EDS 223: week 5 lab notes

Ruth Oliver

2022-10-24

introduction

In this lab, we'll explore the basics of working with raster data, including attribute, spatial, and geometry operations. This lab follows chapters 3, 4, and 5 of Geocomputation with R by Robin Lovelace.

prerequisites

```
library(terra)
library(dplyr)
library(spData)
library(spDataLarge)
library(tmap)
```

manipulating raster objects

Raster data represents continuous surfaces, as opposed to the discrete features represented in the vector data model. Here we'll learn how to create raster data objects from scratch and how to do basic data manipulations.

Let's create a **SpatRaster** object using a digitial elevation model for Zion National Park.

```
raster_filepath <- system.file("raster/srtm.tif", package = "spDataLarge") #establish file path
my_rast <- rast(raster_filepath) # create raster object

class(my_rast) # test class of raster object

my_rast # gives summary information

plot(my_rast)
```

We can also create rasters from scratch using the **rast()** function. Here we create 36 cells centerd around (0,0). By default the CRS is set to WGS84, but we could change this with the "crs" argument. Because we are working in WGS84, the resolution is in units of degrees. **rast()** fills the values of the cells row-wise starting in the upper left corner.

```
new_raster = rast(nrows = 6, ncols = 6, resolution = 0.5,
                  xmin = -1.5, xmax = 1.5, ymin = -1.5, ymax = 1.5,
                  vals = 1:36)
```

```
tm_shape(new_raster) +
  tm_raster()
```

The SpatRaster class can also handle multiple layers.

```
multi_raster_file <- system.file("raster/landsat.tif", package = "spDataLarge")
multi_rast <- rast(multi_raster_file)
multi_rast

nlyr(multi_rast) # test number of layers in raster object
```

We can subset layers using either the layer number or name

```
multi_rast3 <- subset(multi_rast, 3)
multi_rast4 <- subset(multi_rast, "landsat_4")
```

We can combine SpatRaster objects into one, using the c function

```
multi_rast34 <- c(multi_rast3, multi_rast4)
```

Let's create an example raster for elevation

```
elev <- rast(nrows = 6, ncols = 6, resolution = 0.5,
             xmin = -1.5, xmax = 1.5, ymin = -1.5, ymax = 1.5,
             vals = 1:36)
```

Rasters can also hold categorical data. Let's create an example raster for soil types.

```
grain_order <- c("clay", "silt", "sand") # set soil types
grain_char <- sample(grain_order, 36, replace = TRUE) # randomly create character string of soil types
grain_fact <- factor(grain_char, levels = grain_order) # convert to factors

grain <- rast(nrows = 6, ncols = 6, resolution = 0.5,
              xmin = -1.5, xmax = 1.5, ymin = -1.5, ymax = 1.5,
              vals = grain_fact)
```

raster subsetting

We can index rasters using, row-column indexing, cell IDs, coordinates, other spatial objects.

```
# row 1, column 1
elev[1, 1]
# cell ID 1
elev[1]
```

If we had a two layered raster, subsetting would return the values in both layers.

```
two_layers <- c(grain, elev)
two_layers[1]
```

We can also modify/overwrite cell values.

```
elev[1, 1] = 0
elev[]
```

Replacing values in multi-layer rasters requires a matrix with as many columns as layers and rows as replaceable cells.

```
two_layers[1] <- cbind(c(1), c(4))
two_layers[]
```

summarizing raster objects

We can get info on raster values just by typing the name or using the summary function.

```
elev
summary(elev)
```

We can get global summaries, such as standard deviation.

```
global(elev, sd)
```

Or we can use **freq()** to get the counts with categories.

```
freq(grain)

hist(elev)
```

spatial subsetting

We can move from subsetting based on specific cell IDs to extract info based on spatial objects.

To use coordinates for subsetting, we can "translate" coordinates into a cell ID with the **terra** function **cellFromXY()** or **terra::extract()**.

```
id <- cellFromXY(elev, xy = matrix(c(0.1, 0.1), ncol = 2))
elev[id]
# the same as
terra::extract(elev, matrix(c(0.1, 0.1), ncol = 2))
```

Raster objects can also subset with another raster object. Here we extract the values of our elevation raster that fall within the extent of a masking raster.

```
clip <- rast(xmin = 0.9, xmax = 1.8, ymin = -0.45, ymax = 0.45,
             resolution = 0.3, vals = rep(1, 9))

elev[clip]

# we can also use extract
terra::extract(elev, ext(clip))
```

In the previous example, we just got the values back. In some cases, we might want the output to be the raster cells themselves. We can do this use the "[" operator and setting "drop = FALSE"

This example returns the first 2 cells of the first row of the "elev" raster.

```
elev[1:2, drop = FALSE]
```

Another common use of spatial subsetting is when we use one raster with the same extent and resolution to mask the another. In this case, the masking raster needs to be composed of logicals or NAs.

```
# create raster mask of the same resolution and extent
rmask <- elev
# randomly replace values with NA and TRUE to use as a mask
values(rmask) <- sample(c(NA, TRUE), 36, replace = TRUE)

# spatial subsetting
elev[rmask, drop = FALSE]    # with [ operator
mask(elev, rmask)            # with mask()
```

We can also use a similar approach to replace values that we suspect are incorrect.

```
elev[elev < 20] = NA
```

map algebra

Here we define map algebra as the set of operations that modify or summarize raster cell values with reference to surrounding cells, zones, or statistical functions that apply to every cell.

local operations

Local operations are computed on each cell individually. We can use oridinary arithemetic or logical statements.

```
elev + elev
elev^2
log(elev)
elev > 5
```

We can also classify intervals of values into groups. For example, we could classify a DEM into low, middle, and high elevation cells

- first we need to construct a reclassification matrix
- the first column corresponds to the lower end of the class
- the second column corresponds to the upper end of the class
- the third column corresponds to the new value for the specified ranges in columns 1 and 2

```
rcl <- matrix(c(0, 12, 1, 12, 24, 2, 24, 36, 3), ncol = 3, byrow = TRUE)
rcl

# we then use this matrix to reclassify our elevation matrix
recl <- classify(elev, rcl = rcl)
recl
```

For more efficient processing, we can use a set of map algebra functions. - **app()** applies a function to each cell of a raster to summarize the values of multiple layers into one layer - **tapp()** is an extension of "app()" that allows us to apply on operation on a subset of layers - **lapp()** allows us to apply a function to each cell using layers as arguments

We can use the **lapp()** function to compute the Normalized Difference Vegetation Index (NDVI). Let's calculate NDVI for Zion National Park using multispectral satellite data.

```
multi_raster_file <- system.file("raster/landsat.tif", package = "spDataLarge")
multi_rast <- rast(multi_raster_file)
```

We need to define a function to calculate NDVI.

```
ndvi_fun = function(nir, red){
  (nir - red) / (nir + red)
}
```

So now we can use **lapp()** to calculate NDVI in each raster cell. To do so, we just need the NIR and red bands.

```
ndvi_rast <- lapp(multi_rast[[c(4, 3)]], fun = ndvi_fun)

tm_shape(ndvi_rast) +
  tm_raster()
```

focal operations

Local operations operate on one cell, though from multiple layers. Focal operations take into account a central (focal) cell and its neighbors. The neighborhood (or kernel, moving window, filter) can take any size or shape. A focal operation applies an aggregation function to all cells in the neighborhood and updates the value of the central cell before moving on to the next central cell

We can use the **focal()** function to perform spatial filtering. We define the size, shape, and weights of the moving window using a matrix. Here we find the minimum.

```
elev <- rast(system.file("raster/elev.tif", package = "spData"))

r_focal <- focal(elev, w = matrix(1, nrow = 3, ncol = 3), fun = min)

plot(elev)
plot(r_focal)
```

zonal operations

Similar to focal operations, zonal operations apply an aggregation function to multiple cells. However, instead of applying operations to neighbors, zonal operations aggregate based on "zones". Zones can are defined using a categorical raster and do not necessarily have to be neighbors

For example, we could find the average elevation for different soil grain sizes.

```
zonal(elev, grain, fun = "mean")
```

geometric operations

When merging or performing map algebra, rasters need to match in their resolution, projection, origin, and/or extent

In the simplest case, two images differ only in their extent. Let's start by increasing the extent of a elevation raster.

```
elev = rast(system.file("raster/elev.tif", package = "spData"))
elev_2 = extend(elev, c(1, 2)) # add one row and two columns

plot(elev)
plot(elev_2)
```

Performing algebraic operations on objects with different extents doesn't work.

```
elev + elev_2
```

We can align the extent of the 2 rasters using the extend() function. Here we extend the elev object to the extent of elev_2 by adding NAs.

```
elev_4 <- extend(elev, elev_2)
```

the origin function returns the coordinates of the cell corner closes to the coordinates (0,0). We can also manually change the origin.

```
origin(elev_4)
origin(elev_4) <- c(0.25, 0.25)
origin(elev_4)
```

aggregation and disaggregation

Faster datasets can also differ in their resolution to match resolutions we can decrease the resolution by aggregating or increase the resolution by disaggregating.

Let's start by changing the resolution of a DEM by a factor of 5, by taking the mean.

```
dem <- rast(system.file("raster/dem.tif", package = "spDataLarge"))
dem_agg <-  aggregate(dem, fact = 5, fun = mean)

plot(dem)
plot(dem_agg)
```

We have some choices when increasing the resolution. Here we try the bilinear method.

```
dem_disagg <- disagg(dem_agg, fact = 5, method = "bilinear")
identical(dem, dem_disagg)

plot(dem_disagg)
```

resampling

Aggregation/disaggregation work when both rasters have the same origins. What do we do in the case where we have two or more rasters with different origins and resolutions? Resampling computes values for new pixel locations based on custom resolutions and origins.

In most cases, the target raster would be an object you are already working with, but here we define a target raster.

```
target_rast <- rast(xmin = 794600, xmax = 798200,
                    ymin = 8931800, ymax = 8935400,
                    resolution = 150, crs = "EPSG:32717")

dem_resampl <- resample(dem, y = target_rast, method = "bilinear")

plot(dem)
plot(dem_resampl)
```