R Data Science: Spatial Data Science 1

steppe

1/6/2022

Contents

List of Figures

List of Tables

Assigned Reading: Geocomputation with R Chapter 2.

1 Example Data

1.1 Libraries

1.2 Example Data Methods

I used the United States Geological Surveys 'Earth Explorer' to view images taken from the Sentinel 2 Satellite over the last year at a location on the border of California and Nevada near Reno. I downloaded several images which both A) covered the area of analysis well, and B) did not have much cloud cover. I used the Open Source QGIS, which has a graphical user interface (GUI), to manually geo-reference the images to locations on the earths surface. I then manually marked the edges of bodies of water, and created polygons which cover them. These data were saved as polygon vector files, in a format know as a 'shapefile' and imported them to R.

These steps can all be done programmatically, but I only wanted a few good images, so went through the process by hand.

Here we import our vector data, immediately create a 'Simple Feature' object, and add some attributes regarding the data

```
## Warning in strptime(x, format, tz = tz): unknown timezone 'US/Pacific-New'
```

```
## Warning in as.POSIXct.POSIXlt(as.POSIXlt(x, tz, ...), tz, ...): unknown timezone
## 'US/Pacific-New'
```

```
## Warning in strptime(x, format, tz = tz): unknown timezone 'US/Pacific-New'
```

```
## Warning in as.POSIXct.POSIXlt(as.POSIXlt(x, tz, ...), tz, ...): unknown timezone
## 'US/Pacific-New'
```

We import the Sentinel 2 images here. We will use them as a template to create a raster.

We will very quickly reclassify the input images to populate our new example rasters with data.

We will create empty example rasters.

2 What is a Geographic Information System?

2.1 What is a GIS

- **G**eographic **I**nformation **S**ystem is a system for producing, managing, displaying, and analyzing geographic information.
- Spatial Data Science is an emergent field which utilizes data science approaches in a **GIS**, and is a natural extension of a **GIS**

When we think about it, nearly all data has a spatial dimension, it just tends to be ignored in much of science. Historically this is sensible as performing these analysis is computationally expensive, and requires considerable expertise.

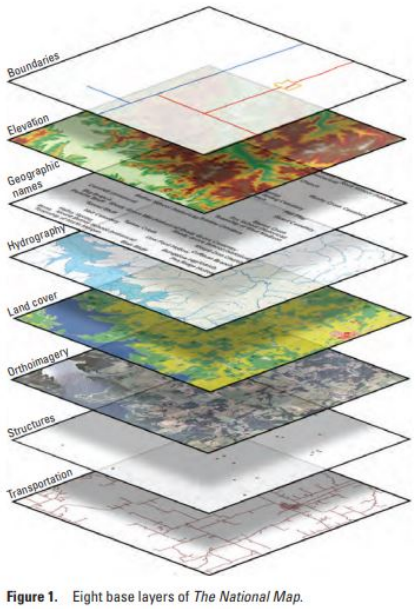


Figure 1: A Visualization of a GIS, by Anne Sexton

This is I think really one of the best and simplest way's of showcasing at heart what a GIS is. A GIS is a system wherein we can explore and study the relationships of different attributes on a process in a spatial context.

2.2 A Brief History

- Dr. John Snow suspected Cholera was spread by water
- 1854 outbreak of Cholera in the Soho neighborhood of London kills 616 persons
- Snow used both *mapping* and *statistics* to identify the water source and stop the outbreak.
- Essentially founded both Epidemiology and GIS in one stroke

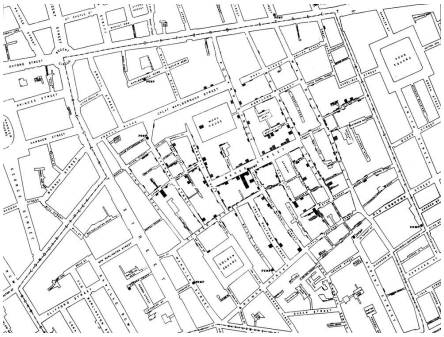


Figure 2: The Broad Street Pump, A Cholera Outbreak in London, By: John Snow and digitized by National Geographic

A good article with a quote by Tufte, anXKCD, and meaningful insight by a journalist at the Guardien is below:

https://www.theguardian.com/news/datablog/2013/mar/15/john-snow-cholera-map

- Since then GIS has continually made more use of computers and come to be it's own discipline as computers have become more available.

2.3 What is Spatial Data Science?

- The application of Geographic insights, and geospatial analyses to big data sets
- Including spatial terms in statistical models
- Bringing Geographic information to data science questions
- Developing your own spatial products and pipelines

2.4 Why Use R as a GIS and for Spatial Data Science ?

- Work Flow Automation
- Rich Ecosystem (packages, functions, code-sharing, etc.)
- Reproducible
- Self Documenting
- Computationally Efficient
- Parallel Processing/HPC interfaces

There is a great range of computer Geographic Information Systems to choose from. Many of you are likely to be familiar with a software program call 'ArcGIS' and to a lesser extent 'ArcMap' produced by the ESRI company. These products are widely used in nearly all branches of the federal and many state governments, as well as at many large companies especially environmental consultants. I think that this is a good system for you to learn about geospatial operations and workflows in, but nearly all of the work that can be accomlished in ESRI products can be accomplished in R. If you go into several environmental fields, you will need to learn to interface with ESRI products. I advise you to familiarize yourself with them.

On the other hand, I encourage you all to use R as the central piece of your geospatial analysis. If you are not developing new spatial products, you can run all of your analyses in R. If you are interested in making maps - cartography, R is now quite capable for making publication quality maps, as well as for reports and projects. It does lack slightly in cartography aspects, but you can create the data you want to style in another software, such as the open source QGIS, here and export it. But as we will see, cartography is not equivalent to GIS, and GIS is not equivalent with cartography. Do not expect to be able to make *National Georgraphic* quality maps no matter which GIS you use.

If you are very interested in geospatial work than you will see that you use a number of software programs to perform your research. In most cases you will still keep R as the centerpiece, but will likely work within a Linux environment and make good use of Unix and Bash scripts, Python, and QGIS to help fill in some parts of work-flows which R cannot address as well. You will find that R, Python, and QGIS all use many of the same software components and libraries such as GDAL/OGR, GEOS, PROJ, GRASS GIS, which have been developed by the Open Source Geospatial Foundation which was actually founded in Chicago around 20 years ago.

We will draw heavily from the Open Source Geospatial Foundation in these lectures.

3 Geodesy

- 'Geodesy is the science of accurately measuring and understanding the Earth's geometric shape, orientation in space, and gravity field.' - NOAA

This is a very highly specialized field, of which I have no formal training in when it comes to the theory of it. I do believe we have a couple specialists in Earth and Planetary Sciences, but I am far from them. Here I will cover the most basic facets of this field as they pertain to geospatial analysis among non-specialists. A more apt title for this section could be *'Geodesy taught by a dummy'*

3.1 Earth is not a perfect sphere

- Circumference at equator: 40,075 km (24,901 mi)
- Circumference along meridians: 40,009 km (24,860 mi)

Earth is not a sphere; it is ever so slightly longer than wide. The earth is actually around 43 km/27miles wider at the equator. Note the average radius of the earth is around 6371 km/3959 mi so this is quite small! However, trying to represent the Earth as a perfect sphere in geospatial models will lead to inaccurate representation of the location of objects on it.

Figure 3: Blue Marble 2012, by Suomi NPP

3.1.1 The Earth can be represented as an Ellipsoid

- A simple 3d geometric shape
- An ellipsoid is a slightly to greatly ovaliform shape
- Modeling the Earth as an ellipsoid increases the accuracy of point locations

Because the earth is slightly wider than long, the earth is technically and can be modeled practically as an ellipsoid. This model of representing the earth assumes no terms of gravity, winds, or tides. I.e. it is a perfect geometric shape with bilateral symmetry.

You can imagine that there are certain limitations to the accuracy of our model of the earth without including gravity.

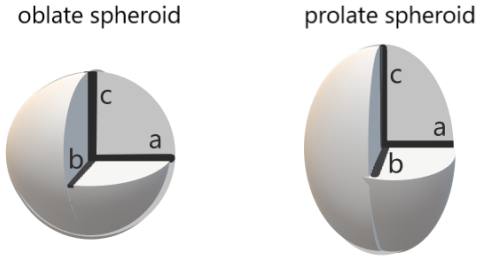


Figure 4: Ellipsoids

3.1.2 However, the Earth's surface is not smooth - Geoid

- Types of models to represent planet Earth
- Include gravity, excluding winds and tides
- Highly accurate since application of GPS technology

The Earth can also be represented as a geoid, which is a model of earth still lacking the effects of wind and tides on the earth – but which includes the major force - gravity. Because the effect of gravity is retained, the earths overall shape is both elliptical however the surface is irregular in regard to distance from the center. The surface of the Geoid is oftentimes roughly equivalent to what we could call a global mean sea level.

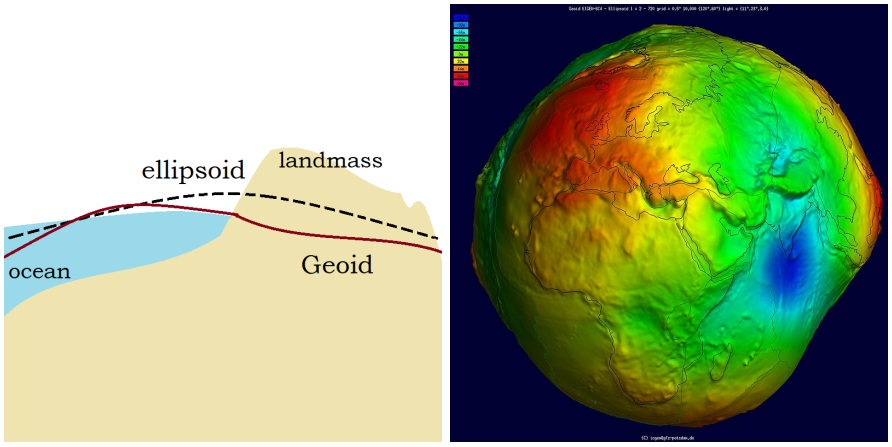


Figure 5: Left: Geoid Cross Section. Right: IGCM Geoid

Oftentimes now, very finely resolved geoids are combined with satellite measurements, and these observations are then fit with ellipsoid models, which we then use as our models of the earths surface.

By International Centre for Global Earth Models (ICGEM) - http://icgem.gfz-potsdam.de/vis3d/longtime / Ince, E. S., Barthelmes, F., Reißland, S., Elger, K., Förste, C., Flechtner, F., Schuh, H. (2019): ICGEM – 15 years of successful collection and distribution of global gravitational models, associated services and future plans. - Earth System Science Data, 11, pp. 647-674,DOI: http://doi.org/10.5194/essd-11-647-2019., CC BY 4.0, https://commons.wikimedia.org/w/index.php?curid=81462823

3.2 Geodetic Datums

- Reference frame established to represent locations within the frame.
- Historically these were locally focused and based on Geoids.
- A datum can serve either horizontal (X & Y) or vertical (Z) features.
- Components:
  - reference ellipsoid or geoid
  - origin point (from which measurements run)
  - control points very strictly measured from the origin,

Other locations are then measured from in relation to the control points.

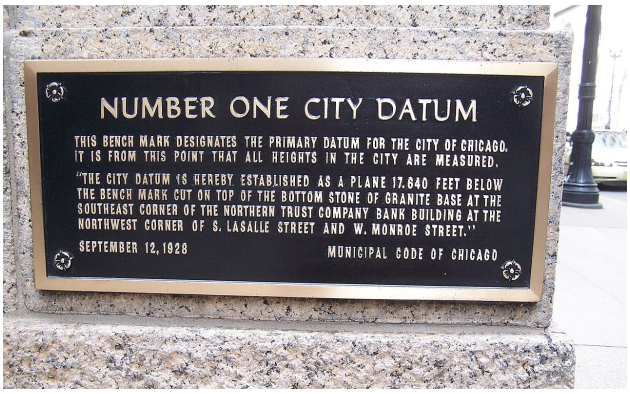


Figure 6: Number one City Datum. by: Cosmo1976

When we think of the Origins of datum points, the Equator and Prime Meridian are the most famous. However, there have been a great many datums in history.

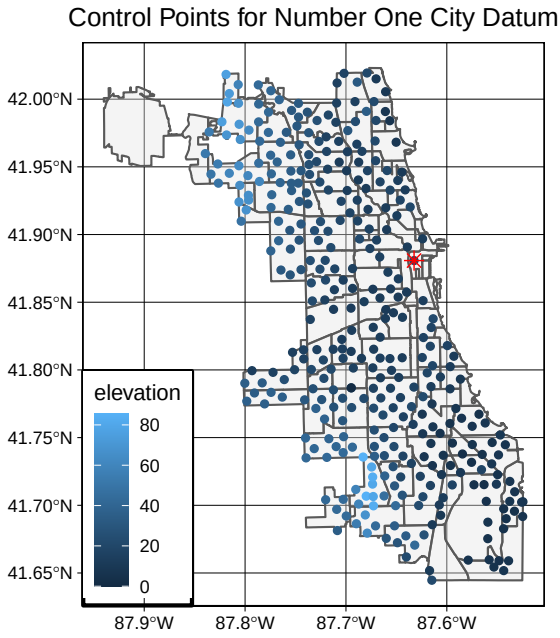


Figure 7: Control Points Number one City Datum

In fact the city of Chicago established it's own elevation datum in the late 1920's. If we look at this quick map, we can see where the Origin point (also pictured) and control points are located.

Also do you really think Chicago is only 80 feet at most above sea level? No of course not! The lowest elvations in the city are roughly 580 feet above sea level, but in the 1920's, it was much easier to define elevation from an adjacent location. So all the values in the map should be X + ~580 ! This is exactly why so many datums are out there.

3.2.1 Geodetic Coordinatees

Since our area is 3-dimensional, we use a system related to Cartesian coordinates, however given the curve of the earths surface curvilinear rather than linear coordinates are used.

"The geodetic (or geographic) latitude is the angle between the equatorial plane and the normal (vertical) to the ellipsoid surface at the considered point." - Proj

- phi = geodetic latitude (north/south)
- lambda =longitude (east/west)
- h = ellipsoidal height
- N = Normal (A plane at a right angle to the surface of the ellipsoid)

In the sphere image we have a latitude of 40*, but we do not know to which hemisphere it is associated with.

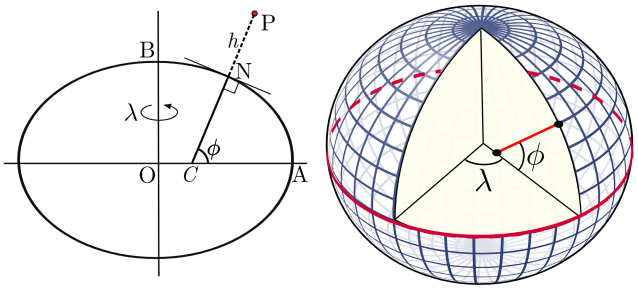


Figure 8: Angles on Ellipsoid and Geodetic Coordinates. by: Peter Mercator

By Peter Mercator - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=17717979

3.2.2 Coordinate Notation

- Trigonometric
  - Degrees Minutes Seconds (DMS), e.g. 42°03'27.7" N
    - Sexagesimal/base 60 (think: minutes in an hour)
  - Decimal Degrees (DD), e.g. 42.05759 N
    - Decimal Fractions of a Degree (portion of 360/2)
- DMS; seldom used in digital formats
- DD; almost exclusively used

Recorded in many forms DMS, DD, (trigonometry) UTM.

As a circle has 360 degrees, so does the earth. Given the nature of datums we split the world into a positive and negative sign for both latitude and longitude; we refer to the 0 lines as '0 meridians'. We have 180 positive and 180 negative degrees, and as these are of a course resolution we retain the decimals of degrees to better locate objects. As you all know, the Western Hemisphere is the negative component of a 180 degree half.

$$\text{Decimal Degrees} = \text{Degrees} + \frac{\text{Minutes}}{60} + \frac{\text{Seconds}}{3600}$$

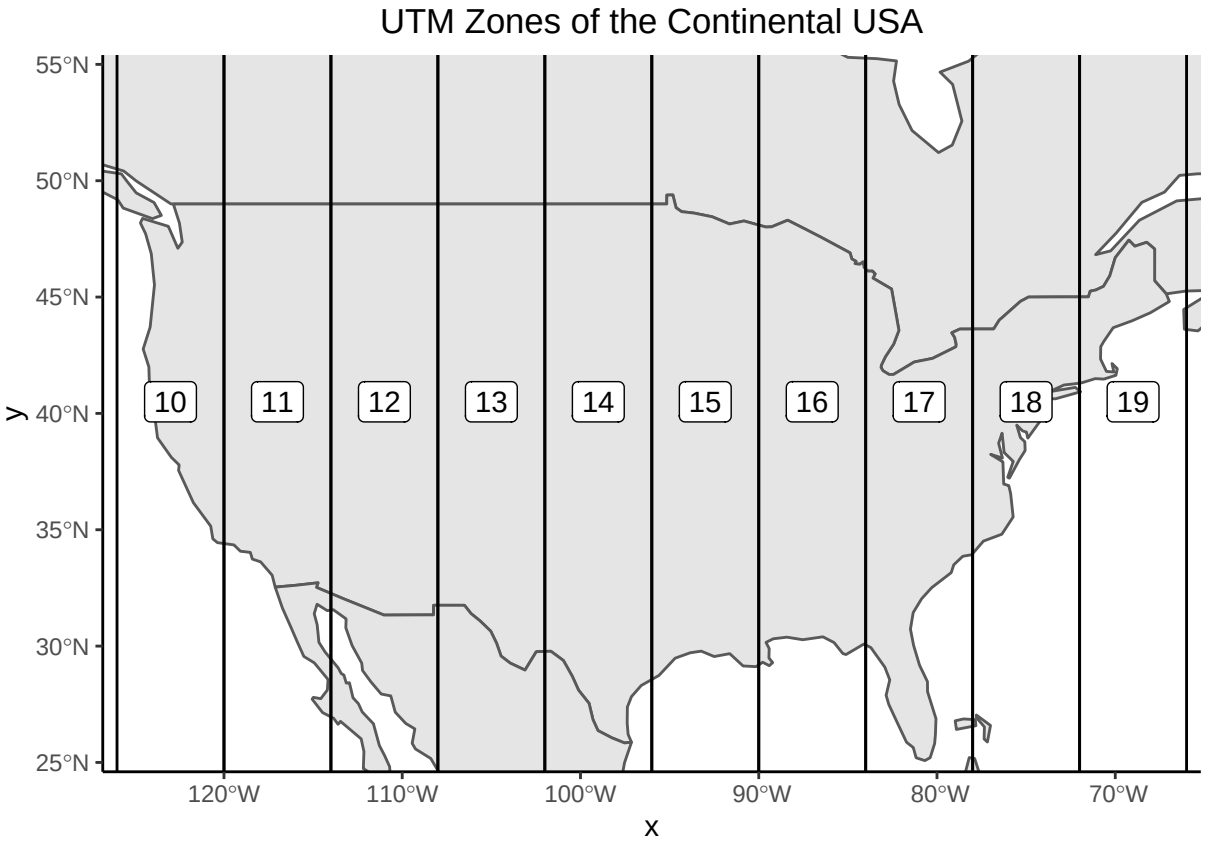
$$\text{Latitude of Tech} = 42\check{r}03'27.7"N$$

$$42 + (\frac{03'}{60}) + (\frac{27.7"}{3600})$$

$$42 + 0.05 + 0.007694 = 42.05759 \text{ Decimal Degrees}$$

- Universal Transverse Mercator (UTM)
  - Divides the world into 60 zones
  - Flattens each zone
  - Measures distances in Meters

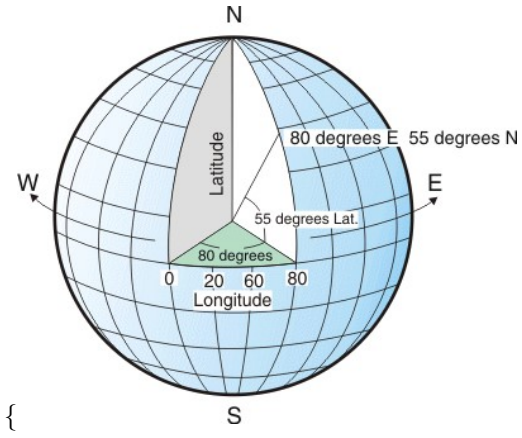
UTM - Common for field work, planar measurements of meters.



3.3 Coordinate Reference System

The comprehensive system of specifying locations on the Earth's surface. Each Coordinate Reference System is composed of an Ellipsoid (or Geoid), a geodetic datum, and for cartographic purposes a projection.

\begin{figure}



{

}

\caption{Geographic_Coordinate_Reference_System. by: Anna Krystalli} \end{figure}

3.4 Problems with working on flat surfaces

3D to 2d does not work well. But we have worked around this. **sorta**

Figure 9: Orange Peel. by: Nathan Belz

3.4.1 Geographic Coordinate System

Where the location is on located earth. We are able to pinpoint locations, quite easily using one of the Global Navigation Satellite Systems, for example the 'Global Positioning System' or GPS. We have a coordinate system for this known as the World Geographic Datum. As we are merely recording the location of an object on a 3d object this is now essentially a trivial process.

3.4.2 Projected Coordinate System

Most of us work on flat desks and flat computer screens. We need to represent locations in space on these surfaces. We project coordinates from their geographic locations on earth to locations on flat representations of earth.

3.4.3 Major Map Projections

Many different ways to create two dimensional maps – thousands of projections. None are perfect. There are four main types of projections, each with their own strengths and examples. One of the most common examples of each of these is included in the table below.

- Equal Area: Lambert Cylindrical equal-area
  - Pro: No distortion of area near equator (here)
  - Con: Distorts area at the Polar regions
- Equal Distance: Equi-rectangular (plate carrée projection)
  - Pro: Looks good in mapping applications
  - Con: Distorts both shape and directions
- Conformal:
  - Pro: Boundaries are accurate
  - Con: Distorts Polar area
- Compromise:
  - Pro: Sensitive to both area and direction
  - Con: Sensitive to both area and direction

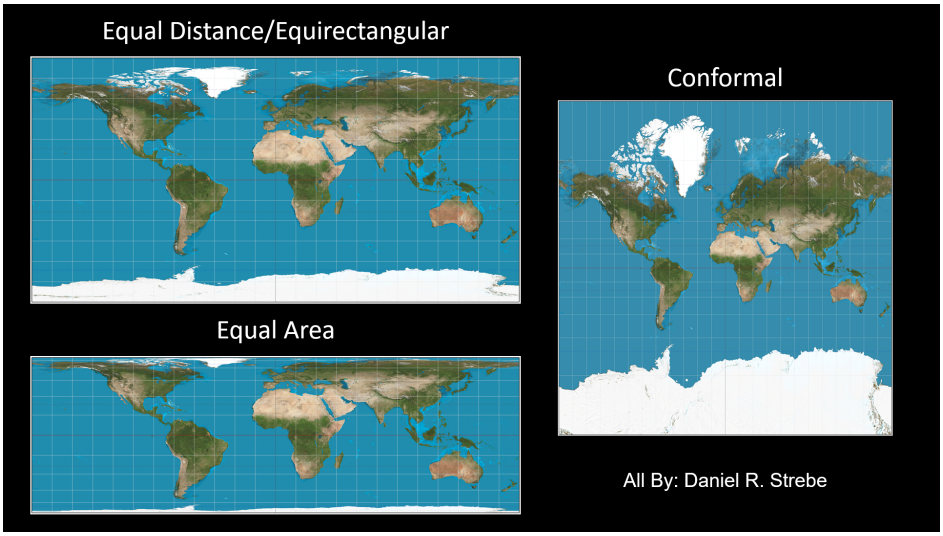


Figure 10: Map Projections. by: Daniel Strebe

3.5 Geodesy Takeaways

Unless you really focus on GIS, hardly anything I have said this lecture will matter to you.

- There are various models (geoids & ellipsoids) to represent the shape of the earth
- Different datums are used to represent different parts of the earth. This is in part due to legacy effects.
- You will almost always use WGS 84 (based on a ellipsoid – which is fit through a special model of earths gravitational fields a geoid) NAD83 (based on a ellipsoid) for a geographic coordinate system. These +/-1 m from each other across much of North America. More useful than a geoid.
- You will usually want to use a UTM grid based on WGS for or State Planes based on NAD83 for projections.
- Different coordinates notation systems are used, focus on Decimal degrees.

4 An Introduction to Geographic Data Models

- **Vector Data Model** represents discrete features on the planet using geometries such as: points, lines, and polygons.
- **Raster Data Model** represents (usually) continuous features on the plant using continuous surfaces, like a tile.

Vector data tends to only include features of interest, e.g. bodies of water; whereas a Raster will include, **explicitly**, the absence of features (e.g. both water and terrestrial areas).

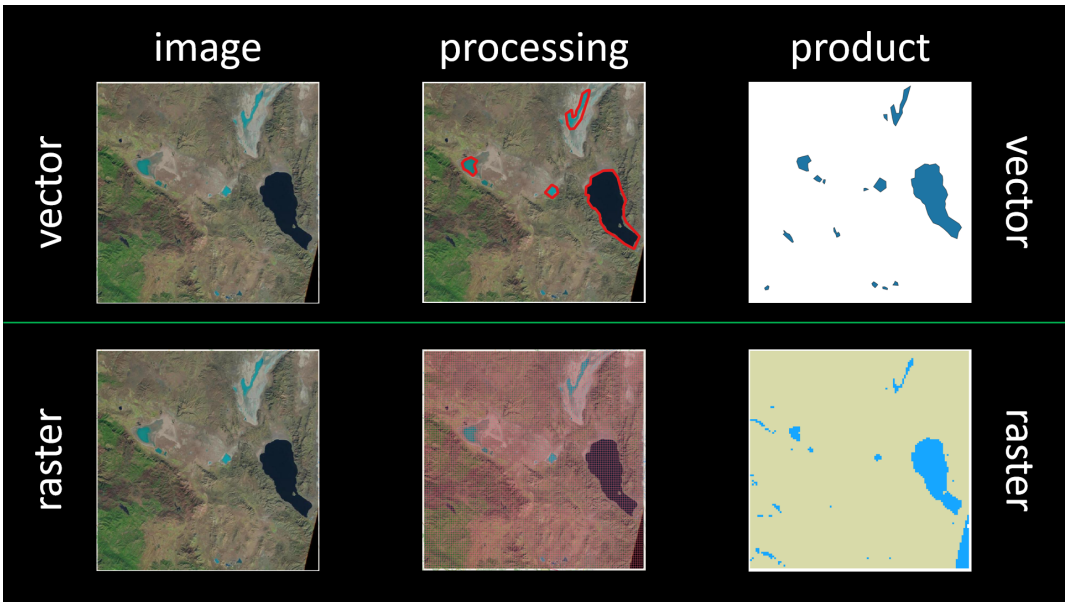


Figure 11: Vector and Raster Data Models

Talking:

In our field, we generally take values from rasters to serve as predictors to our sample unit, many of us store our sample as spatial points. We also tend to predict our models back onto raster surfaces.

5 R Data Types Reviewed

Remember that different data types take up different amounts of memory in pretty much all software systems.

```
When the values in raster cells are stored as integers as opposed to floats (with decimal points) they
```

```
When the values in raster cells are stored as integers as opposed to words they take up only 0.5 as many
```

Please note that our Raster in this scenario is *very* small, small enough we can wrap our brains around it with only minimal inspection.

Without knowing much about a raster (yet!) We can simulate some of these comparisons to give a clue as to why it contains values like they do.

Rasters generally come in 8-bit signed (unsigned are not uncommon) integers. Pictures are huge amounts of data, these values help reduce them.

```
The size of the our empty raster is 0.01416 MB
```

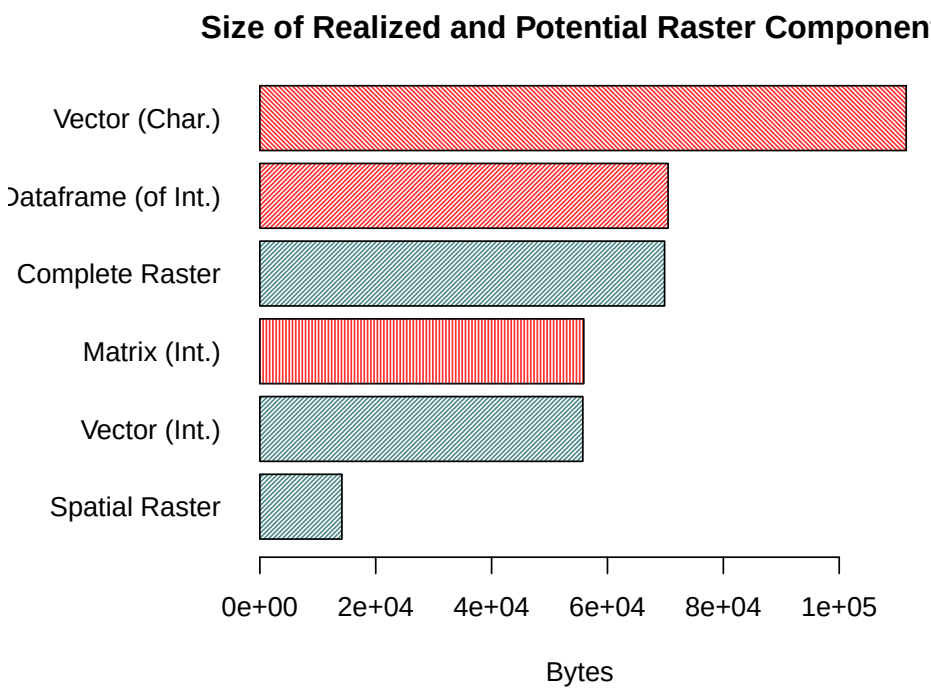
```
The size of the data for our raster as a character vector is 0.11156 MB
```

```
The size of the data for our raster as a numeric (integer !) vector is 0.055744 MB
```

```
The size of the data for our raster in matrix form is 0.055912 MB
```

```
The size of the data for our raster in dataframe form is 0.055912 MB
```

```
The size of our complete raster is 0.069856 MB
```



6 Vector Data in R

- **Vector Data Model** represents discrete features on the planet using geometries such as: points, lines, and polygons.

6.1 Simple Features - Standards

- Open Geospatial Consortium ISO 19125-1:2004: Currently adhered to in ESRI, used in GDAL.
- Features have *geometries* describing their locality on Earth, and properties described by *attributes.*
- If the geometry of a feature is a polygon, it is composed of points, connected by straight lines.
- Lines composing polygons cannot intersect

6.1.1 Simple Features 1 - Attributes

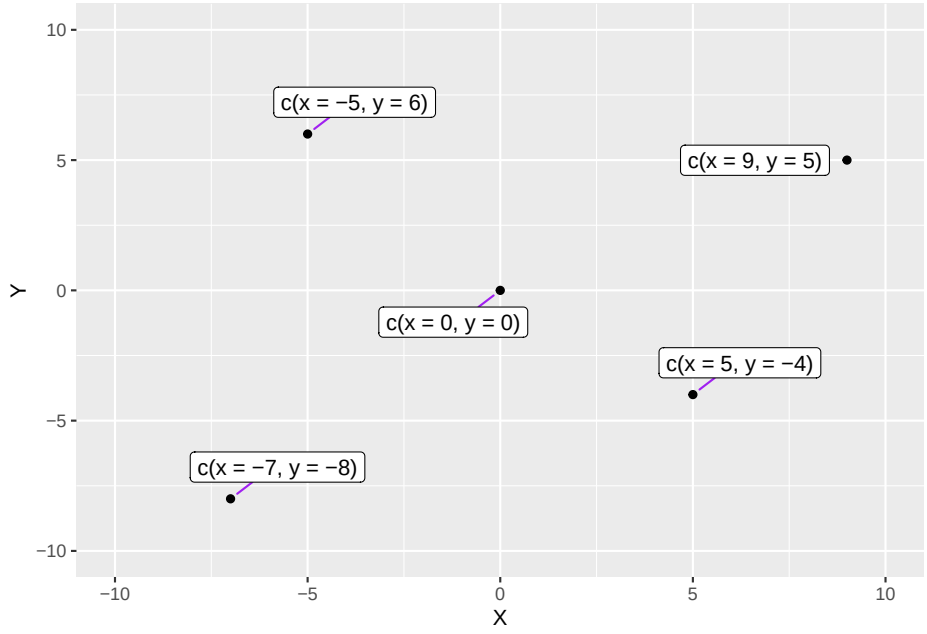
- Attributes of the feature, theoretically devoid of spatial context.
- Described in text, numbers, stored in a data frame type object.

TAXON	DBH	HEIGHT
Robinia pseudoacacia	40	24
Quercus alba	32	21

6.1.2 Simple Features 2 - Coordinates form a Point

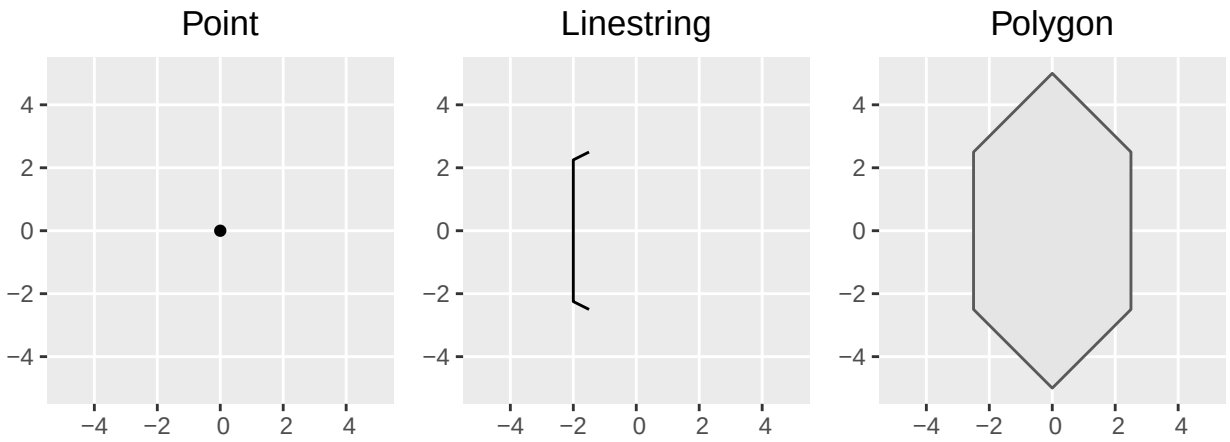
- all geometries are composed of points.
- points only require two coordinates, X & Y.
- Y = Latitude (necessary), X = Longitude (necessary)
- Z = Elevation (somewhat uncommon)

- M = Time or Uncertainty of Measurement (rather uncommon)



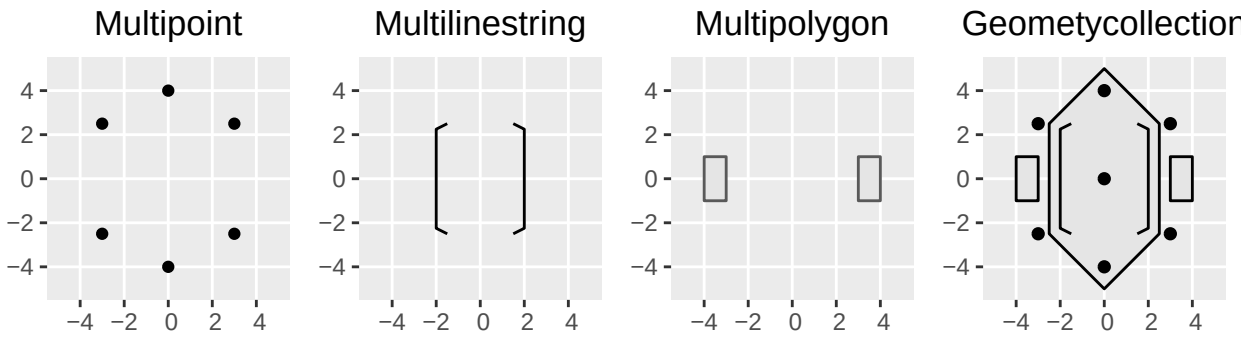
6.1.3 Simple Features 3 - Points are/form a SF Geometry (sfg's)

- SFG is the spatial topology associated with a feature
- POINT - A true dimensionless 1 dimensional point
- LINESTRING - Sequence of points connected by strings.
- POLYGON - Sequence of points connected by strings, which close upon themselves. – i.e. the first and last point of the polygon are the same.



6.1.4 Simple Features 4 - A Geometry can be combined to form Geometries (sfg's)

- Two or more Points, and sets of Linestring, and polygons can be the geometries of a single feature
- Multi(POINT), multi(LINESTRING), multi(POLYGON) - Forming collections of geometries
- GEOMETRYCOLLECTION - A mixed set of these three or more geometries in the same geometry



6.1.5 Simple Features 5 - Geometries and Spatial Information form an SF Collection (sfc)

- A list column of S3 type containing certain parameters regarding the Geometry/Geometries
- Coordinate Reference System ('crs')
- Precision ('precision')
- Bounding box ('bbox')
- Number Empty ('n_empty')

Wherein precision refers to the precision of the coordinates forming the geometries. the bbox is a rectangle which includes all pairs of coordinates in the geometry. N_empty, denotes whether the geometry is missing coordinates in a position.

6.1.6 Simple Features 6 - Recapped.

TAXON	DBH	HEIGHT	LONG	LAT	geometry
Robinia pseudoacacia	40	24	-87.67607	42.05663	POINT (-87.67607 42.05663)
Quercus alba	32	21	-87.67399	42.05715	POINT (-87.67399 42.05715)

```
Geometry set for 1 feature
Geometry type: POINT
Dimension:     XY
Bounding box:  xmin: -87.67607 ymin: 42.05663 xmax: -87.67607 ymax: 42.05663
Geodetic CRS:  WGS 84
```

```
Precision:     2
```

```
sfc_POINT of length 1; first list element:  'XY' num [1:2] -87.7 42.1
```

- We now finally have *both* attributes and coordinates which are forming a point geometry.
- This is a Simple Feature.

6.2 sp

- Predecessor to the 'SF' R package
- Many spatial statistics programs still only take sp objects as input
- A good introduction to S4 objects; popular in new spatial packages too!

6.2.1 sp - History

- Released in 2005
- First package to hold all major vector types of geometries
- Unified Spatial class allowed for support in many spatial statistics packages
- Improved mapping of spatial objects
- Formed the centerpiece of spatial statistics in R for over a dozen years.

6.2.2 sp - Spatial Classes for Topology

- SpatialPoints
- SpatialLines
- SpatialPolygons
- SpatialGrids

Similar in theory to the simple feature geometries we learned about...

6.2.3 sp - Structures of The Spatial* Class - Topology Only

- Bounding box
- Coordinate Reference System
- ... Coordinates!

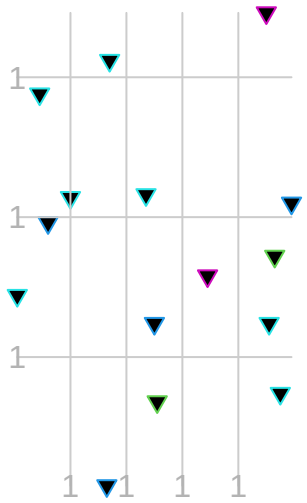
6.2.4 sp - Attributes - DataFrame

- Attachment of attributes, in a data frame, to a Spatial topology can create:
- SpatialPointsDataFrame
- SpatialLinesDataFrame
- SpatialPolgonsDataFrame
- SpatialGridsDataFrame
- Each S*DF is accessed the same way e.g. SPDF@data[['col_name']]

We see here that SF contains all of the spatial information in a Simple Feature Collection which can be tidily tucked away in the geometry list column, in SP the analogue to the SFC is the object itself.

SP is mostly geometry with a data slot, SF is mostly data frame with a geometry column.

6.2.5 sp Overview of a Spatial*DataFrame



- Data frame is held in a different slot from the geometry and topology
- Data frame columns may be subset
- Data frame columns accessed via object@data[['colname']] indexing

6.2.6 sp - Recapped.

- If you need to use spatial statistics, you will come across these objects.
- Don't sweat them **too** much; you can always convert from and to sf to run certain analyses

7 Raster data in R

- Reminder: the Raster data Model is a way to represent (continuous) features on the planet using grids
- Excellent format for storing and sharing data
- Divides space into continuous grids
- Raster data sets are often distributed in *tiles*

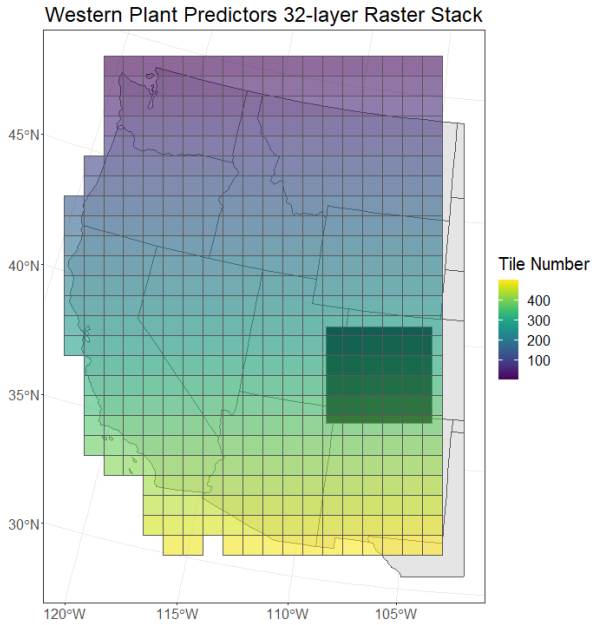


Figure 12: Western Plant Predictors. Note each cell is the extent of a single raster tile, each tile contains cells.

7.1 Raster Components

- Bounding Coordinate(s)
- Cell Resolution (Size of cell)
- Dimensions (No of cells in rows and columns)
- Coordinate Reference System (CRS)

Note that both *cell sizes*, and the *resolution of the values* in cells can, within reason, be converted to finer and coarser resolution. We will discuss these types of calculations next class.

7.1.1 Example Raster 1 Create Frame

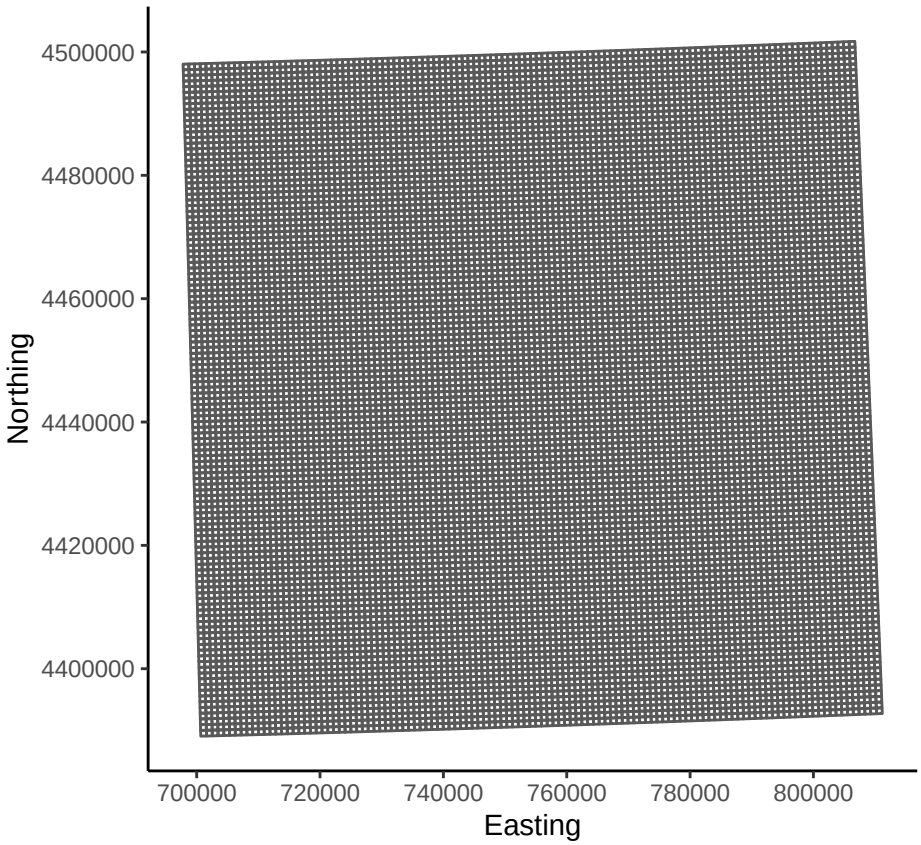
Rasters tend to confuse people. We are going to create our own example here before we talk about them much more.

Fairy shrimp live in bodies of water which dry out in late Spring, and refill in early Fall. We seek to determine where suitable habitat for fairy shrimp are in the Great Basin. In order to do so, we will use satellite imagery to detect valleys which fill with water in Fall, and dry out in late Spring.

Figure 13: Suprise Valley by: John Glen

```
empty_raster <- raster(

  # rasters have 4 bounding edges
  # Here we define each 'corner' of the raster'
    xmn = 697129.7,
    xmx = 811775.7,
    ymn = 4388466,
    ymx = 4502382,

  # Here we set the number of cells 118*118
    nrows = 118,
    ncols = 118,

    # we do NOT manually specify the cell size here.

    crs = "+proj=utm +zone=10 +datum=WGS84",
    # set the rasters Coordinate Reference System
)
```

In the code above, we are going to define all four of the essential components of a raster. Clearly, we are defining three explicitly (Bounding Coordinates, Dimensions, CRS), and the remaining one implicitly (Cell resolution).



We can imagine that the raster we are currently creating looks like this. A frame without content. We can see where the bounding coordinates are,

7.1.2 Example Raster 2 Set Values

```
raster_matrix <- matrix(rast_vals_num, # # fill matrix with values,
                        nrow = empty_raster@nrows, # create matrix same dimensions.
                        ncol = empty_raster@ncols, # create matrix same dimensions
#ensure filling of matrix goes from upper left to lower right.
                        byrow = T)

raster_matrix[90:118,112:118] <- 1 # fix the clipping image at edge.
```

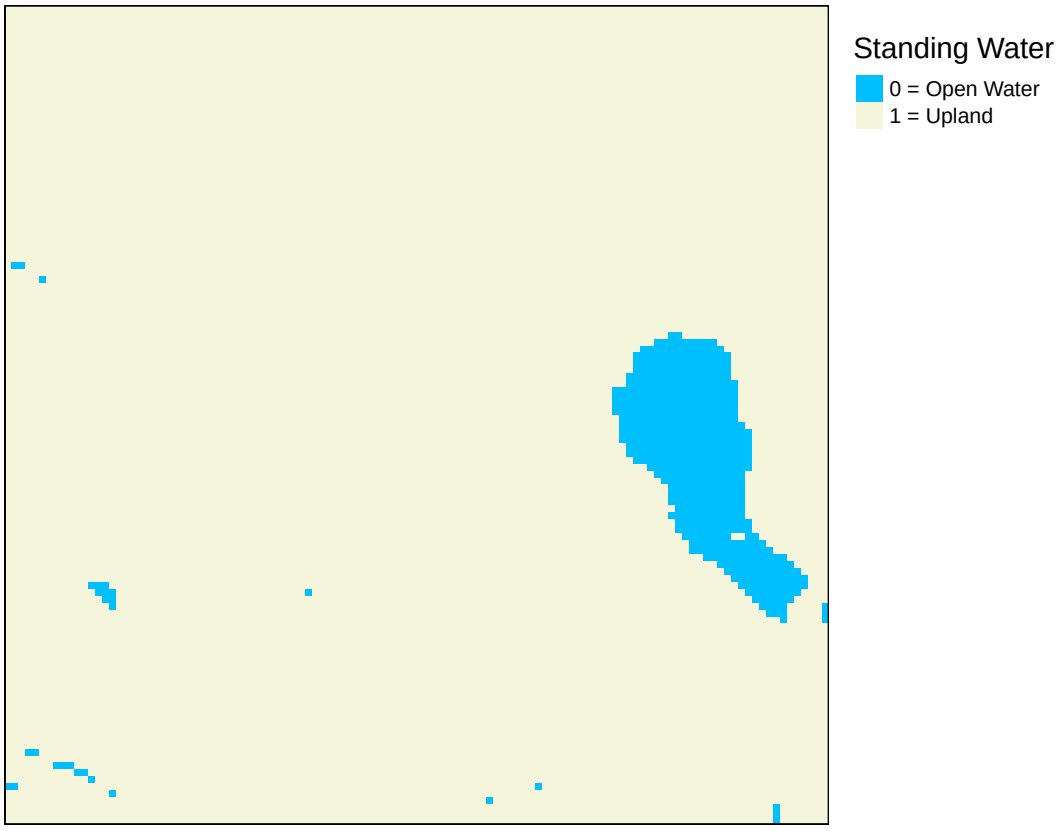
Table 3: Example Matrix showing values underlaying a raster layer.

0	1	1	0	1	0	0	1
1	0	1	0	0	1	1	0
1	0	0	1	1	0	1	0
0	1	1	0	1	0	0	1
1	0	1	0	0	1	1	0
1	0	0	1	1	0	1	0

I think it is easiest to imagine that the values within a raster are in the shape of a matrix. Hence, using the code above we could take a vector of values, and bend them, so that each position within the vector matches up with the beginning of a new row.

```
example_raster_dec <- setValues(empty_raster, raster_matrix)
```

7.1.3 Example Raster 3




```
The width of each raster cell is: 971.57627 meters
The height of each raster cell is: 965.38983 meters

This example_raster_dec contains: 13924 elements
```

7.2 Raster Package - in R!

- Does not need to load all files into active memory at once
- Many functions based on functions in 'base' R.

7.2.1 Attributes

- Cells
- Values

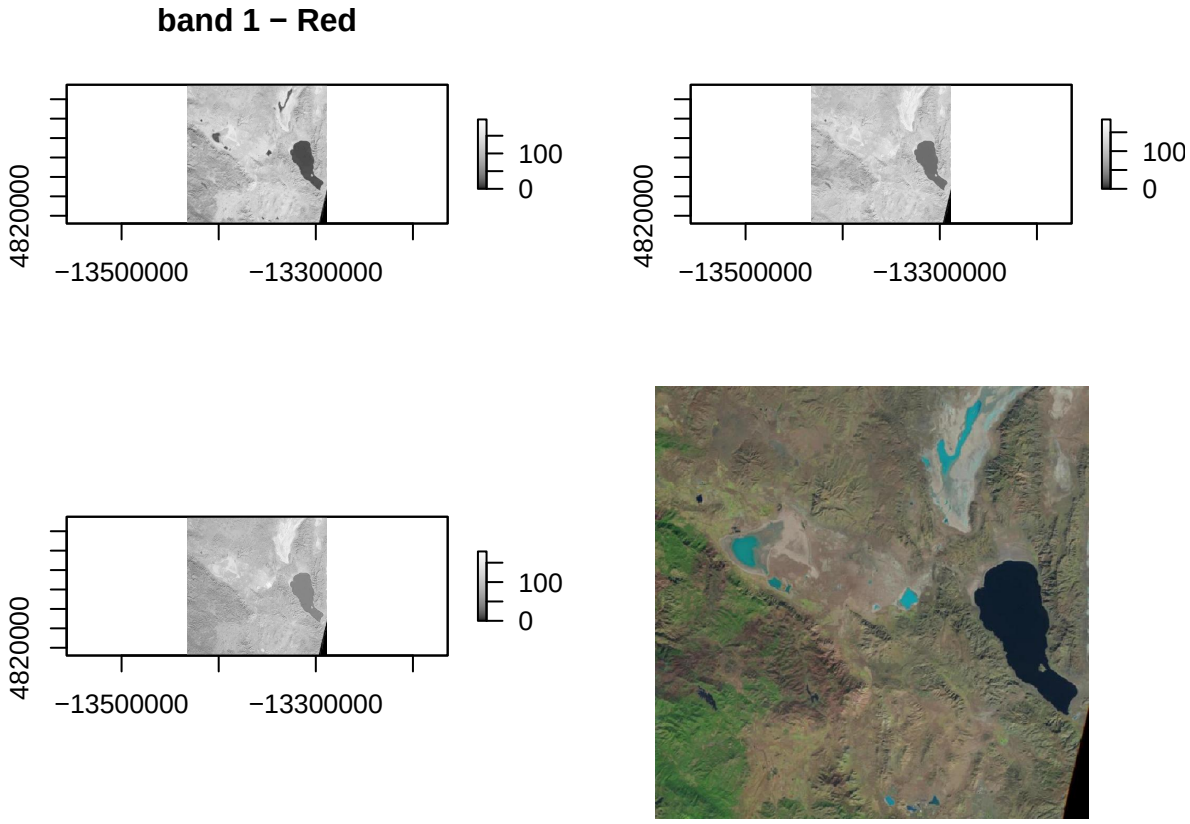
7.2.2 Raster Layer

A single raster .. do something cool here doofus.

7.2.3 Raster Brick

As I have mentioned rasters are often generated from satellite imagery; we will discuss rasters which are not developed this way next lecture. Historically most pictures are imaged via the use of three bands. These

having spectral values of Red, Green, and Blue (RGB) associated with them. For example our .tif file, is composed of three bands. Note that a Rasterbrick is most often used for loading in these types of imaging data, which can then be processed to form a more typical 'raster' dataset.



Rasters are often built from satellite imagery. Historically most pictures are split into three copies, each one having spectral values of Red, Green, and Blue (RGB) associated with them. For example our .tif file, is composed of three layers. Note that a Rasterbrick is most often used for loading in these types of data.

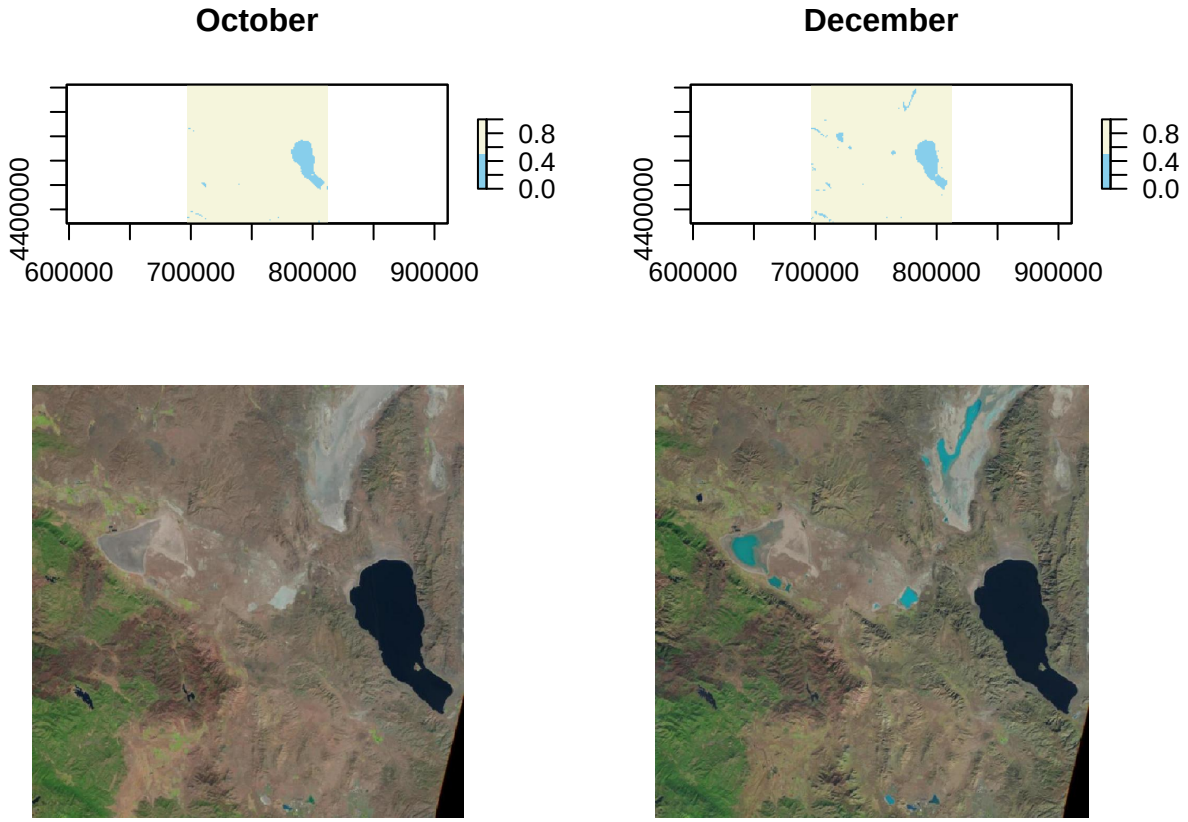
Bricks are very important for processing and classifying raw image data. While we have three bands here, nowadays LIDAR equipped with Hyperspectral sensors are likely to have many more bands up to a couple hundred depending on the application.

One important technical point to note with the Rasterbrick is that each band of the brick loads from the same individual file, for example picture. And that bands are not typically combined from different picture sources.

Raster bands also do not need to be held in memor, allowing one to work through large amounts of them.

7.2.4 Raster Stack

In general these are rasters which have been classified and we want to extract values from or run calculations with. Now what is great about layers, is that we can do one big thing bricks cannot do, we can load in many layers from many files and create a stack of attributes we are interested in studying on the fly.



The main use of Raster Stacks is to hold layers of similar themes.

Two common examples: 1) Each raster layer in the stack is a variable from a different month, e.g. mean monthly temperature from January -> December (12 layers per stack) 2) Each raster layer in the stack is a variable of interest in an analysis, e.g. yearly mean temperature, mean precipitation, etc. from which we want to extract values.

Does not need be held in memory

7.3 Terra

- Developed by the same team as the Raster Package
- Functionally virtually identical to Raster, but calculations run more quickly ! :-)
- Rasterbricks/layers no longer need specification, all objects are Spatstats ?
- Will supercede Raster, but see points 1 & 2.

8 Obect Orientated Programming in R

"Object-oriented programming (OOP) is a programming paradigm based on the concept of"objects", which can contain data and code: data in the form of fields (often known as attributes or properties), and code, in the form of procedures (often known as methods)." - from Wikipedia

To date we have largely focused on functional programming. That changes now.

R uses upwards of 25 types of objects. Many of these are highly specialized and we will not go into them, however it is necessary to elaborate on some of the differences between them. To date we have largely been using what may be called 'base' objects, i.e. objects which lack a 'class'. A class contains attributes which define certain parameters of the object, note that each attribute is based off of a 'base' object at some point.

8.1 10.1 Base Objects Reviewed

Some of the most common base objects are vectors, matrices, and lists.

```
rast_vals_char[1:3]
```

```
[1] "Terrestrial" "Terrestrial" "Terrestrial"
```

```
The first element of this vector is: Terrestrial
```

```
A vector is an object of type: base
```

```
This vector objects inhereits of class: character
```

if we use the function 'otype' from the sloop package we can see that this object is a 'base' object and is in a class defined as character

```
raster_matrix <- matrix(c(1,0,1,
                          0,1,1,
                          1,0,0),
                        ncol = 3)
```

1	0	1
0	1	0
1	1	0

```
A matrix is an object of type: base
```

```
A matrix inherits classes of: matrix
A matrix inherits classes of: double
A matrix inherits classes of: numeric
```

```
list
```

```
I made a list out of the last 2 objects and a dataframe:
```

```
$Vector
[1] "Terrestrial" "Terrestrial" "Terrestrial"

$Matrix
     [,1] [,2] [,3]
[1,]    1    0    1
[2,]    0    1    0
[3,]    1    1    0

$Dataframe
  1 2 3
1 1 1 1
2 1 1 1
3 1 1 1
```

```
A list is an object of type: base
```

These base objects are more of less capable of just *storing* values. I.e. In a sense we can put whatever we want into a vector and it will work out - if we have mixed data types theywill be coerced to characters, *but* we can do it.

```
junk <- c(1, 'A', 'alpahbet soup', '$')
print(junk)
```

```
[1] "1"             "A"             "alpahbet soup" "$"
```

8.2 'Introduction' to S3 Objects

```
Warning in as.POSIXlt.POSIXct(x, tz): unknown timezone 'US/Pacific-New'
```

```
This is an example of a POSIXct time: 2021-12-05, to me it just looks like a character vector
```

Let's take a look at our vector of POSIXct time zones, which superficially to me (an human) look like numbers separated by dashes.

```
$tzone
[1] "US/Pacific-New"

$class
[1] "POSIXct" "POSIXt"
```

We see that two attributes are stored in that object. The first is the timezone, which is defined elsewhere in R (and I believe is inherited by R from other systems), but is notated within this object. Critically, We also have the formal 'class' definition of a POSIXct format.

```
Warning in as.POSIXlt.POSIXct(x, tz): unknown timezone 'US/Pacific-New'
```

```
The amount of seconds since 1970 and 2021-12-05 is: 1638662400
```

POSIXct dates are all actually treated in R as the amount of seconds between an arbitrary date e.g. January 1st 1970, and the date of observation; so R is really keeping dates in a numeric format - but is hiding this from us! And converting the values into a more human friendly format more or less for display purposes only.

This is a new level of functionality that we have not directly considered yet this quarter. As mentioned we have used objects much like book pages. Now we see that certain objects, are able to refer to themselves to perform, for example, calculations and conversions.

```
A POSIXct is an object of type: S3
```

Objects with the capability to store values in fields, and perform procedures upon themselves form the basis of Object Orientated Programming. In R we have two main OOP classes, S3 and S4. In general objects of both of these classes are *'large'*, however this is not always the case.

Another S3 object which superficially looks like a vector is an 'Units' object.

When we define a units object, we can simply supply a number for value, and a unit of measurement.

```
fifty_meters <- units::set_units(50, meter)
print(fifty_meters)
```

```
50 [m]
```

These objects are capable of performing procedures on themselves, such as converting between meters and yards.

```
units(fifty_meters) <- units::make_units(yards)
print(fifty_meters)
```

```
54.68066 [yards]
```

```
A Units is an object of type: S3
```

```
It is its own units class: units
```

```
$units
$numerator
[1] "yards"

$denominator
character(0)
```

```
attr(,"class")
[1] "symbolic_units"

$class
[1] "units"
```

A more shocking S3 object is the data frame

```
S3

$names
 [1] "c" "d" "a" "b" "g" "j" "i" "h" "k" "f"

$row.names
 [1] "T" "Q" "Y" "S" "R" "W" "X" "U" "V" "N"

$class
[1] "data.frame"
```

We see that of the non-base objects, the most common object is S3. Actually, surprise surprise the amazing data frame has been hiding out as an S3 object this whole time! You may have actually seen an error message indicating this at some point... So if we look at the attributes of a data frame we see that we have both row and columns names, and we also have a formal definition of the class being a data frame, which in part consists of code demanding that our data frame be rectangular in nature.

Finally we turn our attention to one slightly more complex S3 object.

8.2.1 Simple Features

The incredibly popular Simple Features 'SF' package actually stores all it's features in S3 object. Given that a data frame is an S3 object, this is a necessity - but there are some big jumps under the hood. I bet, even more so than these objects being 'accessible' via tidyverse syntax the simplicity of the S3 object compared to S4 sp objects is what spurred there popularity

```
A simple feature is an object of type: S3
```

What is really neat about tibbles, and I am not sure if you all recall is that while they just look like a data frame they are capable of holding a list in a column, hence a 'list column'.

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5.0	3.6	1.4	0.2	setosa
5.4	3.9	1.7	0.4	setosa

For example we can take the four columns of the Iris dataset which contain the measurement variables, and reduce them to two thematic list columns using the 'nest' function.

```
iris <- iris %>%
  nest(petal = starts_with("Petal"),
       sepal = starts_with("Sepal")
       )

iris
```

```
# A tibble: 3 x 3
  Species    petal             sepal
  <fct>      <list>            <list>
1 setosa     <tibble [50 x 2]> <tibble [50 x 2]>
2 versicolor <tibble [50 x 2]> <tibble [50 x 2]>
3 virginica  <tibble [50 x 2]> <tibble [50 x 2]>
```

```
# Check the first row of the second column
head(iris[[2]][[1]])
```

```
# A tibble: 6 x 2
  Petal.Length Petal.Width
         <dbl>       <dbl>
1          1.4         0.2
2          1.4         0.2
3          1.3         0.2
4          1.5         0.2
5          1.4         0.2
6          1.7         0.4
```

```
# see the structure of the first row of the petal column
str(iris$petal[[1]])
```

```
tibble [50 x 2] (S3: tbl_df/tbl/data.frame)
 $ Petal.Length: num [1:50] 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
 $ Petal.Width : num [1:50] 0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
```

If we take a close look at the first for of the second column, that for *I. setosa*, we see all of the short and narrow petal measurements we are familiar with for this species.

In this example we see that each element in 'petal' contains a list of its own, each of which has a data frame with a column storing *both* the Petal Length and Petal Width values. Each row of each list column (petal & sepal) we see in this tibble is like this.

So what the developers of SF did is to create an S3 object which can hold all of the spatial information in a list column - without you really realizing this information is there.

```
[1] "sf"         "tbl_df"     "tbl"        "data.frame"
```

```
Warning in as.POSIXlt.POSIXct(x, tz): unknown timezone 'US/Pacific-New'
```

Water	geometry	Date
Pyramid_Lake	POLYGON ((-119.599 40.21368...	2021-12-05
Davis_Lake	POLYGON ((-120.5374 39.9152...	2021-12-05
Frenchman_Lake	POLYGON ((-120.2149 39.936,...	2021-12-05
Silver_Lake	POLYGON ((-119.9159 39.6573...	2021-12-05
Swan_lake	POLYGON ((-119.8356 39.6816...	2021-12-05
Gold_Lake	POLYGON ((-120.6635 39.6694...	2021-12-05

```
$n_empty
[1] 0

$crs
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]]

$class
[1] "sfc_POLYGON" "sfc"

$precision
[1] 50

$bbox
      xmin       ymin       xmax       ymax
-120.66350   39.64317 -119.40150   40.57838
```

This is output is understandably a little much, but these are *all* of the attributes of a Simple Feature Collection which is hidden in the geometry column of a simple feature. Clever right?

8.3 Introduction to S4 Objects

Now S3 objects, are somewhat lax. However there is an S4 object which is a little bit more complex.

Examples of S4 objects include the lovely Raster, and Spatial*(e.g. Points, Polygons, Dataframes...) objects. I find that lists, and S4 objects terrify virtually all of our students each year. Myself included, and in fact I am still slightly spooked by them. While I cannot teach you all how exactly to deal with them, I can teach you all of use them day to day.

S4 objects are not lax, they are the stricter implementation of S3 objects. The classes which compose S4 objects are incredibly well defined, and they are designed for very specific use cases. This may at times make them obnoxious to work with, when you need to build or modify values in them, but thye are worth the pain.

```
writeLines(otype(dec_lakes_sp))
```

```
S4
```

```
writeLines(s3_class(dec_lakes_sp))
```

```
SpatialPolygonsDataFrame
```

8.3.1 Spatial* Objects

Just like as in S3 objects in S4 objects the classes follow certain schema. Note we use a SpatialPolygonsDataFrame as our example here.

In addition to their more strict definitions of classes, S4 objects have another feature which S3 objects lack - Slots. Slots superficially resemble lists in the viewer of RStudio, i.e. they have nested components, but are really their own distinct entities. A SpatialPolygonsDataFrame contains 4 slots, each of these contains a type of data. Note that each slot is accessed using an '@' (still pronounced: 'at') symbol.

```
dec_lakes_sp@plotOrder
```

```
 [1]  1 11  7 13  2  8  3 12 10  5  6  4 14  9
```

```
otype(dec_lakes_sp@plotOrder)
```

```
[1] "base"
```

```
writeLines(s3_class(dec_lakes_sp@plotOrder))
```

```
integer
numeric
```

Here we access the contents of a slot named 'plotOrder'. We see that this slot simply contains an integer vector. What this vector specifies is the order in which the polygons in this object should be plotted if and when used for making maps. While this information is stored simply as an object of the class 'base', other parts of the SpatialPolygonsDataFrame know how to perform operations with this information.

```
head(dec_lakes_sp@data)[,c(1:3,5)]
```

```
Warning in as.POSIXlt.POSIXct(x, tz): unknown timezone 'US/Pacific-New'
```

```
  id          Water Data_source       Date
1  1   Pyramid_Lake   Sentinel2 2021-12-05
2  2     Davis_Lake   Sentinel2 2021-12-05
3  3 Frenchman_Lake   Sentinel2 2021-12-05
4  5    Silver_Lake   Sentinel2 2021-12-05
5  6      Swan_lake   Sentinel2 2021-12-05
6 12      Gold_Lake   Sentinel2 2021-12-05
```

We can see that the 'DataFrame' portion of the SpatialPolygonsDataFrame has actually been relegated to it's own slot as well.

```
attributes(dec_lakes_sp@data)
```

```
$names
[1] "id"          "Water"       "Data_source" "Processing"  "Date"

$class
[1] "data.frame"

$row.names
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14
```

```
otype(dec_lakes_sp@data)
```

```
[1] "S3"
```

```
writeLines(s3_class(dec_lakes_sp@data))
```

```
data.frame
```

While the two slots above, @plotOrder & @data, contain relatively simple objects (base and S3 respectively). The remaining two slots, which are both S4 classes contain all of the spatial information.

```
CRS
```

```
[1] "S4"
```

```
$projargs
[1] "+proj=longlat +datum=WGS84 +no_defs"
```

```
$class
[1] "CRS"
attr(,"package")
[1] "sp"

$comment
[1] "GEOGCRS[\"WGS 84\",\n    DATUM[\"World Geodetic System 1984\",\n        ELLIPSOID[\"WGS 84\",63781
```

The first of the S4 slots is the proj4string (Proj is an awesome open source library for performing conversions between projections), which contains a class "CRS" which defines the Coordinate Reference System of this SpatialPolygonsDataFrame. Hence any time you modify the CRS of a SPDF, you are interacting with this slot.

The final, and most complex of the two S4 slots in a SpatialPolygonsDataFrame is the Polygons slot. This is where the coordintaes, and topological information, fo every single polygon is stored.

```
S4
```

```
$Polygons
$Polygons[[1]]
An object of class "Polygon"
Slot "labpt":
[1] -120.26621   40.15194

Slot "area":
[1] 0.0002266631

Slot "hole":
[1] FALSE

Slot "ringDir":
[1] 1

Slot "coords":
          [,1]     [,2]
[1,] -120.2684 40.16119
[2,] -120.2577 40.16024
[3,] -120.2604 40.14826
[4,] -120.2657 40.14006
[5,] -120.2657 40.14006
[6,] -120.2758 40.14942
[7,] -120.2684 40.16119



$plotOrder
[1] 1

$labpt
[1] -120.26621   40.15194

$ID
[1] "9"

$area
[1] 0.0002266631
```

```
$class
[1] "Polygons"
attr(,"package")
[1] "sp"

$comment
[1] "0"
```

Here we are looking at the smallest of the polygons in our SPDF (see the plotOrder?). These data are replicated appropriately for each of the other polygons in this object.

While this S3 object is more complex, we can see how the components held in multiple slots are able to work together to perform operations using the data held in disparate fields throughout the object.

8.3.2 Raster Objects

Finally we have the Raster Layer which is a rather large and complex S4 unit, composed of 12 main S4 slots... Each of these then having from 1 to 13 slots (these second slots quite small, and not uncommonly consisting of a single value).

```
Formal class 'RasterLayer' [package "raster"] with 12 slots
  ..@ file    :Formal class '.RasterFile' [package "raster"] with 13 slots
  .. .. ..@ name        : chr ""
  .. .. ..@ datanotation: chr "FLT4S"
  .. .. ..@ byteorder   : chr "little"
  .. .. ..@ nodatavalue : num -Inf
  .. .. ..@ NAchanged   : logi FALSE
  .. .. ..@ nbands      : int 1
  .. .. ..@ bandorder   : chr "BIL"
  .. .. ..@ offset      : int 0
  .. .. ..@ toptobottom : logi TRUE
  .. .. ..@ blockrows   : int 0
  .. .. ..@ blockcols   : int 0
  .. .. ..@ driver      : chr ""
  .. .. ..@ open        : logi FALSE
  ..@ data    :Formal class '.SingleLayerData' [package "raster"] with 13 slots
  .. .. ..@ values    : num [1:13924] 1 1 1 1 1 1 1 1 1 1 ...
  .. .. ..@ offset    : num 0
  .. .. ..@ gain      : num 1
  .. .. ..@ inmemory  : logi TRUE
  .. .. ..@ fromdisk  : logi FALSE
  .. .. ..@ isfactor  : logi FALSE
  .. .. ..@ attributes: list()
  .. .. ..@ haveminmax: logi TRUE
  .. .. ..@ min       : num 0
  .. .. ..@ max       : num 1
  .. .. ..@ band      : int 1
  .. .. ..@ unit      : chr ""
  .. .. ..@ names     : chr ""
  ..@ legend  :Formal class '.RasterLegend' [package "raster"] with 5 slots
  .. .. ..@ type      : chr(0)
  .. .. ..@ values    : logi(0)
  .. .. ..@ color     : logi(0)
  .. .. ..@ names     : logi(0)
  .. .. ..@ colortable: logi(0)
```

```
  ..@ title   : chr(0)
  ..@ extent  :Formal class 'Extent' [package "raster"] with 4 slots
  .. .. ..@ xmin: num 697130
  .. .. ..@ xmax: num 811776
  .. .. ..@ ymin: num 4388466
  .. .. ..@ ymax: num 4502382
  ..@ rotated : logi FALSE
  ..@ rotation:Formal class '.Rotation' [package "raster"] with 2 slots
  .. .. ..@ geotrans: num(0)
  .. .. ..@ transfun:function ()
  ..@ ncols   : int 118
  ..@ nrows   : int 118
  ..@ crs     :Formal class 'CRS' [package "sp"] with 1 slot
  .. .. ..@ projargs: chr "+proj=utm +zone=10 +datum=WGS84 +units=m +no_defs"
  .. .. ..$ comment: chr "PROJCRS[\"unknown\",\n    BASEGEOGCRS[\"unknown\",\n        DATUM[\"World Geo
  ..@ history : list()
  ..@ z       : list()
```

Raster Layers are large enough that we unfortunately cannot go into it much during class, but I do encourage the curious you to investigate it on your own. Do keep in mind that it has a slot of 'CRS' which we just saw in our SP object, and it also has a slot of 'extent' which we defined ourselves earlier.

```
Class: S4

Class type: RasterLayer

.SingleLayerData

double
numeric
```

If we look at our data slot we see that we have an 'integer numeric' type - which is typically how a raster is loaded. I will concede up to this point I said to think about a raster as matrix, but we see ours is actually loaded with an integer. Remember that in R both matrices and data frames are vectors. Remember a main difference between a vector and a matrix, is that a matrix has dimensions, i.e. it knows how many values need to be in each row. While these data are not explicit here, they can be found in another slot.

9 Object Orientated Programming Bonus: Make our own S3 and s4 objects.

So we approached S3 and S4 objects from a very top down approach there, we looked at those constructed by others. But we can also create our own objects, so we can look at these from the bottom up.

9.1 Create a simple S3 object

Maybe none of you have noticed yet but you will oon enough; your TA has some traits in common with say Luna Lovegood, and finds it hard to make it to meetings and things. Here we makes a course object which he can save into his R environment so he can just click on it to remind him when he is suppose to show up for class.

```
course <- list(
  name = c("Lecture", "Lecture", "Laboratory", "Laboratory"),
  wing = c('L', 'L','M','L'),
  room = c(170, 170, 166, 62),
  day  = c('T', 'TH', 'F', 'F'),
  time = c('3:30-4:50', '3:30-4:50', '12:00-12:50', '2:00-3:50')
```

```
)

writeLines(sloop::s3_class(course)) # we have only made a list so far
```

```
list
```

```
writeLines(sloop::otype(course))
```

```
base
```

```
class(course) <- "Class_times" # by setting a class attribute we have
# created an S3 object

writeLines(sloop::s3_class(course))
```

```
Class_times
```

```
sloop::otype(course)
```

```
[1] "S3"
```

```
rm(course)
```

We see that the above object is honestly, just a list that we arbitrarily made an S3 object. That is more or less what it takes to become an S3 object, us just saying hey 'this is a class' !

But we can do things with S3 objects which make them useful to construct. For example, we can add quality assurance checks to our objects.

9.2 Create a more complex S3 object

Your TA made a very poor first list of places he had to be for class, and ended up wandering around tech lost for an hour. He decided to make a slightly more robust Sw object so this did not happen again.

```
course <- function(n, w, r, d, t){

  values <- list(name = n,
                 wing = w,
                 room = r,
                 day = d,
                 time = t)

  '%notin%' <- Negate('%in%')
  type <- c('Lecture', 'Laboratory', 'Seminar')
  tech_letters <- LETTERS[1:13]
  days <- c('M', 'T', 'W', 'TH', 'F', 'S', 'SU')

  if(any(w %notin% tech_letters)) stop("This wing is not valid")
  if(any(d %notin% days)) stop("This day is not valid")
  if(any(n %notin% type)) stop("This course type is not valid")

  attr(values, "class") <- "Course"
  return(values)

}
```

```
course_success <- course(c("Lecture", "Lecture", "Laboratory", "Laboratory"),
                         c('L', 'L','M','L'),
```

```
                        c(170,170,166,62),
                        c('T','TH','F','F'),
                        c('3:30-4:50','3:30-4:50', '12:00-12:50', '2:00-3:50')
                        )
```

Here we see that an object of the S3 class can be much more than merely a collection of attributes. An S3 object can perform quality assurance steps to ensure the data comply to certain types, and that values are in appropriate units etc.

If the data are these thoroughly defined, we then see this opens the opportunity for an S3 object to act upon other parts of itself. While in the example above we put in values which match the acceptable criteria of the class we have defined, in the example below will violate the standards of our S3 Object.

```
course_fail <- course("Lecture",
            'Z', # THIS IS NOT DEFINED
            213, 'TH', '2:00-3:50')
```

simulated output (R does not like errors, intentional or not):

"Error in course("Lecture","Z", 213,"TH","2:00-3:50") : This wing is not valid"

In the above example, if you input the wrong Wing in Tech, or the wrong day of week abbreviation, this class will angrily let you know and refuse to take your input. I assume this error can be relegated to a warning - but we will not get into those aspects of coding in this class.

One, possible, draw back of an s3 object is that it is relatively lax.

9.3 Create a simple S4 object

The s4 object is not lax. Here we define the data type which each of these slots will accept. If the value you try to put in does not match, the object will not be created.

```
setClass("Office_Hours",
         slots=
           list(
            Instructor="character",
            Wing= "character",
            Number="numeric",
            Days="character",
            Time="numeric",
            Smartroom="logical",
            Windows="logical"
            )
         )
```

- Seven slot object, each slot with a specified data type.
- Will only input to each column of the correct data type.

```
s4_ob_office_hours <- new("Office_Hours",
  Instructor = c('Benkendorf','Scholl','Benkendorf','Scholl'),
  Wing = c('F', 'F', 'G', 'B'),
  Number = c(380, 380, 278, 138),
  Days = c("M", "W", 'T', 'F'),
  Time = c(8, 11, 9, 3),
  Smartroom = c(TRUE, F, T, F),
  Windows = c(FALSE, T, T, F)
)
```

In it's simplest form an s4 object may be constructed via the setClass function. While this object is able to regulate the data types which are entered to each column, it cannot do much more.

9.4 Create a more complex S4 object

An S4 object can have validity functions, which basically ensures the data you put into it is appropriate.

```
office_hrs <- setClass("Office_Hours",

         slots=c(
           Instructor="character",
           Wing= "character",
           Number="numeric",
           Days="character",
           Time="numeric",
           Smartroom="logical",
           Windows="logical"
           ),

        validity=function(object){

            '%notin%' <- Negate('%in%')
            tech_letters <- LETTERS[14:26]
            days <- c('M', 'T', 'W', 'TH', 'F', 'S', 'SU')
            instructors <- c('Scholl', 'Benkendorf')

            if(any(object@Instructor != instructors))stop("This Instructor is not valid")
            if(any(object@Wing %in% tech_letters))stop("This Wing is not valid")
            if(any(object@Days %notin% days)) stop("This Day is not valid")
  }
)
```

- Mandates the appropriate data type is entered
- Checks that the 'Wing' we enter exists in the Tech Building
- Ensures that a valid Instructor is entered
- Ensures that the appropriate abbreviation for a day is entered.
- Object clearly capable of performing procedures on itself.

```
s4_ob_office_hours <- office_hrs(
  Instructor = c('Scholl', 'Benkendorf', 'Scholl', 'Benkendorf'),
  Wing = c('A', 'F', 'B', 'B'),
  Number = c(123, 412, 278, 138),
  Days = c("M", "W", 'T', 'F'),
  Time = c(8, 11, 9, 3),
  Smartroom = c(TRUE, F, T, F),
  Windows = c(FALSE, T, T, F)
)

otype(s4_ob_office_hours)
```

```
[1] "S4"
```

```
s3_class(s4_ob_office_hours)
```

```
[1] "Office_Hours"
attr(,"package")
```

```
[1] ".GlobalEnv"
```

```
rm(s4_ob_office_hours, office_hrs)
```

- Easy examples, but we could also script in conversion functions like with the ealier Units and POSIX S3 objects.
- S3/S4 objects can have a lot going on 'under the hood'
- At their heart, they are performing operations on themselves.

You will realize in short time, that the biggest hurdle to dealing with spatial data in R is how complex some of the structures may be. But I guarantee you all have seen more of the intricate workings of these objects than the vast majority of folks which utilize them. Just remember, to appease the validity functions and you will be fine.

10 Assignments for the Duration of the Spatial Data Science Module:

For next Lab:

Please install these packages (if you have not done so already):

```
install.packages("sf", "raster", "terra", "sp", "tmap", "leaflet", "ggmap")
# optional packages for using parallel processing at a step
# (in our example it won't actually speed up the analyses
# they actually may slow them down!)
install.packages('snow','parallel')
```

Download the labs .R script from the course website.

If you are interested in how Drone/Lidar data are collected please check out the 'RMBL Spatial Data Science Webinar Series':

- https://github.com/ikb-rmbl/SpatialDataScienceWebinars2020
- Collecting UAS Data (Video): https://youtu.be/Pq8btEZRCvM (1.25 hours)

For next Lecture: Assigned Reading: Chapter 3 of Spatial Data Science

Future Bonus SDS Office Hours Wednesday Night at 5:00 - 6:00.

Notes on this Lab My lecture notes are in the R script I used to generate all of the novel figures for this presentation. Likewise this presentation is an .HTML file and can be launched from your computer (it was rendered directly from R using the script).

11 Works Cited

Krystalli, A. https://annakrystalli.me/intro-r-gis/gis.html Accessed 01.20.2022

Advanced R. Wickham, H. https://adv-r.hadley.nz/index.html Accessed 01.09.2022

https://geocompr.robinlovelace.net/spatial-class.html Accessed 01.09.2022

Hijman, R. 05.12.2019 'The raster Package'

https://www.datamentor.io/r-programming/s3-class/ Accessed 01.18.2022

https://rspatial.org/raster/RasterPackage.pdf Accessed 01.09.2022

https://www.neonscience.org/resources/learning-hub/tutorials/dc-multiband-rasters-r Accessed 01.19.2022

Pebesma, E.J., R.S. Bivand, 2005. Classes and methods for spatial data in R. R News 5 (2).

Pebesma, E. https://r-spatial.github.io/sf/articles/sf1.html Accessed 01.10.2022

https://proj.org/operations/conversions/geoc.html Accessed 01.20.2022.

https://rspatial.org/raster/spatial/8-rastermanip.html Accessed 01.11.2022

https://en.wikipedia.org/wiki/Open_Source_Geospatial_Foundation Accessed 01.09.2022

https://cran.r-project.org/web/packages/vctrs/vignettes/s3-vector.html Accessed 01.14.2022

https://en.wikipedia.org/wiki/Object-oriented_programming Accessed 01.19.2022

NOAA. What is geodesy? National Ocean Service website, https://oceanservice.noaa.gov/facts/geodesy.html, 1/25/17.

Geocomputation with R. Lovelace, R., Nowosad, J., Muenchow, J. 2022-01-25.