“Warning: Discarded datum Unknown:” What Does it Mean?

Richard E. Plant

May 27, 2022

Additional topic to accompany *Spatial Data Analysis in Ecology and Agriculture using R, Second Edition*

http://psfaculty.plantsciences.ucdavis.edu/plant/sda2.htm

Chapter and section references are contained in that text, which is referred to as SDA2.

1. Introduction

The reason that I felt the need to write this Additional Topic is shown in the following code sequence, taken from SDA2 Section 1.3. The objective is to create the image shown in Fig. 1, which is Fig. 1.5 in SDA2.The code is from the file 1.3.1.r on the SDA2 website.

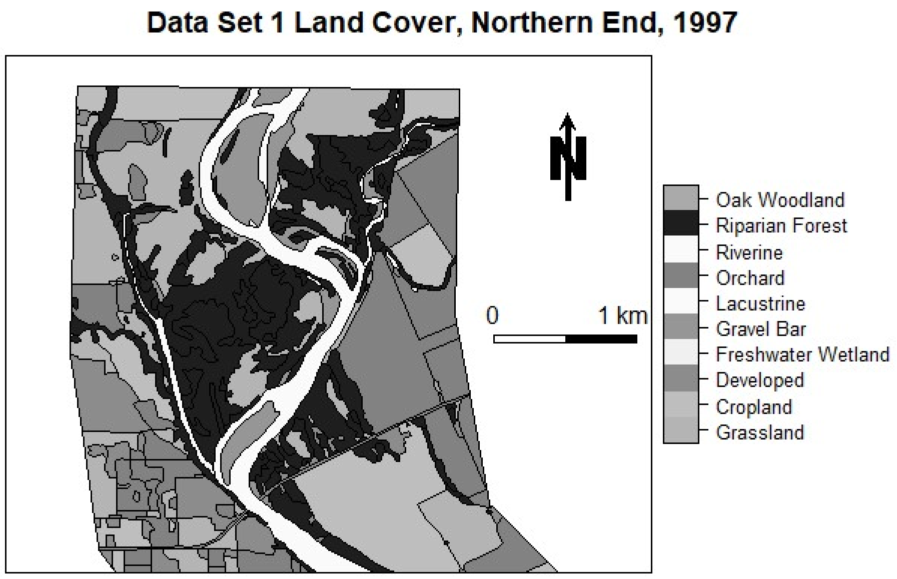

Figure 1. Reproduction of Figure 1.5 in SDA2.

The first step is to read the shapefile containing the data.

```
> library(rgdal)
> library(sf)
> data.Set1.sf <- st_read("set1\\landcover.shp")
Reading layer `landcover' from data source `C:\Data\Set1\landcover.shp' using
driver `ESRI Shapefile'
Simple feature collection with 1811 features and 6 fields
geometry type:  POLYGON
dimension:      XY
bbox:           xmin: 576446.9 ymin: 4396700 xmax: 589578.4 ymax: 4420790
CRS:            NA
```

Both of the `library()` function calls generate output that we will skip over for now but come back to in a moment. After reading it, we convert the `sf` object `data.Set1.sf` to an `sp` object and assign a projection. Here something unexpected happens.

```
> data.Set1.cover <- as(data.Set1.sf, "Spatial")
> proj4string(data.Set1.cover) <- CRS("+proj=utm +zone=10 +ellps=WGS84")
Warning message:
In showSRID(uprojargs, format = "PROJ", multiline = "NO") :
  Discarded datum Unknown based on WGS84 ellipsoid in CRS definition
```

SDA2 was written using R version 3.4.3, and there was no warning message. The code above was run using R version 4.0.2, so clearly something has changed. Now, on the one hand, this is only a warning. The code still runs, and the continuation was used to produce Fig. 1. On the other hand, today's warning may be tomorrow's error message, so it is probably a good idea to figure out what is going on. Moreover, most practitioners (myself included) generally have handled geodetic input/output and calculations by copying and pasting code from a trusted source. This is not a good idea if done blindly, however, and the purpose of this Additional Topic is to lay out what is going on "under the hood" when R does these operations, and why changes were necessary that ultimately led to warning messages like the one above. Armed with this knowledge, the practitioner can move forward and make the appropriate code changes as R evolves.

We will start by going back to the output produced by the function call `library(rgdal)`. Here are the last two lines.

```
To mute warnings of possible GDAL/OSR exportToProj4() degradation,
use options("rgdal_show_exportToProj4_warnings"="none") before loading rgdal.
```

The function in the code sequence above that gives the warning is `proj4string()`. This, together with the statement just printed, indicates that we should look here to understand the warning. In the code sequence above the function `proj4string()` assigns a coordinate reference system to the object `data.Set1.cover` by calling the function `CRS()`. The function `CRS()` accesses an *application programming interface*, or API, called PROJ. R depends on PROJ and two other APIs, GDAL and GEOS, to handle spatial data input, output, and manipulation. For our purposes, the fundamental issue in understanding how R handles spatial data is to understand how R interacts with these APIs. To understand the cause of the warning we saw above we need to appreciate how and why changes to these APIs were made.

Wikipedia defines an application programming interface as follows. "An application programming interface (API) is a computing interface which defines interactions between multiple software intermediaries. It defines the kinds of calls or requests that can be made, how to make them, the data formats that should be used, the conventions to follow, etc." Fig. 1, which is extracted from a more complex figure in Bivand (2020), shows the relationships between the package `sf` and the three APIs we will be discussing. The relationships between the `sp` package and these APIs is similar.

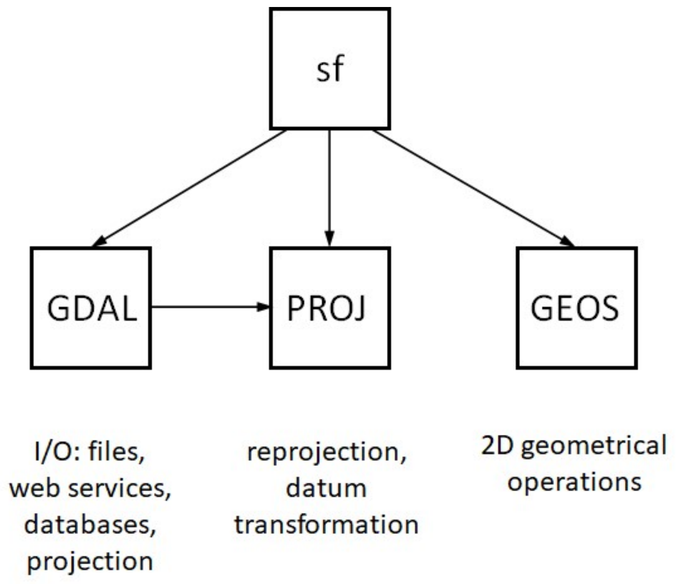
sf
GDAL
PROJ
GEOS
I/O: files, web services, databases, projection
reprojection, datum transformation
2D geometrical operations

Figure 1. Relationships between `sf` and open source APIs. Based on a portion of a similar figure in Bivand (2020). The direction of the arrows indicates how information is accessed, so for example `sf` accesses GDAL which in turn accesses PROJ.

The GEOS API represented in Fig. 1 is responsible for carrying out GIS operations such as union and intersection and does not really enter into our discussion, so we will not consider it further.

The changes in the GDAL and PROJ APIs that ultimately led to the warning message we saw above can be traced to two primary factors: the dramatic increase in precision with which location on the earth can be determined as a result of the introduction of the GPS, and the evolution of the software making up the APIs, including both the code in which they are written and the environment in which they are used. In order to fully understand the first issue, the increase in precision, it is necessary to know the basics of how location on the earth is identified, that is, the basics of geodesy. This is the topic of Section 2. Section 3 contains a brief review of the concept of open-source software and its implications for the construction, maintenance and use of these packages. Section 4 discusses the changes that have been made in how GDAL and PROJ handle data and how these changes led to the generation of warning messages such as the one we saw above. Section 5 describes the specific parts of the software that produced the warning messages, and the motivation for displaying them. It concludes by describing how to modify the R code to avoid the warning. The transition discussed here is a part of a larger transition in which the packages sp and raster are being replaced by sf, terra, and stars. To aid readers of SDA2 in making this transition I have rewritten all the book's code using these new packages. A discussion of the transition together with the code is available in the *Additional Topics* section of the book's website, http://psfaculty.plantsciences.ucdavis.edu/plant/sda2.htm.

2. Review of Geodesy

Wikipedia defines geodesy as "the Earth science of accurately measuring and understanding Earth's geometric shape, orientation in space and gravitational field." If we are using R to carry out spatial data analysis, then our particular interest lies in correctly reading, writing, and manipulating spatial data, that is, data referring to locations on the earth.

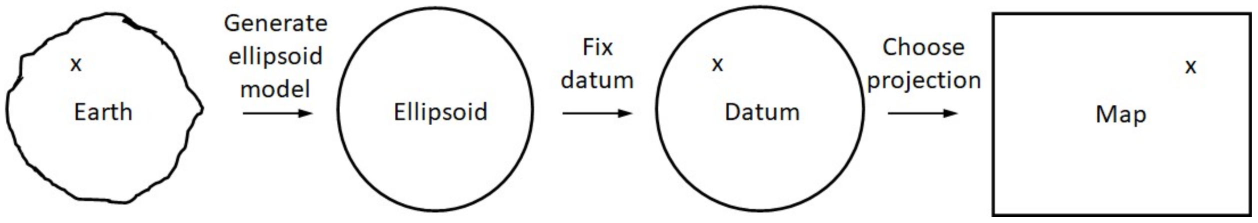

Figure 2. Stages in moving from a location on the earth to a map. Modified from Fig. 2.15 of Lo and Yeung (2005, p. 36). The "x" marks a particular location on the earth, which is transferred to

the map. It is missing from the ellipsoid model because that model has not yet been fixed to the earth as a datum.

The results of spatial analysis are frequently displayed in the form of maps in which the approximately spherical surface of the earth is mapped onto the flat surface of the computer screen or paper. The major steps of this process are displayed in Fig. 2. The first step is to generate an ellipsoid model of the surface of the earth. The irregularity of the "earth" shown in Fig. 2 is of course greatly exaggerated; in reality the deviation of the earth from a perfect sphere would, at the scale of the figure, be undetectable by eye. The most widely used ellipsoid model has an equatorial radius of 6,378,137 m and a polar radius of 6,356,752.3142 m, for a difference of about 21,385 m, or 0.3%.

Once the ellipsoidal model is established, the next step is to fix it to the earth in the form of a datum. The ellipsoid described in the previous paragraph is used to generate the WGS 84 datum as follows. The center of the ellipsoid is matched with estimated location the center of mass of the earth, which is known to within a few meters. The "equator" of the ellipsoid, defined by its two largest principal axes, is matched to the equator of the earth, which can be approximated as a plane perpendicular to the axis of rotation of the earth and halfway between the north and south poles. Because of slight wobbles and irregularities, the actual location of the earth's equator must be fixed by international convention. Once the equator of the ellipsoid is matched with the earth's equator, the final step in the creation of the WGS 84 datum is to establish the *prime meridian*, which is the location of 0° longitude. This is located 5.3 arc seconds or 102 meters east of the Greenwich meridian at the latitude of the Royal Observatory in Greenwich, UK.

If one stands at a particular location on the surface of the earth in, say, North America or Australia, and records a very precise location using a GPS, and then returns a few years later to the exact same location, the GPS coordinates will have changed slightly. This is because the GPS is providing the location on the WGS 84 datum and, due to plate tectonics, the continents are moving "underneath" that datum, which is fixed to a location in Great Britain. Moreover, the WGS 84 datum may provide a less accurate representation of specific parts of the earth than other datums. For this reason, countries that occupy a continent located on a different plate from Great Britain will often employ a datum fixed to a location on their own plate, and periodically adjust their coordinates relative to those of WGS 84. For example, the United States and Canada use the NAD 83 datum, and Australia uses the Geocentric Datum of Australia. Because datums may change with time, when one specifies a location by a set of coordinates, one must specify not only the datum (e.g., WGS 84 or NAD 83), but also the date on which that datum was referenced. This latter specification is called the *epoch* of the datum.

The final step in transferring locational data to a map is the process of projection, in which the angular coordinates (ϕ,λ) on the ellipsoid are projected to coordinates (x,y) on the map. A note of caution: here traditionally ϕ represents *latitude*, which corresponds to the y coordinate, and λ represents *longitude*, which corresponds to the x coordinate. Note that these are in opposite order; we will return to this point in Section 5. Returning to the issue of projection, the formulas for the UTM projection are given on this website. I am not going to reproduce them here and when you visit the site you will see why. You should visit the site, however, to get an idea of how complex these calculations can be, and why approximations may be necessary.

The action of finding a precise location on the datum, and by implication on the surface of the earth, is called *georeferencing*. Prior to the advent of the global positioning system just about the only endeavor in which an accuracy of more than a few meters was required was surveying, which developed its own methodology. For more general mapping the usual standard for precision (accuracy and precision are discussed in SDA2 Sec. 5.2.2) was the width of a line on a map, which by convention was taken to be 0.5 mm. For example, fitting a map of a 25 ha field on a sheet of writing paper or a computer screen requires a map scale of about 1:2000. At this scale, 0.5 mm corresponds to 1 m. This is representative of the greatest precision that was generally required in georeferencing at that time. For maps of a larger area, correspondingly less precision was required. Although the GPS on a cell phone is currently only capable of about 1 m of accuracy, real-time kinetics (RTK) systems such as those currently used with tractor autosteering equipment are accurate to the centimeter level. This great increase in accuracy, and concomitant need for increased precision, is part of what has driven the modifications in the software. Before taking up the issues associated with this specific software, however, it is worthwhile to pause and briefly consider the broader picture. That is the subject of the next section.

3. Open Source Software: Benefit and Responsibility

Many data analysts of a certain age got their start with packaged statistical software using SAS, manufactured by the SAS Institute in Cary, North Carolina. The SAS Institute is a for profit company, and SAS software must be purchased. If you write a SAS program, there are various ways that you can indicate the location of the data. One of these is the statement “CARDS.” This is a legacy of the fact that at one time data values, along with everything else, were entered into the computer via IBM cards. If you don’t know what an IBM card is or have never seen one, look here. Probably no-one has entered data into a computer via IBM cards in the last 30 years, and the statement is no longer mentioned in SAS manuals, but, as far as I know, it still works. The reason that SAS maintains this sort of backward compatibility is this. Imagine you are working as a data analyst for a company and your team has been running the same SAS program for several years in routine data analysis. The program was originally written by a colleague who is now retired. If one day after a software update the code suddenly crashes and gives you an error message, you are in for a lot of work figuring out what went wrong, and your company is not going to be pleased with SAS. When people pay money for something that causes them too many problems, they will take their business elsewhere. Tracking down the reason that a program has suddenly stopped working and figuring out what to do about it is one such problem. GDAL, PROJ, and GEOS, on the other hand, are projects of the Open Source Geospatial Foundation, or OSGeo. According to its website, OSGeo “is a not-for-profit organization whose mission is to foster global adoption of open geospatial technology by being an inclusive software foundation devoted to an open philosophy and participatory community driven development.” R itself is also open-source software. The people who write open-source code, although they may receive some indirect compensation, are basically volunteering their time for the general benefit of the user community. Many users of R (again, myself included) have at one time or another been confronted with an R error message and a statement that some function has been “deprecated.” This can be annoying and involve some extra work, but it is a consequence of the fact that when you use open-source code you are not a customer but rather a member of a community in which everyone must do their part. As a user, in return for free access to this great software you must accept the responsibility of understanding your code and keeping it up to date as the software changes.

4. The Evolution of PROJ

Fig. 1 in Section 1 illustrates the flow of information among the APIs used in transferring and processing spatial data. We have already said that the GEOS API shown in the figure is actually peripheral to the story and can be removed from the discussion. Pebezma and Bivand (2020) describe concisely but specifically the functions of GDAL and PROJ. To quote them, "PROJ provides methods for coordinate representation, conversion (projection) and transformation, and GDAL allows reading and writing of spatial raster and vector data in a standardized form and provides a high-level interface to PROJ for these data structures, including the representation of coordinate reference systems (CRS)." The issues that led to the modifications in the software resulting in the warning message involve representing and transforming coordinate reference systems and transformations, and thus they come down to PROJ. Our remaining discussion will therefore focus entirely on this API.

We have already seen one major change in circumstance whose ultimate consequence was the warning message we saw in Section 1: the great increase in accuracy of georeferencing. To understand the second major change, it helps to consider the history of the PROJ software, as described by Butler (2017) and the PROJ.6 website. The software that eventually became PROJ was initially developed in the late 1970s by Jerry Evenden, working for the US Geological Survey for use in ocean floor mapping. At that time all computing was done using central mainframe computers. Computer memory was at a premium and processing speed was, compared to today, glacial, so programming that required complex mathematical calculations frequently included simplifying approximations. Provided that the resulting answer fit within the bounds of the required precision, there was nothing lost by using these approximations. As with most software of the time, PROJ was originally written in FORTRAN. In 1985 the code was rewritten in C and called PROJ.2. In 1990 a further updating was called PROJ.3, and PROJ.4 followed in 1995. After this Evenden stopped working on the code and it remained unchanged for several years. In 1998, after Evenden's retirement, Frank Warmerdam began using the code for a different USGS project. In the meantime, the GRASS GIS, originally developed by the US Army Corps of Engineers in the 1980s, had incorporated Evenden's code, which thus was moved into the open-source domain. It was, however, still far from the form that could be used by an R package. Warmerdam expanded the capacity of the software in very many ways, but maintained the fundamental PROJ.4 structure, which did not change for almost twenty years.

At the time PROJ was initially developed in the 1970s the WGS 84 coordinate reference system, which was established in 1984, did not exist. At that time the CRS in almost universal use was based on the Clarke 1866 ellipsoid, so the issue of converting between coordinate reference systems did not come up. As PROJ was moved into the API domain and evolved into PROJ.4, it became necessary to have a simple means of specifying factors such as the CRS and projection. This led to the creation of the *proj4string*. This is the character string that we see in all calls of the `proj4string()` function. It consists of a series of statements, each starting with a + and followed by an assignment. We saw an example of this in the line of code in Section 1, which is repeated here:

```
> proj4string(data.Set1.cover) <- CRS("+proj=utm +zone=10 +ellps=WGS84")
```

Gimond (2020) describes the proj4string syntax. The function `proj4string()` is a *replacement function* (see SDA2 Section 2.4.3), so that it can appear on either the left hand side or the right hand side of the assignment operator. We can therefore display the projection information of the object `data.Set1.cover` as follows

```
> proj4string(data.Set1.cover)
[1] "+proj=utm +zone=10 +ellps=WGS84 +units=m +no_defs"
Warning message:
In proj4string(data.Set1.cover) :
  CRS object has comment, which is lost in output
```

Again we get a warning message. Before considering this message, let's look at the output. There are five parameters: `proj`, `zone`, `ellps`, `units`, and `no_defs`. The meanings of the first four are obvious. Warmerdam explains that the parameter `no_defs` "ensures that no defaults are read from the defaults files." The parameter `no_defs` is now obsolete.

Now we can move on to this warning message and the one we saw in Section 1 (these result from the same change in the code). At around the same time as the development of the proj4string syntax, the Open Geospatial Consortium, or OGC, began the development of a standardized, text-based format for the representation of coordinate reference systems, called "Well Known Text," or WKT. It is sometimes called WKT-CRS to distinguish from a different, but presumably equally well known, text used to describe geometry, and it is also called WKT2 because it is the second version of the standard. We will use the latter abbreviation, WKT2. Evolution of the WKT2 standard proceeded apace, and by 2015 it had basically reached the level at which it stands today. The development of PROJ4 had meanwhile become somewhat stagnant, and the decision was reached in 2019 that a new upgrade was needed. In the meantime, the `sf` package had been developed based on the *simple features* concept, also formulated by the OGC. The other WKT that I just mentioned is used to describe simple features. It therefore made complete sense to abandon the proj4string representation in favor of WKT2. As we will see in a bit, however, WKT2 text is too complex to be manually entered into computer code. Therefore it is necessary to have a means to specify the CRS and projection in a simple but unambiguous way. That is the next stage in our story.

At around the same time that Evenden was developing the software that would become PROJ and the Army Corps of Engineers was developing GRASS, the Northern European countries involved in exploration for oil in the North Sea were finding that they had different and incompatible coordinate reference systems and thus could not accurately exchange information about geographic locations at sea, where there are no landmarks to serve as guides. This led to the development of the European Petroleum Survey Group, or EPSG, which set about the task of establishing a uniform mapping system that all the countries could use. This system uses a numerical code to identify each coordinate refence system and each projection using that CRS. For example, the EPSG code for the WGS 84 CRS is 4326. We can use the `sf` function `st_crs()`, which is also a replacement function, to assign or display a CRS or projection in the same way that we used `proj4string()`. Let's use it to display the WKT2 representation of the WGS 84 CRS.

```
> st_crs(4326)
Coordinate Reference System:
  User input: EPSG:4326
  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["geodetic latitude (Lat)",north,
            ORDER[1],
            ANGLEUNIT["degree",0.0174532925199433]],
        AXIS["geodetic longitude (Lon)",east,
            ORDER[2],
            ANGLEUNIT["degree",0.0174532925199433]],
    USAGE[
        SCOPE["unknown"],
        AREA["World"],
        BBOX[-90,-180,90,180]],
    ID["EPSG",4326]]
```

In 2005 the EPSG was disestablished and its function was taken over by the International Association of Oil and Gas Producers, but the abbreviation EPSG is still used. This representation of projections and CRS has become the worldwide standard. Note that latitude is displayed first in the EPSG representation.

In the code segment in Section 1 we read into memory the `sf` object `data.Set1.sf` and then immediately converted it into the `sp` object `data.Set1.cover`, to which we assigned a projection. We never assigned a projection to the original `sf` object, so let's do it now as an example. The first step is to find out the EPSG code for UTM Zone 10N. One can go to the website https://epsg.org/home.html and enter "UTM Zone 10N" in the search box. The site returns the codes for several coordinate reference systems, and in particular the one for UTM Zone 10N is 32610. (EPSG code can also be found at the site http://epsg.io/). We can now assign a projection to the object `data.Set1.sf` and then display it.

```
> st_crs(data.Set1.sf) <- 32610
> st_crs(data.Set1.sf)$proj4string
[1] "+proj=utm +zone=10 +datum=WGS84 +units=m +no_defs"
> st_crs(data.Set1.sf)$epsg
[1] 32610
```

Typing `st_crs(data.Set1.sf)` returns the full WKT2 representation, which is quite long (try it!). Notice, however, that, unlike with the function `proj4string()`, there are no warnings.

In summary, software that had been originally written in a coding language developed in the days of mainframe computers for calculations whose accuracy and precision was measured in meters was now being used in ways far beyond the scope of their original design for calculations whose accuracy and precision is measured in centimeters. The individuals responsible for maintaining this software, volunteers who donate their time for the good of the cause, made the decision that a complete overhaul was necessary. In the next section we will describe what this overhaul involved and why the warnings that it engendered will actually benefit us all in the long run.

5. The Impact of the Changes

Looking again at the WKT2 character string describing the WGS 84 CRS and comparing it with the proj4string for the UTM Zone 10N projection, which should require more information to completely specify it, we can see that the proj4string syntax, which was developed when precision and accuracy requirements were much lower, cannot completely specify all the details of a projection. Moreover, the proj4string syntax contains two means for specifying a datum, "+datum=" and "+towgs84=", and it is possible in principle for these to provide contradictory information. The WKT2 specification was already available and so the decision was made to drop the proj4string syntax in favor of WKT2. For now, compatibility with the proj4string syntax will be retained, but this cannot be guaranteed forever. However, a warning message was placed into the code so that users of the proj4string syntax would be alerted to the changes and have the time to learn about them and adjust their code accordingly.

Here we will take advantage of one of the great features of open-source software to track down the source of the warning, which will help to explain why it was inserted where it was. This great feature is that the source code is open, i.e, accessible, and we can use it to penetrate the code and, in the process, see the programming of people who write really solid code. We did this in the Additional Topic on Neural Networks to examine the code of the MASS function `nnet()`, and here we will examine the code of the `sp` function `CRS()`. We start by typing the name of the function. Only the first line of output is displayed.

```
> CRS
function (projargs = NA_character_, doCheckCRSArgs = TRUE, SRS_string = NULL)
```

The function `CRS`() has three arguments, only the first of which is assigned a value in our use of it.

```
> projargs <- "+proj=utm +zone=10 +ellps=WGS84"
```

In Exercise 1 you are asked to manually follow the code of `CRS()` to the line at which the warning message is displayed. Here is that line.

```
> res <- rgdal::checkCRSArgs_ng(uprojargs = uprojargs,
+                     SRS_string = SRS_string)
Warning message:
In showSRID(uprojargs, format = "PROJ", multiline = "NO") :
  Discarded datum Unknown based on WGS84 ellipsoid in CRS definition
```

This says that we need to follow the trail into the `rgdal` function `checkCRSArgs_ng()` (remember that specifying the package and the symbol `::` distinguishes between multiple polymorphic functions in case there are more than one currently active functions with the same name). In this case there is only one such function in use, and we can display it. Again only the first line of output is shown.

```
> checkCRSArgs_ng
function (uprojargs = NA_character_, SRS_string = NULL)
```

In Exercise 2 you are asked to manually follow the code of `checkCRSArgs_ng()` to the line at which the warning message is displayed. Here it is.

```
>         uprojargs1 <- showSRID(uprojargs, format = "PROJ",
+             multiline = "NO")
Warning message:
In showSRID(uprojargs, format = "PROJ", multiline = "NO") :
  Discarded datum Unknown based on WGS84 ellipsoid in CRS definition
```

This finally takes us to the function `showSRID()`, which is the function mentioned in the warning message. Unsurprisingly, in Exercise 3 you are asked to type `showSRID`, copy the resulting code into your editor, and trace the steps. The first relevant portion includes a call to the R function `try()`.

```
>             res <- try(.Call("P6_SRID_show", as.character(inSRID),
+                 as.character(format), as.character(multiline),
+                 in_format, as.integer(epsg), as.integer(out_format),
+                 PACKAGE = "rgdal"), silent = TRUE)
```

If you type `?try` you will see the self-explanatory statement "`try` is a wrapper to run an expression that might fail and allow the user's code to handle error-recovery." Similarly, typing `?.Call` gives us "Functions to pass **R** objects to compiled C/C++ code that has been loaded into **R**." In other words, it is here that the PROJ API is accessed. Let's look at three key arguments of the function `.Call()` and the returned value `res`.

```
> as.character(inSRID)
[1] "+proj=utm +zone=10 +ellps=WGS84"
> as.character(format)
[1] "FORMAT=PROJ"
> as.integer(epsg)
[1] NA
> res
[1] "+proj=utm +zone=10 +ellps=WGS84 +units=m +no_defs"
attr(,"towgs84")
[1] "" "" "" "" "" "" ""
attr(,"datum")
[1] "Unknown based on WGS84 ellipsoid"
attr(,"ellps")
[1] "WGS 84"
```

The returned object `res` contains the contents of the warning message as well as the appropriate ellipsoid. The value of `attr(,"towgs84")` contains no information. There are seven locations which, if the CRS were not already WGS 84, could contain the values needed to transform the CRS into WGS 84 (or any CRS). The value of `attr(,"datum")` contains the warning message because the proj4string parameter `+datum` is considered redundant and is therefore ignored. This is done specifically to generate the warning message. The actual message is assembled further on in the code.

```
>         if ((!is.null(attr(res, "ellps"))) && (nchar(attr(res,
+                 "ellps")) > 0L) && (length(grep("ellps|ELLIPSOID",
+                 c(res))) == 0L)) {
+                 if (length(grep("datum|DATUM", c(res))) ==
+                   0L) {
+                   msg <- paste0("Discarded ellps ", attr(res,
+                     "ellps"), " in CRS definition: ",
+                     c(res))
+ }
+ }
> msg
[1] "Discarded datum Unknown based on WGS84 ellipsoid in CRS definition"
```

In summary, the function takes advantage of the redundancy of specification of the datum to generate a warning message. Without failing to execute the code, the warning message alerts the user to the fact that changes have been made and that at some point in the future the code may not work. Why it this all necessary? If in the future the proj4string is completely deprecated, then R software using this will no longer work. Many people's programs, as well as the many R packages that depend on the `sp` package, would, if they are not altered, cease to function. This warning provides an alert without actually doing any damage.

Now that we know what is going on, we can see that it is very easy to avoid the warning. Indeed, we have already done it. We simply assign the UTM projection to the `sf` object before converting it to an `sp` object for use in `spplot()`.

```
> data.Set1.sf <- st_read("set1\\landcover.shp")
> st_crs(data.Set1.sf) <- 32610
> data.Set1.cover <- as(data.Set1.sf, "Spatial")
```

Spatial analysis in R is clearly moving from the use `sp` objects to `sf` objects, and in future Additional Topics I hope to discuss how to carry out some of the SDA graphics and analysis in this domain.

Finally, we must discuss a seemingly trivial issue. We mentioned in Section 2 that angular coordinates (ϕ,λ) on the ellipsoid are traditionally specified with latitude first, whereas the coordinates (x,y) on a map are traditionally displayed with the x coordinate, which corresponds to longitude, first. We saw that the epsg specification of the WGS 84 CRS is displayed with latitude first. There is another representation, due to the Open Geospatial Consortium, or OGC, which can also be used to create a WKT2 representation.

```
> st_crs("OGC:CRS84")
```

```
Coordinate Reference System:
  User input: OGC:CRS84
  wkt:
GEOGCRS["WGS 84",
    DATUM["World Geodetic System 1984",
        ELLIPSOID["WGS 84",6378137,298.257223563,
            LENGTHUNIT["metre",1]],
        ID["EPSG",6326]],
    PRIMEM["Greenwich",0,
        ANGLEUNIT["degree",0.0174532925199433],
        ID["EPSG",8901]],
    CS[ellipsoidal,2],
        AXIS["longitude",east,
            ORDER[1],
            ANGLEUNIT["degree",0.0174532925199433,
                ID["EPSG",9122]]],
        AXIS["latitude",north,
            ORDER[2],
            ANGLEUNIT["degree",0.0174532925199433,
                ID["EPSG",9122]]]]
```

This displays longitude first. The epsg specification is by far the most commonly used, but it is worthwhile to be aware of this issue lest it cause silly errors, and to be aware of the availability of an alternative in case it does.

6. Further Reading

R. Bivand and his colleagues have published articles explaining various aspects of the transition from PROJ4 to later versions, up to and including PROJ8. Bivand and Pebesma (2020) and Bivand (2020) are good places to begin to learn more about this. This site, published by the US Geological Survey, contains a good discussion of the geoid concept, and this site contains a more GDAL specific discussion. Finally, if you are interested in the history and philosophy of open-source software, this Curious Minds Podcast has an excellent two-part description.

7. Exercises

1) In R, type `CRS` to display the code of the function `CRS()`. Copy the code to your own editor, make the appropriate assignments to the arguments, and work your way to the point in the code at which the warning message is displayed.

2) Repeat Exercise 1 for the function `checkCRSArgs_ng()`.

3) Repeat Exercise 1 for the function `showSRID()`.

4) Consider the following code sequence:

```
> x <- st_crs(data.Set1.sf)
> x$epsg
[1] 32610
> x$proj4string
[1] "+proj=utm +zone=10 +datum=WGS84 +units=m +no_defs"
> str(x)
List of 2
 $ input: chr "EPSG:32610"
 $ wkt  : chr "PROJCRS[\"WGS 84 / UTM zone 10N\",\n    BASEGEOGCRS[\"WGS 
84\",\n        DATUM[\"World Geodetic System 1984\",\"| __truncated__
 - attr(*, "class")= chr "crs"
```

Normally if an R object is a data frame then we expect the function `str()` to show us its data fields (See SDA2 Section 2.4). That clearly doesn't happen here, however. Why not? Hint: Remember that all R operators are actually polymorphic functions. Type `?″$″` and then type `?Extract`.

8. References

Bivand, R.S. (2020). Progress in the R ecosystem for representing and handling spatial data. J. Geographical Systems. https://doi.org/10.1007/s10109-020-00336-0

Bivand, R.S. and E. Pebesma (2020). R spatial follows GDAL and PROJ development. R-spatial.

Butler, H. (2017). History of PROJ, history-of-proj4-foss4g2017-howard-butler.pdf

Gimond, M. (2020) Intro to GIS and Spatial Analysis. https://mgimond.github.io/Spatial/index.html

Lovelace, R, J. Nowosad, and J. Muenchow (2020). Geocomputation with R https://geocompr.robinlovelace.net/index.html.

9. Acknowledgement

I am grateful to Alex Mandel for some comments and corrections.