and latitude columns on a spreadsheet, R doesn't automatically know what system it uses, so you have to enlighten it.

```
## Setting the coordinate system

> st_set_crs <- (sites_sf, value = 4326) # 4326 is the EPSG code for WGS84

### 'value' can be a character string, an EPSG code, a proj4 string, or a crs object
### (see link above).
### WGS84 is a commonly used geographic coordinate system (esp. for GPS points)

## It's easier to set the coordinate system when first making the sf object:

> sites_sf <- st_as_sf(sitesdf, coords = c("lon", "lat"), crs = 4326)
```

Reprojecting data

At some point you'll need data to be in a different coordinate system. Maybe you need to align different datasets. Some functions require maps to be in a projected (flat) crs. When working with distances you've got to bear units in mind: are we working with meters, or angular units like decimal degrees?

Reprojecting your data to a new crs is easy:

```
> sites_repro <- st_transform(sites_sf, crs = "ESRI:102008") # This is an Albers equal
# area projection for North America just as an example

# or if you want to align two spatial datasets, transform one to the crs of the other:

> sites_repro <- st_transform(sites_sf, crs = st_crs(other_data))
```

You can compare the new coordinates to see the transformation:

```
> st_coordinates(sites_sf)
        X    Y
 1 -111.9 40.8
 2 -111.0 41.0
 3 -109.6 38.5
 4 -113.4 38.0
 5 -108.6 39.0
```

```
> st_coordinates(sites_repro)
         X           Y
 1 -1256289   200009.24
 2 -1182071   211734.27
 3 -1114689   -96688.56
 4 -1433769 -103678.74
 5 -1025425   -49646.73
```

Some useful sf functions

Manipulate spatial data with **tidyr + dplyr**

You can basically treat an sf object like any ol' dataframe

```
# We only want Utah:

> utah <- filter(states, state == "Utah")

# Select multiple states and create a new column (attribute) too:

> utco <- states %>%
            filter(state %in% c("Utah", "Colorado")) %>% # yes this is the original df
            mutate(new_column = "whatever")
```

Get the spatial extent of your object with **st_bbox()**

```
# This draws a box around all features and returns the coordinates of the vertices

> st_bbox(states)

> st_bbox(sites_sf)
```

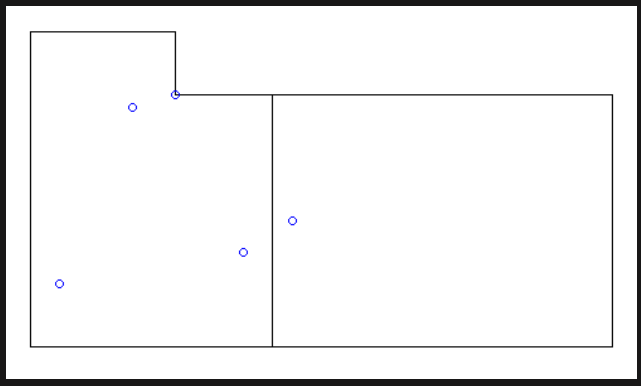
Create a buffer zone around features with **st_buffer()**

```
# st_buffer gives you a warning if the units are decimal degrees, so
# double check your units:
> st_crs(sites_repro)$units


# For a 10 km buffer around sites:
# distance is in meters for our equal area crs
> sites_buff <- st_buffer(sites_repro, dist = 10000) # now these are polygons
```

Some useful sf functions

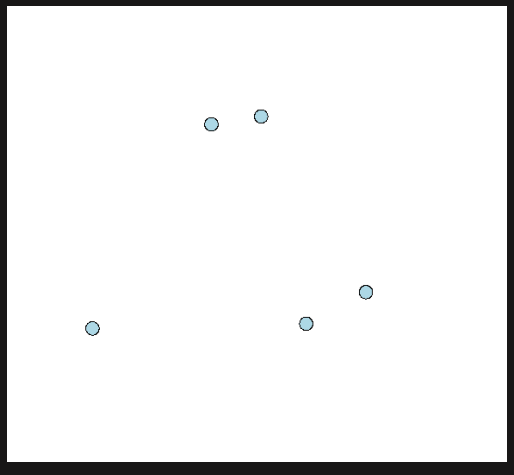
Quickly plot data with base R **plot()**

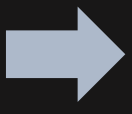
```
# Use plot with st_geometry() to plot ONLY the geometry of an object
> plot(st_geometry(states))
# plot(states) will give you a plot for each attribute

# Add sites (points) on top of the lake plot:
> plot(st_geometry(sites_sf), col = "blue", add = TRUE)
```

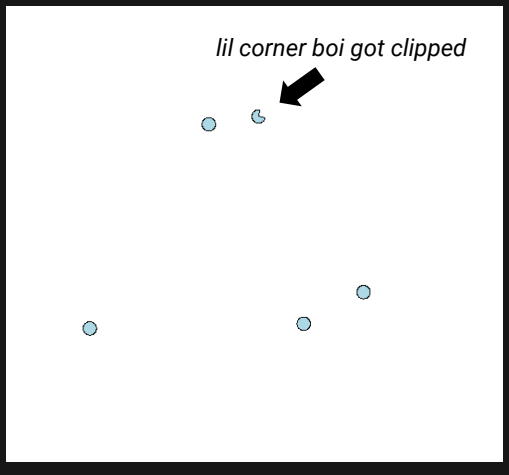


Clip polygons with **st_intersection()**

```
# First reproject states to be in the same flat projection as the sites:
> states_repro <- st_transform(states, crs(sites_buff))
# Clip buffered sites using state boundaries:
> clip <- st_intersection(sites_buff, states_repro)
```







Some useful sf functions

Join data based on location, aka **spatial join** in GIS

There are several options for how to join the data. For this example say we want to assign points to states. We can use **st_intersects()**. If you wanted sites within 10 meters of a state, you could use st_within_distance(). Or if you wanted only sites *inside* states, st_contains().

```
> sites_states <- st_intersects(sites_repro, states_repro)
Sparse geometry binary predicate list of length 5, where the predicate was `intersects`
1: 2
2: 2
3: 2
4: 2
5: 1
```

This means the first 4 points intersect polygon 2 (Utah), and point 5 intersects polygon 1 (Colorado)

You can finagle this info into a new "state" column for your sites:

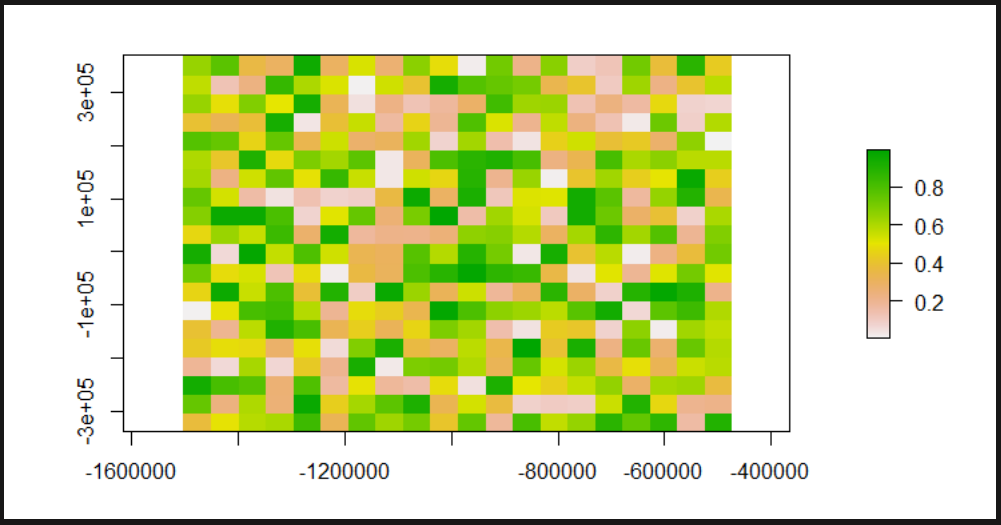
```
> siteswst <- sites_sf %>%
            mutate(state = as.data.frame(sites_states)[,2]) %>%
            mutate(state = states$state[state])
```

This is a little ugly there's probably a better way to do it... Basically it takes that 2nd column with the polygon IDs which are row IDs (2, 2, 2, 2, 1) and pulls the name of the state from the 'states' object depending on the row number.

Some useful raster functions

We'll make a random raster just to use here:

```
# create a 20x20 grid...
a_raster <- raster(ncol=20, nrow=20)

# give it the same spatial extent as 'states_repro'
extent(a_raster) <- extent(states_repro)

# give it the same flat coordinate system as the buffered sites
# raster needs crs in 'proj4string' format, doesn't take EPSG codes I don't think
crs(a_raster) <- st_crs(sites_buff)$proj4string

# give each grid cell a random value
values(a_raster) <- runif(ncell(a_raster))

# Plot it:
plot(a_raster)
```



Some useful raster functions

Extract data from a raster with **extract()**

```
# Extract the mean values of raster cells within your buffer zone:
# Note: ensure your raster and zone data have the same crs (you'll get an error if not)
> ex <- raster::extract(a_raster, sites_buff, fun = mean, na.rm = TRUE, df = TRUE)
# This ignores NA values and returns a dataframe; the default df = FALSE returns a list

# If you don't supply a fun = argument, it will return the values of every cell in each
# zone. This is useful for categorical rasters.

# Attach these values to your data:
> sites_buff$value <- ex
```

Extract data from multiple rasters simultaneously with raster **stacks**

```
# Rasters must have the same extent and resolution before you can stack them
> r_stack <- stack(raster1, raster2, raster3)

# Extract the mean values from the stack:
> ex <- raster::extract(r_stack, sites_buff, fun = mean)

# *Note: if you have tidyr (or tidyverse) loaded you must specify that you want the
# extract function from the raster package with 'raster::extract'
```

Some useful raster functions

Change raster resolution with **aggregate** and **disaggregate**

```
# Reduce resolution by 2-fold:
> r2 <- aggregate(a_raster, fact=2, mean) # this will aggregate cells by averaging

# For categorical rasters, use 'modal' instead of 'mean' to take the mode (most common
# cell value in aggregate area)
```

Clip raster to polygon shape, or another raster

```
> r_clip <- mask(a_raster, states_repro) # sets values outside the boundary to NA
```

Crop raster

```
> r_crop <- crop(a_raster, extent(sites_buff))

# extent() gives you an extent object in the raster package, same as st_bbox in sf.

# You can specify your own extent, or if you put in another spatial object, the
# function will automatically take its extent.

# This crops a_raster to the same extent as our sites
```

Troubleshooting

A super common error you'll come across working with shapefiles is one about invalid geometries. This can happen when you're clipping or otherwise messing with polygon shapes, especially if they're complex (like squiggly lake boundaries). It looks something like:

```
## Error in CPL_geos_op2(op, st_geometry(x), st_geometry(y)): Evaluation error:
TopologyException: Input geom 1 is invalid: Ring Self-intersection at or near point ...
```

But sometimes R (and GIS) kicks up a fuss about this even when there's not really an issue. There's a well-known GIS hack to try and overcome it: add a 0 width buffer. This is always a good thing to try if you're getting weird errors about geometry or topology.

```
> sites_buff <- st_buffer(sites_buff, dist = 0)
```

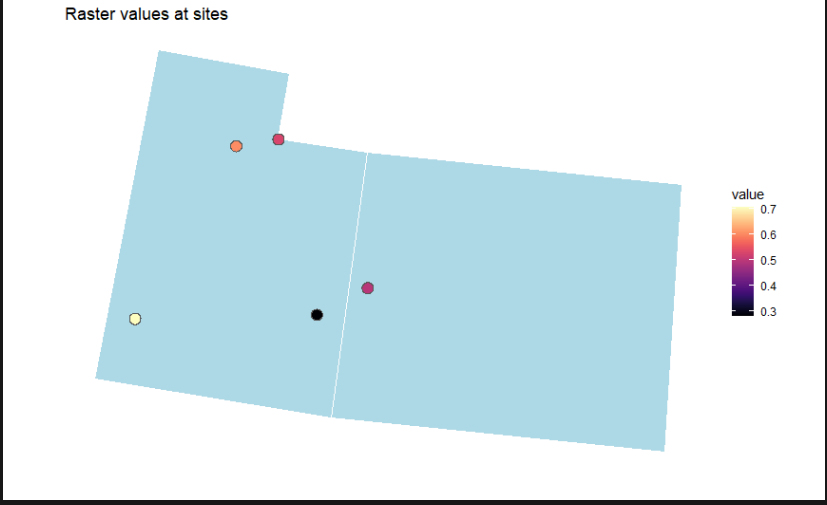
If that doesn't do the trick, try simplifying your polygons a teensy bit after adding the 0 buffer.

```
> sites_buff <- st_simplify(sites_buff, dTolerance = 1)
# dTolerance is a distance measured in the units of your crs, so make sure you're using
# a projected system. Higher numbers will simplify more.
```

Mapmaking

sf objects are super easy to plot in ggplot. This is a nice detailed guide to making maps. I'll just throw down some basics here...

```
> ggplot() +
  geom_sf(data = states_repro, color = "#FFFFFF", fill = "lightblue") +
  geom_sf(data = sites_buff, aes(fill = value)) +
  scale_fill_viridis(option = "magma") + # requires 'viridis' package
  theme_minimal() +
  labs(x = "", y = "", title = "Raster values at sites", fill = "value") +
  theme(line = element_blank(), # remove axis lines
        axis.text = element_blank()) # remove axis text
```



geom_sf() is what you need to plot sf objects. Specify outline **color** (color) and **fill** color; you can also fill polygons according to an attribute (here, 'value' that we extracted a couple pages ago).

coord_sf() sets the projection, zoom, and map center

Mapmaking

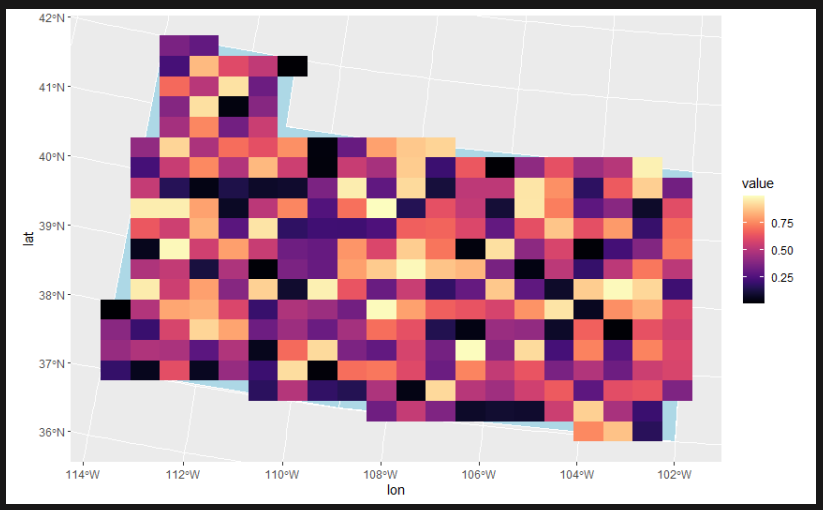
Plot raster data with **geom_tile()**

But we have do to a little prep first...

```
# We'll work with the raster we clipped to be the same shape as the states:
r_clip <- mask(a_raster, states_repro)

# Now we have to turn it into a dataframe that ggplot can use
rasdf <- rasterToPoints(r_clip) # converts raster (cells) to just points with lon + lat
rasdf <- data.frame(rasdf) # turn it into a dataframe
names(rasdf) <- c("lon", "lat", "value") # name columns whatever you want
```

Now it's ready to plot:

```
> ggplot() +
  geom_sf(data = states_repro, color = "#FFFFFF", fill = "lightblue") +
  geom_tile(data=rasdf, aes(x=lon, y=lat, fill = value)) +
  scale_fill_viridis(discrete = FALSE, option = "magma")
```



Looks a little funky because our cells are so huge (we only made a 20x20 cell raster). Probably datasets you're using will have a finer resolution

Free data!

Good places to get base maps, or boundaries for clipping, etc.:

- Packages for world maps:
  - **'maptools'**

```
> worldmap <- data("wrld_simpl", package = "maptools")
```

  - **'rnaturalearth' & 'rnaturalearthdata'**

```
> worldmap <- ne_countries(scale = "medium") # specify more detailed or simpler maps
```

You can filter these data to get sf objects for specific countries or regions.

- US geographic data from the census (TIGER line/shapefiles)
  - **'tigris'**

```
> states <- states()

# There are a lot of options in this package. You can get states, counties, census
# tracts and blocks, school districts, water features in counties...
```

Resources

- **Geocomputation with R:** https://geocompr.robinlovelace.net/intro.html
- **GIS and Spatial Analysis with R:** https://mgimond.github.io/MEGUG2016/Tutorial.html
- **Simple Features for R:** https://r-spatial.github.io/sf/articles/sf1.html
- **Making Maps with R:** https://www.molecularecologist.com/2012/09/18/making-maps-with-r/