Spatial Data in R!

Justin French

2024-10-07

## Spatial Packages in R: GIS done right

The two most prominent R packages for spatial data are sf (fro vector data) and terra for raster data. Both packages utilize modern computational developments to make working woith spatial data easier, while reducing the intermediate steps to deal with common tasks. They have made coordinate reference systems, geoprocessing, and raster algebra much much simpler than previous packages, including other software packages (e.g., ArcMap and QGIS).

The s and f in sf stand for **simple features**. Simple features are actually an ISO standard data encoding for vector data. They are the guts behind most common vector data formats, including shapefiles. This makes the data more compact and easy to translate across platforms. This has the added benefit of turning vector files into data frames.

The terra package is the next evolution of the original raster data package, raster. This package revised the internal functionality of raster to treat files as if they were on a server, even when stored locally. This makes raster operations dramatically faster, even with relatively light duty computer and large files.

# install.packages(c('sf','sfheaders','terra'))

## Vector Data with sf

The sf package has some nice benefits beyond the things that happen under the hood. To see these, we need to load some data. Let’s grab the counties data set from the data subdirectory:

library(sf);library(sfheaders)  
  
 dat.counties <-  
 st\_read('data/County.shp', quiet = TRUE)

If you use the View() function to look at a simple feature, it looks almost exactly like a data frame. All of the attribute data for a file are treated like any other column in any other data frame. The only difference is the geometry column, which has the data type sfc\_geometry. This allows you to encode all of the spatial information about a distinct feature into a cell of a data frame. It’s really just a new data type.

An sf boject has *two* classes. Of course, it has class sf, but it also has the class data.frame:

class(dat.counties)

## [1] "sf" "data.frame"

This might seem like a very academic thing, but iot gives you a wide set of tools to do GIS work efficiently. For example, if you were to take a data set of all the counties in Texas and reduce it to just a single county, how would you approach that in a point-and-click GIS program? (The answer is clunkily).

In R, it’s as easy as subsetting a data frame, because simple feature inherit all of a data.frame’s methods. Say we only want Coryell County, all we have to do is make a logical subset of dat.counties:

dat.counties[dat.counties$CNTY\_NM == 'Coryell',]

## Simple feature collection with 1 feature and 10 fields  
## Geometry type: POLYGON  
## Dimension: XY  
## Bounding box: xmin: -98.17972 ymin: 31.06953 xmax: -97.4188 ymax: 31.71113  
## Geodetic CRS: WGS 84  
## FID CMPTRL\_CNT CNTY\_NM DPS\_CNTY\_N FIPS\_ST\_CN TXDOT\_CNTY TXDOT\_DIST GID  
## 133 133 50 Coryell 50 48099 50 9 113  
## SHAPE\_Leng SHAPE\_Area geometry  
## 133 2.053109 0.2595117 POLYGON ((-97.68862 31.7092...

Say we only want to work with the Trans-Pecos counties. We can look for all the counties in a set and overwrite dat.counties with just the ones we want:

dat.counties <-  
 dat.counties[dat.counties$CNTY\_NM %in%  
 c('El Paso', 'Hudspeth','Culberson','Reeves',  
 'Pecos', 'Terrell','Brewster',  
 'Presidio','Jeff Davis'),]

Truth in lending, we started with this operation to reduce demand on our RAM. Let’s get back to tearing appart the sf object. There are really only a couple of things that make the sf object unique. One is, of course, the geometry column, but the other is that it has a coordinate reference system (CRS). You can check the coordinate reference system of an sf object with the st\_crs() method:

st\_crs(dat.counties)

## Coordinate Reference System:  
## User input: WGS 84   
## wkt:  
## GEOGCRS["WGS 84",  
## DATUM["World Geodetic System 1984",  
## ELLIPSOID["WGS 84",6378137,298.257223563,  
## LENGTHUNIT["metre",1]]],  
## PRIMEM["Greenwich",0,  
## ANGLEUNIT["degree",0.0174532925199433]],  
## CS[ellipsoidal,2],  
## AXIS["latitude",north,  
## ORDER[1],  
## ANGLEUNIT["degree",0.0174532925199433]],  
## AXIS["longitude",east,  
## ORDER[2],  
## ANGLEUNIT["degree",0.0174532925199433]],  
## ID["EPSG",4326]]

The sf package adopted the proj7 standard for coordinate reference systems, which uses an encoding called **well-known text** (WKT). The reasons for this are tedious and boring, but it makes our life easier because they include the EPSG idenintifyer code for all standard CRSs explicitly. If we need to transform (or project) a spatial data set, this makes it very easy with the st\_transform() method:

dat.counties <-  
 st\_transform(dat.counties,  
 crs = st\_crs(32613))

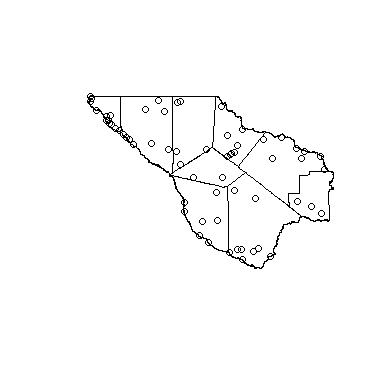
## Geoprocessing with sf

To get into real geoprocessing, we need another data set to play with. Let’s get a data set of Texas cities and then clip it to the Trans-Pecos:

dat.cities <-  
 st\_read('data/City.shp', # Specify were file is   
 quiet = TRUE) # Tell it not to print stuff  
  
 dat.cities <-  
 st\_transform(dat.cities, # Spec. file to transform  
 st\_crs(dat.counties)) # Get CRS from existing obj.  
  
 dat.cities <-  
 st\_intersection(dat.cities, # File to cut  
 dat.counties) # File to cut with

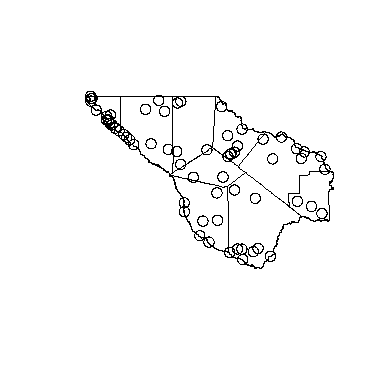
In the code above, we saw that R does not try to think for you and does not do heads-up transformation or projection like ESRI and QGIS products do. Instead, you will get an error if your CRSs do not line up. So, we simply project the dat.cities obect to be the same CRS as the dat.counties one, then we fed both objects into the st\_intersection() function to “clip” the cities to just the Trans-Pecos. The result is this:

plot(st\_geometry(dat.counties))  
 plot(st\_geometry(dat.cities),   
 add = TRUE)



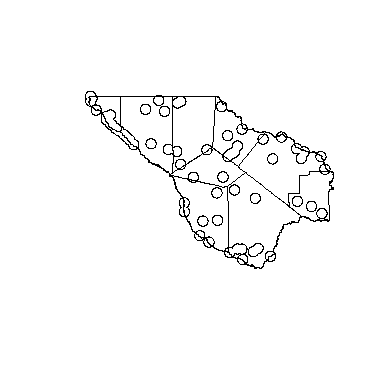
There are many other geoprocessing functions available to you, but we’ll run through a few of the common ones. We often want to buffer certain features for a variety of things. This is easy to do with the st\_buffer() functions:

plot(st\_geometry(dat.counties))  
 plot(st\_geometry(  
 st\_buffer(dat.cities,   
 dist = 10000)  
 ),   
 add = TRUE)



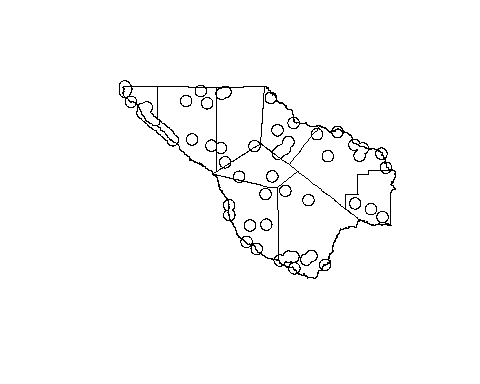
The code above gave us a unique circle for every city. We often want to dissolve these (ESRI term) into continuous polygons when they overlap. This can be done with the st\_union() function:

plot(st\_geometry(dat.counties))  
 plot(st\_geometry(  
 st\_sf(  
 st\_union(  
 st\_buffer(dat.cities,   
 dist = 10000),  
 )  
 )  
 ),   
 add = TRUE)



The operation above got pretty complex, but that’s a good thing. Many of the things we want to do involve steps we often don’t think much about. In the code above, we buffered all the cities by 10km, then we merged the buffers to make a polygon that contained all the are within 10km of a Trans-Pecos “city”. Merging polygons actually nullifies the validity of their attributes, so R (really sf) reduces the output of st\_union() to just a geometry, eliminating invalid attributes. We used st\_sf() to **coerce** the class of the output back to sf. This allows us to attach attributes to it. The example above is a base R approach to nested operations. However, you need to be aware of an alternative, which is the **piped** approach that the tidyverse uses. If we do the same thing with the piped approach, it would look like this:

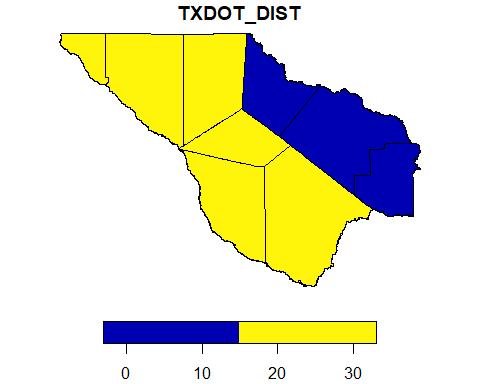
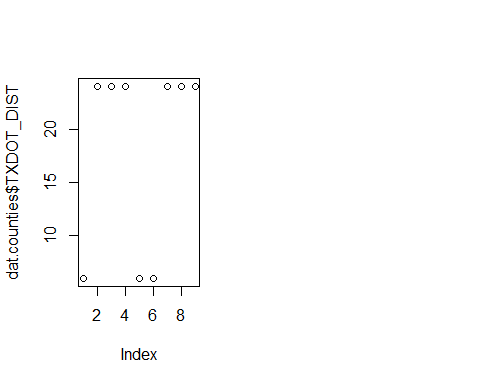
plot(st\_geometry(dat.counties))  
 plot(  
 dat.cities %>%  
 st\_buffer(dist = 10000) %>%  
 st\_union() %>%  
 st\_sf() %>%  
 st\_geometry(),  
 add = TRUE)



Pretty much all of the normal GIS geoprocessing operations are listed on the help page for the st\_buffer() function. These are collectively known as **unary operations** and cover all the bases. Their use follows the principles of all the functions we saw above, and are pretty easy to use. See that help page for details, or ask Dr. Google.

The last function that we’ll talk about today is the st\_geometry() method. It retrieves the geometry from an sf object to provide that to generic methods like plot(). There is an important reason for this; actions on the geometry are different from actions on the data as a whole. Remember, geometries have attributes. If we just plot a simple feature, we are really plotting its attributes. However, we have to be careful with our syntax, like such as:

par(mfrow=c(1,2))  
 plot(dat.counties$TXDOT\_DIST) # Does not do what we want  
 plot(dat.counties[,"TXDOT\_DIST"]) # Does do what we want



par(mfrow=c(1,1))

There is a good reason for this; if we really are focussed on the data (not just the space) we need to be able to summarize the data in non-spatial ways easily. If you pull a column by name from a sf object, it comes out as a vector. This makes summaries and analyses of these data easier (e.g., say we wanted a histogram), but it does not preserve the geometry for that purpose.

The second line above is a subset by name, not a column retrieval. That *does* preserve the geometry, and the plot method will still make a map. This distinction does not matter for other data frames, but it is a big deal for sf data frames.

## Contructing Your Own sf Object

So there are many occasions where you get spatial data in a non-spatial format. This is common with historic data and when working with non-research partners. This can be tricky in sf proper. However, the sfheaders package has several **wrapper functions** that make it easy. A wrapper function is just a function that makes using another function easier. R is full of wrapper functions.

To demonstrate this, let’s make some quasi-random points somewhere out in the ether:

dat.imaginary <-  
 data.frame(name = c('Chewie', 'Han', 'Luke'),  
 x = c(-10.0506, -10.0678, 13.012),  
 y = c(2.070, 2.065, 5.896))

To get this into a sf object, the sfheaders package has a function that is quite sneakily named sf\_point() (note it isn’t st\_; that’s the sneaky part). All it needs to know is which column gives X-coordinates and which give Y-coordinates:

dat.imaginary <-   
 sf\_point(dat.imaginary,  
 x = 'x', y = 'y',  
 keep = TRUE)

If we check the CRS of dat.imaginary we will find there isn’t one. The CRS in a sf object is really just an attribute, kinda like colnames and rownames. We can set it in exactly the same way:

st\_crs(dat.imaginary) <- st\_crs(4326)

It’s very important to remember that the above action does not transform or project coordinates, it just states what system they are in. It does not replace st\_transform().

Now we need to talk about sf geometries again. We’ve only made a POINT geometry, but there are certainly others. The sf package supports the geometry classes POINT, MULTIPOINT, LINESTRING, MULTILINESTRING, POLYGON, MULTIPOLYGON, and GEOMETRY COLLECTION. It’s important to know that these exist and that have different uses. But what happens when you need to change from one type, to another?

If you want the polygon that connects all the places where we found the Millennium Falcon’s passengers (like, to search for wreckage), you would want to cast these points to a polygon. You could just make the points into a polygon instead of a point object, like so:

sf\_polygon(  
 data.frame(name = c('Chewie', 'Han', 'Luke',  
 'Obi Wan', 'C-3PO', 'Leia'),  
 x = c(-10.0506, -10.0678, 13.012,  
 -9.000, 13.015, -10.678),  
 y = c(2.070, 2.065, 5.896,  
 2.686, 6.253, 2.064),  
 group = c(1,1,1,  
 2,2,2)),  
 x = 'x',y = 'y',  
 polygon\_id = 'group'  
 )

## Simple feature collection with 2 features and 1 field  
## Geometry type: POLYGON  
## Dimension: XY  
## Bounding box: xmin: -10.678 ymin: 2.064 xmax: 13.015 ymax: 6.253  
## CRS: NA  
## group geometry  
## 1 1 POLYGON ((-10.0506 2.07, -1...  
## 2 2 POLYGON ((-9 2.686, 13.015 ...

So, in the example above we added a second group of passengers to demonstrate how to make multiple polygons. You just need a column that identifies them. But, that still does not address how to cast one geometry to another. This is done with st\_cast():

dat.imaginary <-   
 sf\_point(data.frame(name = c('Chewie', 'Han', 'Luke',  
 'Obi Wan', 'C-3PO', 'Leia'),  
 x = c(-10.0506, -10.0678, 13.012,  
 -9.000, 13.015, -10.678),  
 y = c(2.070, 2.065, 5.896,  
 2.686, 6.253, 2.064),  
 group = c(1,1,1,  
 2,2,2)),  
 x = 'x', y = 'y', keep = TRUE)

## Working With Rasters in Terra

terra is designed to be user-friendly. The programmers reasonable approximated this goal. However, you may remember our discussion of namespace conflicts yesterday, which centered on terra and raster. We’ll see that in action today.

An important thing to note is that terra has its own object model for vector data. As far as I know, nobody uses it other than to interact with raster data. sf has been the clear winner for vector data. This does make it complicated when we need to crop and mask rasters. However, everything else is easy.

## The spatRaster

We will load the first raster layer from our data folder to initialize a spatRaster object. But first, let’s load the package:

library(terra)

## terra 1.7.29

Working with large raster data sets usually entails a convoluted file structure, so we need a smart way to work across all these files. Let’s get the first layer to see how we need to start:

# The hard-coding way  
 stack.temp <-   
 rast(paste('data',  
 'PRISM\_tmean\_stable\_4kmD2\_20200101\_bil',  
 'PRISM\_tmean\_stable\_4kmD2\_20200101\_bil.bil',  
 sep = '/'))  
  
 # The smart way  
 files <-   
 list.files(pattern = '.bil$', # Use a regex statement to  
 recursive = TRUE) # find all files ending in .bil  
   
 stack.temp <-  
 rast(files[1])

In the first example above, we loaded a file manually. This sucks. When working with raster data sets, we usually need more than one raster. If we have our raster data stored in a logical and well-organized fashion, we can use file system functions like list.files() in combination with search terms to find a list of the files we want. We can then load from that, which we did in the second example above. Now we can tear apart a spatRaster,

An interesting thing about this spatRaster is that, while it does appear in our environment, it is not in our RAM. The only thing in memory is a pointer to a cache file that terra can work with on disk (not in RAM). Basically, we didn’t load a raster, we connected to it. This makes things fast and stable, but scary:

# Explore the bowels of S4  
 str(stack.temp@ptr)

## Reference class 'Rcpp\_SpatRaster' [package "terra"] with 20 fields  
## $ dataType : chr "FLT4S"  
## $ depth : num 0  
## $ extent :Reference class 'Rcpp\_SpatExtent' [package "terra"] with 2 fields  
## ..$ valid : logi TRUE  
## ..$ vector: num [1:4] -125 -66.5 24.1 49.9  
## ..and 28 methods, of which 14 are possibly relevant:  
## .. align, as.points, ceil, compare, deepcopy, finalize, floor, initialize,  
## .. intersect, round, sample, sampleRandom, sampleRegular, union  
## $ hasRange : logi TRUE  
## $ hasTime : logi FALSE  
## $ hasUnit : logi FALSE  
## $ hasValues: logi TRUE  
## $ inMemory : logi FALSE  
## $ messages :Reference class 'Rcpp\_SpatMessages' [package "terra"] with 2 fields  
## ..$ has\_error : logi FALSE  
## ..$ has\_warning: logi FALSE  
## ..and 18 methods, of which 4 are possibly relevant:  
## .. finalize, getError, getWarnings, initialize  
## $ names : chr "PRISM\_tmean\_stable\_4kmD2\_20200101\_bil"  
## $ origin : num [1:2] -0.0208 0.0208  
## $ range\_max: num 24.4  
## $ range\_min: num -23.8  
## $ res : num [1:2] 0.0417 0.0417  
## $ rgb : logi FALSE  
## $ rgbtype : chr ""  
## $ time : num 0  
## $ timestep : chr "seconds"  
## $ timezone : chr ""  
## $ units : chr ""  
## and 258 methods, of which 244 are possibly relevant:  
## addSource, adjacent, adjacentMat, aggregate, align, allnan, anynan, apply,  
## arith\_m, arith\_numb, arith\_rast, as\_lines, as\_multipoints, as\_points,  
## as\_points\_value, as\_polygons, atan2, bilinearValues, boundaries, buffer,  
## canProcessInMemory, cellFromRowCol, cellFromRowColCombine, cellFromXY,  
## cells\_notna, cells\_notna\_novalues, chunkSize, clamp, clamp\_raster, clamp\_ts,  
## classify, colFromX, collapse\_sources, combineCats, combineSources,  
## compare\_geom, costDistance, couldBeLonLat, count, countnan, cover,  
## createCategories, crop, crop\_mask, cum, deepcopy, dense\_extent,  
## disaggregate, droplevels, expand, ext\_from\_rc, extCells, extractCell,  
## extractVector, extractVectorFlat, extractXY, filenames, fill\_range,  
## finalize, flip, focal, focalValues, freq, geometry, get\_aggregate\_dims,  
## get\_aggregates, get\_crs, get\_sourcenames, get\_sourcenames\_long, getBands,  
## getBlockSizeR, getBlockSizeWrite, getCategories, getCatIndex, getColors,  
## getDataType, getError, getFileBlocksize, getLabels, getMessage, getNAflag,  
## getRGB, getScaleOffset, getValues, getWarnings, global,  
## global\_weighted\_mean, gridDistance, hardcopy, has\_error, has\_warning,  
## hasCategories, hasColors, hasWindow, hillshade, hsx2rgb, initf, initialize,  
## initv, intersect, is\_false, is\_in, is\_in\_cells, is\_true, isfinite,  
## isGlobalLonLat, isinfinite, isLonLat, isnan, layerCor, logic\_numb,  
## logic\_rast, lyrsBySource, make\_tiles, make\_tiles\_vect, make\_vrt,  
## makeCategorical, mask\_raster, mask\_self, mask\_vector, math, math2,  
## mem\_needs, metadata, mglobal, modal, ncol, nlyr, nlyrBySource, nonan,  
## not\_na, nrow, nsrc, patches, polygonize, proximity, quantile, rapply,  
## rappvals, rastDirection, rastDistance, rasterize, rasterizeGeom,  
## rasterizeLyr, rasterizePointsV, rasterizePointsXY, rasterizeWindow, readAll,  
## readStart, readStop, readValues, rectify, removeCategories, removeColors,  
## removeRGB, removeWindow, replace, replaceCellValues, replaceCellValuesLayer,  
## replaceValues, resample, reverse, rgb2col, rgb2hsx, roll, rotate,  
## rowColFromCell, rowFromY, rst\_area, same, sampleRandomRaster,  
## sampleRandomValues, sampleRegularRaster, sampleRegularValues,  
## sampleRowColRaster, sampleRowColValues, scale, selRange, separate, set\_crs,  
## set\_depth, set\_resolution, set\_sourcenames, set\_sourcenames\_long, set\_units,  
## setCategories, setCatIndex, setColors, setDepth, setLabels, setNAflag,  
## setNames, setRange, setRGB, setScaleOffset, settime, setTime, setUnit,  
## setValues, setValuesRcpp, setValueType, setWindow, shift, sieve, size, sort,  
## sources\_to\_disk, spatinit, stretch, subset, sum\_area, sum\_area\_group,  
## summary, summary\_numb, terrain, transpose, trig, trim, trim1, unique,  
## update\_meta, valueType, vectCells, vectDirectionRasterize,  
## vectDistanceDirect, vectDistanceRasterize, view, warp, where, wincircle,  
## winrect, wmean\_rast, wmean\_vect, writeRaster, writeStart, writeStop,  
## writeValues, xFromCol, xyFromCell, yFromRow, zonal, zonal\_poly,  
## zonal\_poly\_weighted, zonal\_weighted

As far as you need to be concerned, the spatRaster is just a list of rasters. But, it does have attributes. These are under active development and may change (or at least expand) as time goes on. The important attributes are the CRS, the layer name, the layer timestamp, and the extent. Let’s see how to find each of these with their corresponding methods:

terra::crs(stack.temp)

## [1] "GEOGCRS[\"NAD83\",\n DATUM[\"North American Datum 1983\",\n ELLIPSOID[\"GRS 1980\",6378137,298.257222101,\n LENGTHUNIT[\"metre\",1]],\n ID[\"EPSG\",6269]],\n PRIMEM[\"Greenwich\",0,\n ANGLEUNIT[\"Degree\",0.0174532925199433]],\n CS[ellipsoidal,2],\n AXIS[\"longitude\",east,\n ORDER[1],\n ANGLEUNIT[\"Degree\",0.0174532925199433]],\n AXIS[\"latitude\",north,\n ORDER[2],\n ANGLEUNIT[\"Degree\",0.0174532925199433]]]"

Above, we used explicit package notation to prevent a namespace conflict with raster. The crs() method returned WKT, ust like sf, which makes their CRS information completely transferable between the packages.

names(stack.temp)

## [1] "PRISM\_tmean\_stable\_4kmD2\_20200101\_bil"

We can use the generic method names() to look at the names of the layers, which are just the name of the file they came from by default.

terra::time(stack.temp)

## [1] NA

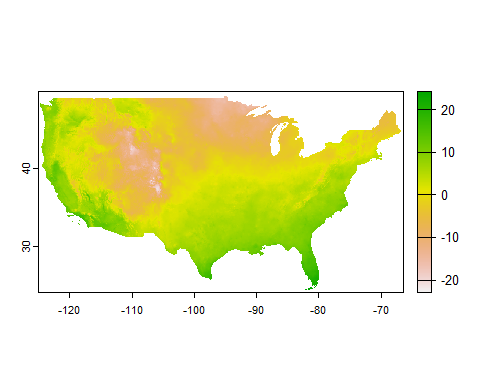
Another attribute with an eye towards raster time series is the time attribute that is unique to spatRasters. At present, it has constraint on time format, or even data type. It will take integers, characters, and POSIXct-class timestamps. Use POSIXct wherever possible.

terra::ext(stack.temp)

## SpatExtent : -125.020833333333, -66.4791666661985, 24.0624999997925, 49.9374999999995 (xmin, xmax, ymin, ymax)

There is a generic plot() method for spatRasters. It’s ugly, and it rearranges graphic parameters without resetting them back to the system default. It will frustrate you. That said, here’s and example of using it:

plot(stack.temp)



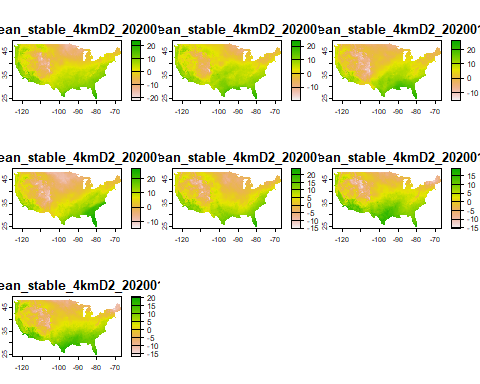
## Building a Stack

We went through fairly tedious pains earlier to make our raster loading method flexible. This was to facilitate adding multiple layers into a stack. Let’s see how we can efficiently stack this initial raster with the rest of the time series data we have:

for(file in files[-1]){  
 add(stack.temp) <- rast(file)  
 };rm(file)

In the chunk above, we looped through all but the first file name and used the add() method to insert additional layers into our spatRaster. That left us with an extra object called file that the loop used, but we don’t need. We added a rm() (remove) function as a second line to toss file when the loop was done with it. Now what does our spatRaster look like?

plot(stack.temp)



We now have seven layers in the stack, but how do we access them? Remember I said that we can think of a spatRaster as a named list of rasters. How do we call an element of a list by name?

stack.temp$PRISM\_tmean\_stable\_4kmD2\_20200101\_bil

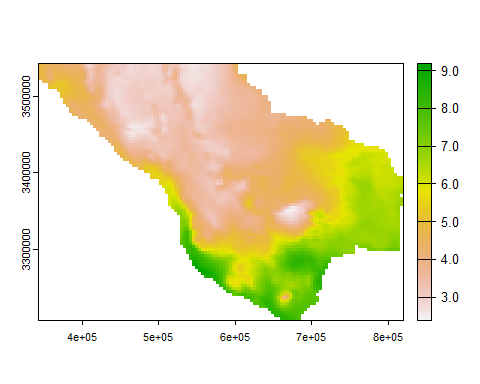
## class : SpatRaster   
## dimensions : 621, 1405, 1 (nrow, ncol, nlyr)  
## resolution : 0.04166667, 0.04166667 (x, y)  
## extent : -125.0208, -66.47917, 24.0625, 49.9375 (xmin, xmax, ymin, ymax)  
## coord. ref. : lon/lat NAD83   
## source : PRISM\_tmean\_stable\_4kmD2\_20200101\_bil.bil   
## name : PRISM\_tmean\_stable\_4kmD2\_20200101\_bil   
## min value : -23.816   
## max value : 24.378

Just like a list! Now that we know that, we might want a little easier name to deal with. Let’s use some text manipulation to make that easier:

names(stack.temp) <-  
 paste0('t',  
 substr(names(stack.temp),  
 start = 26, stop = 33))

Now that the names are easier, let’s do a little raster processing to get them cut down to size. Let’s make them match the extent of our Trans-Pecos counties:

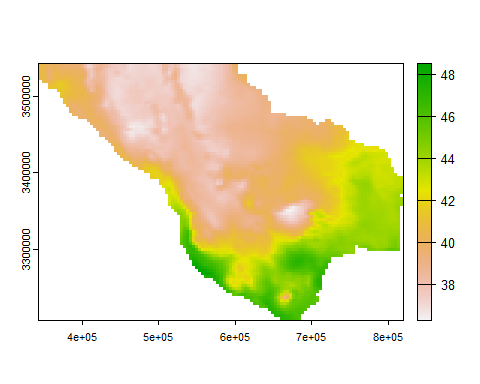
stack.temp <-  
 terra::project(stack.temp,  
 terra::crs(dat.counties))  
  
 stack.temp <-   
 terra::crop(stack.temp,  
 vect(dat.counties),  
 mask = TRUE)  
   
   
 plot(stack.temp$t20200101)



## Raster Algebra

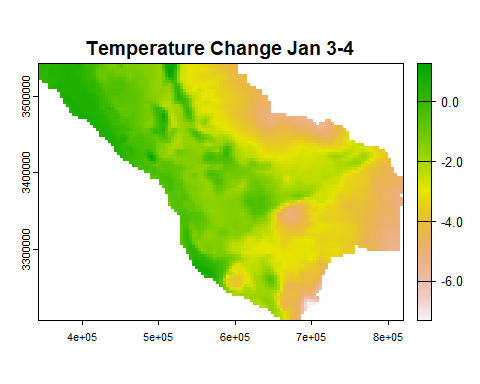
When you work on rasters, you are doing math. terra is great at this because you can use rasters as algebraic elements. For example, we can convert temperature from C to F like this:

plot(32 + stack.temp$t20200101\*1.8)



You can use a spatRaster in an equation and R will run that equation on every pixel in that raster. This is extremely useful when predicting species distributions, quantifying change, and all kinds of other fun stuff. We can even do math on multiple rasters:

plot(stack.temp$t20200104 -   
 stack.temp$t20200103,  
 main = 'Temperature Change Jan 3-4')



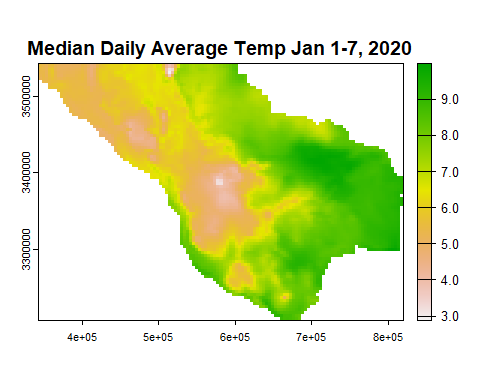
In many cases it’s helpful to pull the values from the raster to do the math. We do that with the values() method. It returns a matrix when we have multiple layers. If we count the rows that don’t have NAs, we can see how many subtraction problems we just did:

nrow(  
 values(stack.temp)[  
 !is.na(values(stack.temp)[,1]),])

## [1] 5457

Finally, there is a specific apply method for spatRasters, called app. It applies a function “vertically” to all values of a pixel across the stack. If we want the mean temperature for the first week of 2020, all we have to do is app the mean() function to the stack:

plot(  
 terra::app(stack.temp, fun = median),  
 main = 'Median Daily Average Temp Jan 1-7, 2020'  
 )



## Just for Funsies

Graphics are cool and people like maps. People really like moving maps. Here’s a cool little method that terra built in:

terra::animate(stack.temp,  
 pause = 0.5,  
 main = c('WOOO', 'BOY',"IT'S",'COLD',  
 'OUT','HEE-','-YAH!'))

Before I decided to be dumb, we used one final important method to determine how many layers a spatRaster has. that method is nlyr(), which mean “n layers”:

nlyr(stack.temp)

## [1] 7

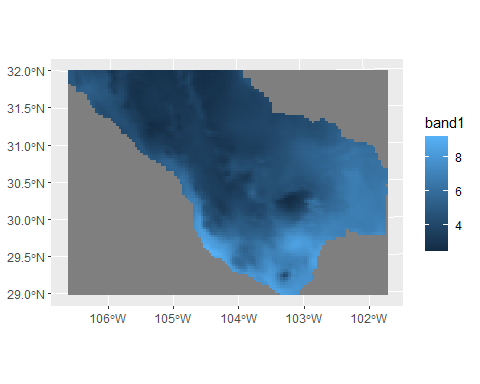
## Basic Cartography in R

Making maps in R is fun! Well, if you’re me it is. While we have been using base R syntax to make quick and dirty maps, I do not recommend that. Instead the ggspatial package extends the grammar of graphics to spatial stuff. We can use all of the things we learned yesterday with spatial things. It turns out spatial ain’t all that special…

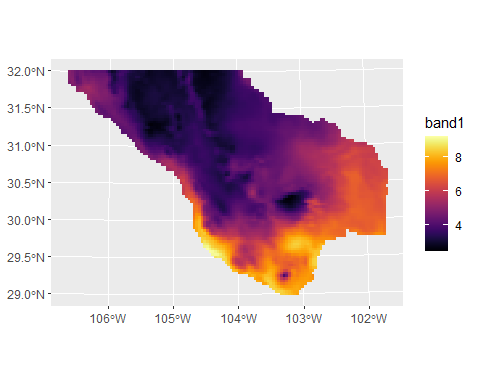
install.packages('ggspatial')

Let’s make a map of average temperature across the Trans-Pecos, with the counties and cities of at least 5k people shown. I’m feeling cute, so we might add some roads. I don’t know.

library(ggspatial)  
  
 # Plot the raster  
 ggplot() +  
 layer\_spatial(data = stack.temp$t20200101)

 The default color ramp for a raster layer in ggspatial is a lonely shade of blue. It is not exactly effective. Yesterday, we installed the viridis package, which gives us colorblind-fiendly color ramps that are easy to use. Let’s use it to swap to a better color ramp:

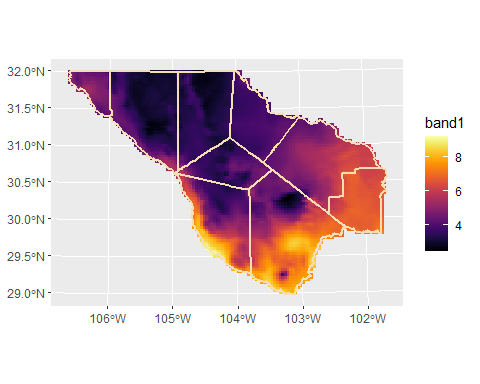
library(viridis)  
  
 # Plot the raster  
 (hotPlot <-  
 ggplot() +  
 layer\_spatial(data = stack.temp$t20200101) +  
 scale\_fill\_viridis(option = 'B',  
 na.value = NA))



Now we can go about adding the vector layers. Let’s add the counties with the geom\_sf() function:

(hotPlot <-  
 hotPlot +  
 geom\_sf(dat = dat.counties,  
 fill = NA,  
 color = 'wheat',  
 linewidth = 0.9))

## Warning: Removed 4828 rows containing missing values (`geom\_raster()`).



Remember we said we want to add the cities with a population over 5k (and less than 500k) people? How do you think we will do that?

hotPlot <-   
 hotPlot +  
 geom\_sf(data =   
 dat.cities[dat.cities$POP2010 > 5000 &  
 dat.cities$POP2010 < 500000,],  
 aes(color = POP2010)) +   
 scale\_color\_viridis(option = 'E')

We used a pair of logical conditions to find cities that were both larger than 5k people *and* less than 500k. We could also find cities that are greater than 5k *or* less than 500k with a | operator.

Every good map needs a title, north arrow, and scale bar. We already know how to do the title with ggtitle(), but now we need to use the ggpsatial commands for our map iconography:

library(ggplot2)  
  
 hotPlot +   
 annotation\_north\_arrow(pad\_y = unit(0.75, 'cm'),  
 width = unit(0.75, 'cm')) +  
 annotation\_scale(aes(unit\_category = 'imperial')) +  
 ggtitle("It looks hot, but it's cold!")

