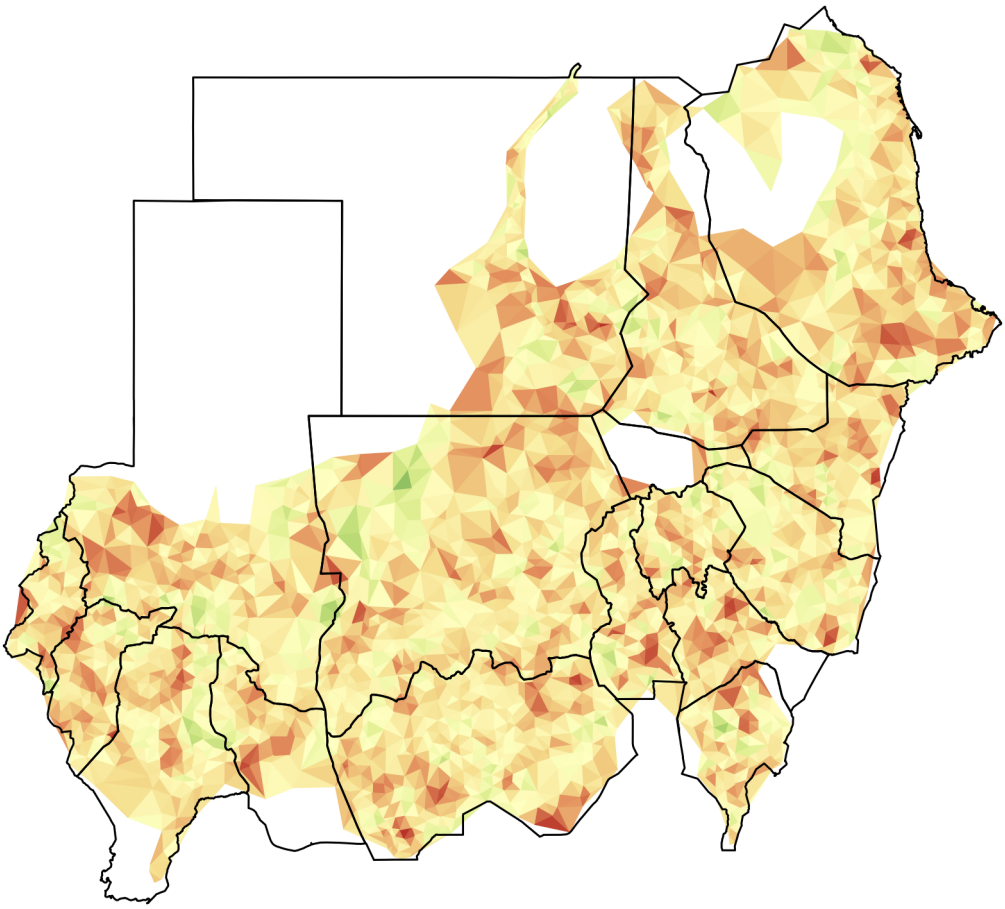
Short course on the use of *R* for the mapping requirements of *S3M*



Course designed and developed for the Sudan S3M National Survey
November 2013

Ernest Guevarra
Valid International

Exercise 1: Retrieving map data in *R*

In this exercise we will use ***R*** to read a shapefile dataset and get oriented with the structure and features of a shapefile dataset. The aim of the exercise is for you to become familiar with the use of ***R*** in handling shapefile datasets.

By this time, you have already learned how to issue a command to retrieve a standard or typical dataset using the **`read.table()`** function

For this exercise, we will use the **`readShapeSpatial()`** function provided by the **`maptools`** package to retrieve shapefile dateset.

First, we need to install and load the **`maptools`** package.

```
> install.packages("maptools")
> library(maptools)
```

We can now try to read the Sudan shapefile. To do this however, we need to have an orientation on what shapefiles are.

A shapefile is a digital vector storage format for storing geometric location and associated attribute information. This format lacks the capacity to store topological information. The shapefile format was initially developed for proprietary use with ArcView GIS version 2 in the early 1990s. It is now possible to read and write shapefiles using a variety of programs including data analysis software such as ***R***.

Shapefiles are simple because they store the primitive geometric data types of *points*, *lines*, and *polygons*. They are of limited use without any attributes to specify what they represent. Therefore, a table of records will store properties/attributes for each primitive shape in the shapefile. Shapes (points/lines/polygons) together with data attributes can create infinitely many representations about geographic data. Representation provides the ability for powerful and accurate computations.

While the term "shapefile" is quite common, a "shapefile" is actually a set of several files. Three individual files are mandatory to store the core data that comprise a shapefile:

`.shp`
`.shx`
`.dbf`

The actual shapefile relates specifically to **`.shp`** files but alone is incomplete for distribution, as the other supporting files are required.

With this knowledge of shapefiles, let us now take a look at the Sudan shapefiles dataset.

The Sudan shapefiles dataset contains the following shapefiles:

sudan01	**sudan01.shp**	Polygon shapefile of Sudan up to state administrative level
	sudan01.shx	
	sudan01.dbf	
	sudan01.prj	
	sudan01.qpj	
sudan02	**sudan02.shp**	Polygon shapefile of Sudan up to locality administrative level
	sudan02.shx	
	sudan02.dbf	
	sudan02.prj	
	sudan02.qpj	
grid12poly	**grid12poly.shp**	Polygon shapefile of rectangular grid at d = 12km
	grid12poly.shx	
	grid12poly.dbf	
	grid12poly.prj	
	grid12poly.qpj	
grid12kmSudan	**grid12kmSudan.shp**	Line shapefile of rectangular grid at d = 12km
	grid12kmSudan.shx	
	grid12kmSudan.dbf	
	grid12kmSudan.prj	
	grid12kmSudan.qpj	

We can now retrieve these shapefile datasets and create an object for each one.

```
> sudan01 <- readShapeSpatial("sudan01")
> proj4string(sudan01) <- "+proj=longlat +datum=WGS84"

> sudan02 <- readShapeSpatial("sudan02")
> proj4string(sudan02) <- "+proj=longlat +datum=WGS84"

> grid12poly <- readShapeSpatial("grid12poly")
> proj4string(grid12poly) <- "+proj=longlat +datum=WGS84"

> grid12kmSudan <- readShapeSpatial("grid12kmSudan")
> proj4string(grid12kmSudan) <- "+proj=longlat +datum=WGS84"
```

This series of commands illustrates key things about the way shapefile data can be read and handled in ***R***.

First is that the retrieval of shapefile datasets follows a very similar syntax as that of other standard datasets but just using the **`readShapeSpatial()`** function. The same principles apply including ensuring that you specify the corresponding directory in which your shapefiles are stored.

Second is the specification of a projection using the function **`proj4string()`**.

Map projection is important in mapping primarily because this sets the coordinates data in the shapefile dataset to the appropriate location on the surface of the earth. As there are many different projections, specifying one for the shapefile datasets that you are retrieving ensures that all geographic datasets that you are working on will be oriented correctly and located appropriately on the surface of the earth.

For the Sudan S3M, we are using the standard GPS projection using **longitude** and **latitude** coordinates with datum of **WGS84**. We use this primarily because this is the projection that is used in the coordinates data of our sampling points using the GPS locators.

Latitude lines run horizontally. Latitude lines are also known as parallels since they are parallel and are an equal distant from each other. Each degree of latitude is approximately 69 miles (111 km) apart. There is a variation due to the fact that the earth is not a perfect sphere but an oblate ellipsoid (slightly egg-shaped). Degrees latitude are numbered from 0° to 90° north and south. Zero degrees is the equator, the imaginary line which divides our planet into the northern and southern hemispheres. 90° north is the North Pole and 90° south is the South Pole.

The vertical **longitude** lines are also known as meridians. They converge at the poles and are widest at the equator (about 69 miles or 111 km apart). Zero degrees longitude is located at Greenwich, England (0°). The degrees continue 180° east and 180° west where they meet and form the International Date Line in the Pacific Ocean. Greenwich, the site of the British Royal Greenwich Observatory, was established as the site of the prime meridian by an international conference in 1884.

WGS stands for the **World Geodetic System** which is one of the many standards for use in cartography, geodesy, and navigation. The latest revision of this system is the **WGS84** which was established in 1984 and recently revised in 2004. **WGS84** is the reference coordinate system (CRS) used by the **Global Positioning System (GPS)**.

The string used in ***R*** to set this projection after a shapefile data has been retrieved is:

```
> proj4string(sudan01) <- "+proj=longlat +datum=WGS84"
```

This projection can also be specified when retrieving the shapefile data itself:

```
> sudan01 <- readShapeSpatial("sudan01", proj4string =
CRS("+proj=longlat +datum=WGS84"))
```

This command is basically instructing R to assign a projection of **longitude** and **latitude** with a datum of **WGS84** to the **`sudan01`** map object.

Now that we have stored the various shapefile data into objects, we can now explore and get ourselves oriented to the structure and features of a shapefile data object. We will do this using standard / basic functions in ***R*** that you have learned already in the previous training.

First, let us learn the class of a shapefile data object. We can find this out using the same function that you are familiar with already and have used previously the **`class()`** function.

```
> class(sudan01)
```

This command gives the following output:

```
[1] "SpatialPolygonsDataFrame"
attr(,"package")
[1] "sp"
```

This tells us that the **`sudan01`** object is of class **`SpatialPolygonsDataFrame`**. It also tells us that this is a special class specific to the **`sp`** package.

The **`sp`** package provides classes and methods for spatial data. The classes document where the spatial location information resides, for 2D or 3D data. Utility functions are provided, e.g. for plotting data as maps, spatial selection, as well as methods for retrieving coordinates, for subsetting, print, summary, etc.

If you check for the class of the other shapefile objects you've created, you will see that all of them are of the same **`SpatialPolygonsDataFrame`** class except for **`grid12kmSudan`**. Checking for the class of **`grid12kmSudan`** revealed the following:

```
> class(grid12kmSudan)

[1] "SpatialLinesDataFrame"
attr(,"package")
[1] "sp"
```

This tells us that the **`grid12kmSudan`** objects is of class **`SpatialLinesDataFrame`**.

You are now getting introduced to two of the most common shapes of a shapefile: **polygons** and **lines**.

A **polygon** consists of one or more rings. A ring is a connected sequence of four or more points that form a closed, non-self-intersecting loop. A polygon may contain multiple outer rings. The order of vertices or orientation for a ring indicates which side of the ring is the interior of the polygon. The neighbourhood to the right of an observer walking along the ring in vertex order is the neighbourhood inside the polygon. Vertices of rings defining holes in polygons are in a counterclockwise direction. Vertices for a single, ringed polygon are, therefore, always in clockwise order. The rings of a polygon are referred to as its parts.

A **line** is an ordered set of vertices that consists of one or more parts. A part is a connected sequence of two or more points. Parts may or may not be connected to one another. Parts may or may not intersect one another.

One of the other shapes that shapefiles take or represent is **points**.

A **point** consists of a pair of double-precision coordinates in the order x, y.

Because of this simple property of a point shapefile (i.e. a basic set of x and y coordinates), the use of shapefile format to store the point shape is not commonly used. The x and y coordinates for points can be contained or stored in other more basic formats such as **CSV**.

For example, the dataset that contains the x and y coordinates of all the known villages in Sudan is named **`settlementsSudan.csv`**. If we create an object called villages for this dataset

```
> villages <- read.csv("settlementsSudan.csv", header = TRUE,
  sep = ",")
```

and use the **`head()`** function to view the first 10 rows of this dataset

```
> head(villages, 10)
```

we get:

```
   ID      Village Pop     Source      State Locality        X        Y
1   1         Kosti          Georef White Nile    Kosti 32.66751 13.14846
2   2      Tandalti     Calculated White Nile    Kosti 31.86393 13.00969
3   3     Qawz kobi            GPS White Nile    Kosti 32.35000 13.60000
4   4     Karjuggle            GPS White Nile    Kosti 32.38333 13.46667
5   5    Idd maktuf            GPS White Nile    Kosti 32.28333 13.50000
6   6        Maryam            GPS White Nile    Kosti 32.55000 13.46667
7   7 Qawz nyaneir            GPS White Nile    Kosti 32.28333 13.60000
8   8        Salogi            GPS White Nile    Kosti 32.45000 13.15000
9   9       Sulayah            GPS White Nile    Kosti 32.25000 13.26667
10 10       Seleima     Calculated White Nile    Kosti 32.05833 13.00833
```

As you will notice here, the villages object contains information on the x and y coordinates of each of the villages in Sudan. So, whilst this dataset is not a shapefile (it is a basic data frame), it has information and a structure that is comparable to a point shapefile as defined above.

In the succeeding exercises, this similarity of a point shapefile and a standard data frame containing x and y coordinates of points will be further discussed and illuminated.

This knowledge on classes and shapes of shapefiles is an important learning particularly when performing functions to handle or manipulate different shapefile objects. The general principle is that functions or operations between two or more shapefile objects require these objects to be of the same class or family of classes. Also, the shapes defined by the shapefile object determine the way the shapefile data is structured which in turn determine how these objects can and should be handled or manipulated in ***R***. These principles will be further illuminated in the succeeding exercises.

After learning about the class of shapefile objects, we now learn about the structure of these objects. We are able to appreciate the structure of a shapefile object by using the function **str()**.

```
> str(sudan01)
```

The output of this command starts off with this line:

```
Formal class 'SpatialPolygonsDataFrame' [package "sp"] with 5
slots
```

which gives the class of the shapefile object and gives us an idea of the data structure as having 5 slots. This is one of the key differences of a **sp** data frame compared to a standard basic data frame. In a basic data frame, you basically have single data set organised in rows and columns similar to that of a table. In a sp data frame, you can think of it as a compound data frame in which each slot contains a specific dataset.

As you look further down into the structure of the shapefile object, you will notice the @ symbol recurring 5 times. This is the symbol used to retrieve the different slots of the shapefile objects.

The 5 slots in a **SpatialPolygonsDataFrame** are:

@data	Contains the index or reference data frame of the shapefile the number of rows of which indicates the number of polygons that comprise the entire shapefile
@polygons	Contains n number of datasets based on the number of polygons that comprise the entire shapefile.
@plotOrder	Contains an integer vector with a length equal to the number of polygons that comprise the entire shapefile and have values starting from 1 to n number of polygons in the entire shapefile. The order of the values of this vector determines which polygon is drawn first when plotting. This order is determined by decreasing area size of the polygons.
@bbox	Contains a matrix the values of which are the minimum and maximum x and y limits of the entire shapefile.
@proj4string	Contains a character string that specifies the projection and datum of the shapefile object

Now let us try to extract the different slots of **sudan01** object. Making a call for the data slot gives:

```
> sudan01@data
```

```
      Old_state      New_State STATE_1 STATEAR Source STATE_CODE ISO_CODE
0   West Darfur    West Darfur    <NA>    <NA>   <NA>       <NA>     <NA>
1   West Darfur    West Darfur    <NA>    <NA>   <NA>       <NA>     <NA>
2   West Darfur Central Darfur    <NA>    <NA>   <NA>       <NA>     <NA>
3  North Darfur   North Darfur    <NA>    <NA>   <NA>       <NA>     <NA>
4  South Darfur    East Darfur    <NA>    <NA>   <NA>       <NA>     <NA>
5  South Darfur   South Darfur    <NA>    <NA>   <NA>       <NA>     <NA>
6     Al Gezira           <NA>    <NA>    <NA>   <NA>       <NA>     <NA>
7     Blue Nile           <NA>    <NA>    <NA>   <NA>       <NA>     <NA>
8       Gedaref           <NA>    <NA>    <NA>   <NA>       <NA>     <NA>
9       Kassala           <NA>    <NA>    <NA>   <NA>       <NA>     <NA>
10     Khartoum           <NA>    <NA>    <NA>   <NA>       <NA>     <NA>
11         Nile           <NA>    <NA>    <NA>   <NA>       <NA>     <NA>
12     Northern           <NA>    <NA>    <NA>   <NA>       <NA>     <NA>
13   N Kordofan           <NA>    <NA>    <NA>   <NA>       <NA>     <NA>
14      Red Sea           <NA>    <NA>    <NA>   <NA>       <NA>     <NA>
15       Sennar           <NA>    <NA>    <NA>   <NA>       <NA>     <NA>
16   S Kordofan           <NA>    <NA>    <NA>   <NA>       <NA>     <NA>
17   White Nile           <NA>    <NA>    <NA>   <NA>       <NA>     <NA>
```

Here we note that the **sudan01** shapefile contains polygons of the each of the states of Sudan with **sudan01@data** specifying the names of each of these states.

To check for the number of polygons in **sudan01** shapefile (the answer to which will also be the number of states in Sudan), we can use the **nrow()** function as follows:

```
> nrow(sudan01@data)
```

which gives a result of

```
[1] 18
```

There are 18 polygons in **sudan01** shapefile which indicate that Sudan has about 18 states (if the shapefile used is up-to-date)

Let us now try extract the polygons slot of **sudan01**. If we make a call for the polygons slot as below:

```
> sudan01@polygons
```

we end up with a very long output which is not easy to review or appreciate. This is understandable because from the previous command extracting the slot data of **sudan01** we know already that the polygons slot will have 18 sets of polygon shapefile data each of which will have it's own datasets and data structure. This means that the output will be very long and not easy to manage.

In this instance, a review of the structure of **sudan01@polygons** may help in getting an insight as to how to handle or manage the datasets contained inside this slot. We call the **str()** function on **sudan01@polygons** as follows:

```
> str(sudan01@polygons)
```

The first thing we learn from the output of this command is that the **sudan01@polygons** is a **list** with a length of 18.

In this case, we can then use our knowledge of the **class list** to our advantage to be able to handle **sudan01@polygons**.

Lists are convenient class or mode of data because they can accommodate multiple objects within them and these objects don't have to be of the same mode / class or length. Lists are of single dimension each of which refer to a object or dataset that has been included into the list.

This means that we can use ***R***'s powerful subscripting (sometimes referred to as indexing) capabilities to access specific components of **sudan01@polygons**. We can try this by running the following command:

```
> sudan01@polygons[1]
```

In this command we are instructing ***R*** to give us the first element / component in the list **sudan01@polygons**. This gives us the dataset and hints on the structure of the first polygon in the list of 18 polygons of **sudan01** shapefile (see below).

```
[[1]]
An object of class "Polygons"
Slot "Polygons":
[[1]]
An object of class "Polygon"
Slot "labpt":
[1] 22.08469 13.78990

Slot "area":
[1] 5.690431e-11

Slot "hole":
[1] FALSE

Slot "ringDir":
[1] 1

Slot "coords":
          [,1]     [,2]
[1,] 22.08470 13.78990
[2,] 22.08470 13.78989
[3,] 22.08468 13.78989
[4,] 22.08469 13.78990
[5,] 22.08470 13.78990

Slot "plotOrder":
[1] 1

Slot "labpt":
[1] 22.08469 13.78990

Slot "ID":
[1] "0"

Slot "area":
[1] 5.690431e-11
```

If we want to view the dataset for the 11th polygon in the list, we use:

```
> sudan01@polygons[11]
```

If we want to view the dataset for the 6th and 7th polygon in the list, we use:

```
> sudan01@polygons[6:7]
```

or

```
> sudan01@polygons[c(6,7)]
```

If we want to view the dataset for the 3rd and the 16th polygon on the list, we use:

```
> sudan01@polygons[c(3,16)]
```

Let us now try to extract the **plotOrder** slot of **sudan01** shapefile. We make a call as follows:

```
> sudan01@plotOrder
```

which gives this result:

```
 [1] 13  4 14 15 17 12  6  9  5 10 16 18  8  3  7  2 11  1
```

The output is a numeric vector of length 18. The values in the vector represent the polygon that gets plotted first. In this case, the 13th polygon is the first to be plotted and the 1st polygon will be the last to be plotted.

The **plotOrder** slot is linked or related to the area slot of each of the polygon found in the dataset for each of the polygons in **sudan01** object. The polygon with the biggest area gets plotted first and the one with the smallest area gets plotted last.

Given this, it would be a good exercise of our newly learned skills of handling and manipulating mapping data to try to create a dataframe that combines the **sudan01@plotOrder** vector with the corresponding area size of each polygon referred to in the plotting order. This is a good way of checking whether indeed the plotting order of polygons is based on decreasing area size. We do this by:

```
> areaAll <- NULL

  for(i in 1:length(sudan01))
    {
    area <- sudan01@polygons[sudan@plotOrder][[i]]@Polygons[[1]]@area
    areaAll <- c(areaAll, area)
    }

  orderByArea <- data.frame(sudan01@plotOrder, areaAll)
```

The result of this is:

```
   sudan01.plotOrder       areaAll
1                 13 3.1366210e+01
2                  4 2.6736778e+01
3                 14 2.0095652e+01
4                 15 1.8573504e+01
5                 17 1.1580067e+01
6                 12 1.1146239e+01
7                  6 6.9050190e+00
8                  9 4.9850507e+00
9                  5 4.4367222e+00
10                10 4.1029046e+00
11                16 1.5250000e-03
12                18 3.1696143e+00
13                 8 3.1603177e+00
14                 3 2.7454324e+00
15                 7 2.2776232e+00
16                 2 1.8595014e+00
17                11 1.7906153e+00
18                 1 5.6904307e-11
```

Let us now take a look at the **bbox** slot of **sudan01**. We make a call as follows:

```
> sudan01@bbox
```

This gives the following results:

```
          min       max
x 21.8131148 38.590921
y  8.6411405 23.142892
```

This is basically telling us that the shapefile has a minimum x coordinate value of **21.8131148** and a maximum x coordinate value of **38.590921**. For the y coordinates, the shapefile has a minimum of **8.6411405** and a maximum of **23.142892**.

Another way of getting the minimum and maximum x and y coordinates of a shapefile is by using the **bbox()** function. It can be used as follows:

```
> sudan01Limits <- bbox(sudan01)
```

The object **sudan01Limits** is equivalent to **sudan01@bbox**.

This minimum and maximum x and y coordinates is for the entire shapefile.

Finally, let us take a look at the **proj4string** slot of **sudan01**. This can be retrieved by calling the following:

```
> sudan01@proj4string
```

The result of this command is:

```
CRS arguments: +proj=longlat +datum=WGS84
```

This indicates that the projection of the shapefile is longitude and latitude based on datum WGS84.

Another way of getting the projection is by using the **proj4string()** function as follows:

```
> proj4string(sudan01)
```

This gives the same result but with a slightly different format.

```
[1] "+proj=longlat +datum=WGS84"
```

Exercise 2: Plotting maps

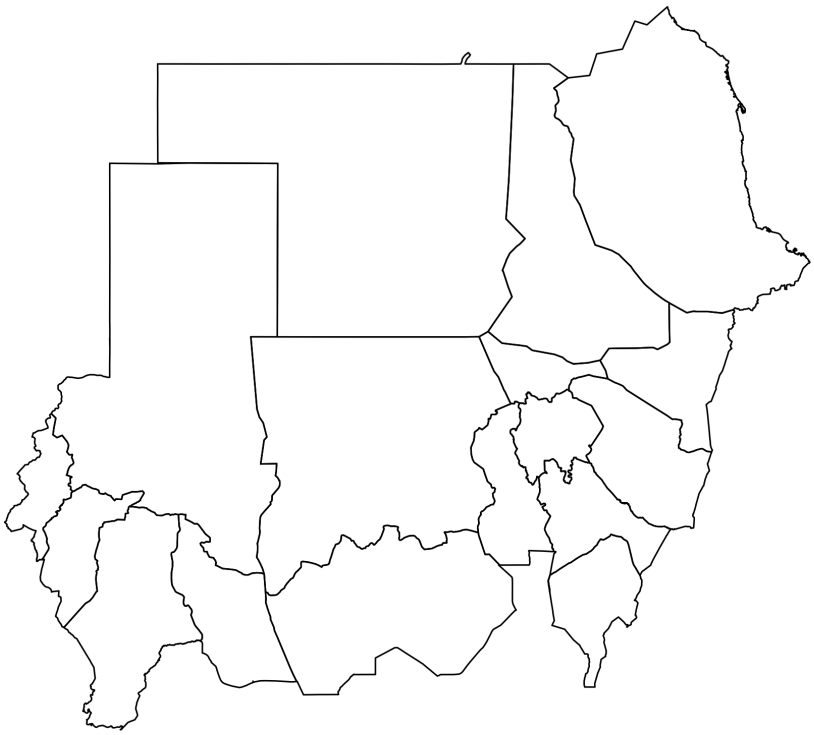
In this exercise, we will learn how to plot maps using your existing knowledge of basic plotting functions and new lessons on additional plotting functions specific to maps.

By now you would be familiar already with the **`plot()`** function which is the most basic of plotting functions. For maps, we use **`plot()`** in the same way as we would for other datasets. The main difference is that the methods used in the plotting functions for shapefiles are set by the **`sp`** package and is optimised for mapping purposes (i.e. no x and y-axis for plots).

Therefore, plotting maps is as easy as calling the **`plot()`** function and passing the shapefile object to it as shown below.

```
> plot(sudan01)
```

This produces the following map:



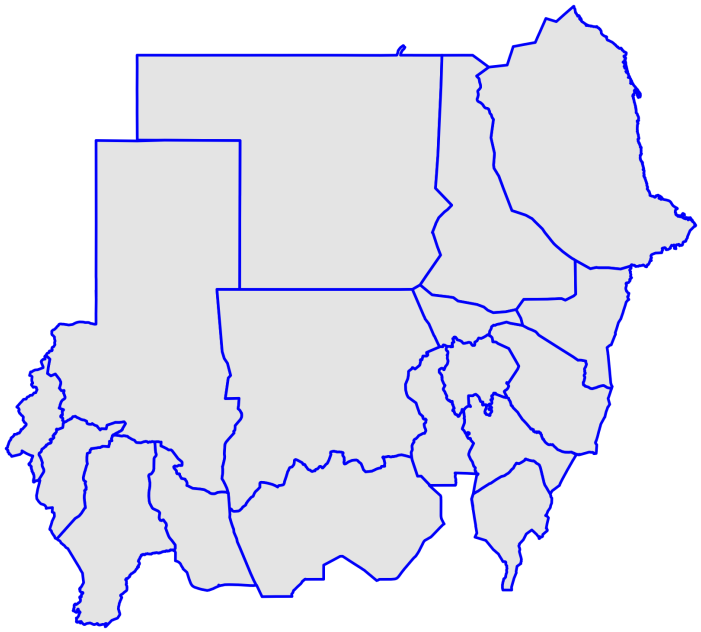
From this plot of **`sudan01`** shapefile object, we learn about some of the default plotting parameters under the **`sp`** package. Line type (**`lty`**) is set as a solid line (value of 1) and line width (**`lwd`**) is set at value 1. Border colour (**`border`**) is set as black while the fill colour (**`col`**) is set at none. We should remember that the class of **`sudn01`** shapefile object is a **`SpatialPolygonsDataFrame`**

hence it is composed of closed rings that are not holes (i.e. empty). Hence, in the plotting parameters, border and fill colours are specified separately.

Let us now try to plot another map of sudan01 shapefile object but this time change some of the default plotting parameters.

```
> plot(sudan01, lty = 1, lwd = 2, border = "blue", col =
  "gray90")
```

This command gives us the following map of Sudan:



The map above illustrates the changes in the plot created by the changes in the plotting parameters that we have specified. The width of the border is now thicker and its colour is now blue. The inside of the map is now coloured gray.

So far we have focused a lot on **`sudan01`** shapefile object. Hence we have learned a lot about the features, structure and characteristics of a polygon and a **`SpatialPolygonsDataFrame`** that contains polygons of each of the states of Sudan. However, we have yet to work with another **`SpatialPolygonsDataFrame`** object named **`sudan02`** which is a collection of polygons for each of the localities in Sudan.

What we will try to learn now is how to work with the **`sudan01`** and **`sudan02`** shapefile objects to create a map of Sudan that shows the different localities and also shows how these localities are grouped in order to make up each of the states in Sudan.

To be able to complete this task, we need to learn two simple mapping concepts / ideas. These are **map layers** and **map limits**.

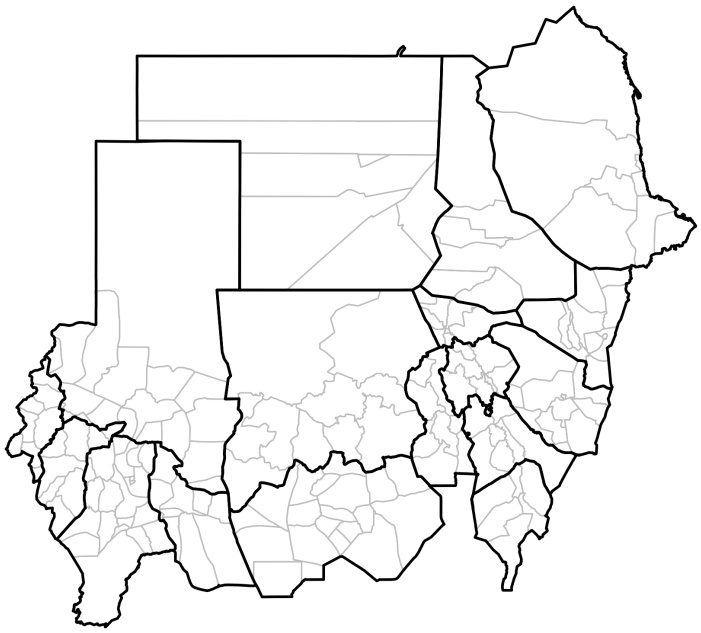
The idea of **map layers** is that different set of map features (i.e. roads, topography, watery systems) that usually have separate shapefile datasets are put together in a single map by plotting each of these features on top of each other in layers. Depending on the type and class of the shapefile objects being layered, the order by which a feature is layered on to another determines which features of which layer is visible or given emphasis.

The **map limits** are the lower and upper limits of the x and y coordinates of the shapefile object that is to be plotted. The other name for this is the **bounding box** (**`bbox`**) which we encountered already in the previous exercise when we were looking at the different slots of a **`SpatialPolygonsDataFrame`**.

Map limits are particularly important in relation to map layers because this ensures that the layered objects match and features are plotted on where they should be located.

Let us now try to layer **`sudan01`** and **`sudan02`** shapefile objects such that the base map is that of the localities and on top is that of the state boundaries. The final map will then show divisions by state and then further subdivisions per state by localities. We call the following commands to produce the map below:

```
> sudan02Limits <- bbox(sudan02)
> plot(sudan02, xlim = sudan02Limits[1,], ylim =
  sudan02Limits[2,], border = "gray")
> plot(sudan01, xlim = sudan02Limits[1,], ylim =
  sudan02Limits[2,], lwd = 2, add = TRUE)
```



Here we get introduced to two additional plotting parameters.

The x and y plotting limits are set using the **xlim** and **ylim** parameters. These arguments require a numeric vector with length of 2 specifying the lower and upper limits of the x and y plot axis. For mapping, this is the upper and lower limits of the x and y coordinates of the shapefile objects which is provided by the bounding box (**bbox**).

So, using our knowledge of the structure of the bounding box (**bbox**) object, we can extract the necessary data required by the **xlim** and **ylim** parameters.

```
> sudan01Limits <- bbox(sudan01)
> sudan01Limits

         min      max
x 21.813047 38.59093
y  8.642271 23.14286
```

The bounding box object **sudan01Limits** is a matrix the first row of which contains the lower and upper limits of the x coordinate and the second row contains the lower and upper limits of the y coordinate. We can therefore use subscripting or indexing as follows to extract the values needed for the **xlim** and **ylim** parameters.

```
> sidan01Limits[1,]

     min      max
21.81305 38.59093
```

This is what we used for the **xlim** parameter.

```
> sudan01Limits[2,]

     min       max
8.642271 23.142863
```

This is what we used for the **ylim** parameter.

The other plotting parameter we used is **add**. This argument basically instructs R to add the new plot onto the same graphic device that is already open. This parameter allows the addition of map layers.

You will note that we used the same **xlim** and **ylim** parameters when plotting both **sudan01** and **sudan02** shapefile objects. This ensures that both layers line up to each other correctly. This is generally good practice and should be observed.

However, this is not always needed particularly if the shapefile objects that we are layering have the same bounding box. In this case, **sudan01** and **sudan02** shapefile objects have the same bounding box as they are basically the same map but just with different administrative divisions.

In the next exercise, we will learn in which context the use of **xlim** and **ylim** is critical.

Now, let us learn how to add layers of other shapefile types / classes onto the base Sudan maps.

Earlier in **Exercise 1**, we have retrieved the dataset for the grid used in the Sudan S3M and assigned it to the object **grid12poly** and **grid12kmSudan**. The difference between these two objects is that the first one is a **SpatialPolygonsDataFrame** while the second is a **SpatialLinesDataFrame**.

We will focus our attention to **grid12kmSudan** to familiarise ourselves with the **SpatialLinesDataFrame** class objects. Let us look at the structure of a **SpatialLinesDataFrame**.

```
> str(grid12kmSudan)
```

The results of this command is as expected quite long. But looking at the first line of the output, we learn that **grid12kmSudan** has 4 slots.

```
Formal class 'SpatialLinesDataFrame' [package "sp"] with 4 slots
```

The 4 slots in a **SpatialLinesDataFrame** are:

@data	Contains the index or reference data frame of the shapefile the number of rows of which indicates the number of lines that comprise the entire shapefile
@lines	Contains n number of datasets based on the number of lines that comprise the entire shapefile.
@bbox	Contains a matrix the values of which are the minimum and maximum x and y limits of the entire shapefile.
@proj4string	Contains a character string that specifies the projection and datum of the shapefile object

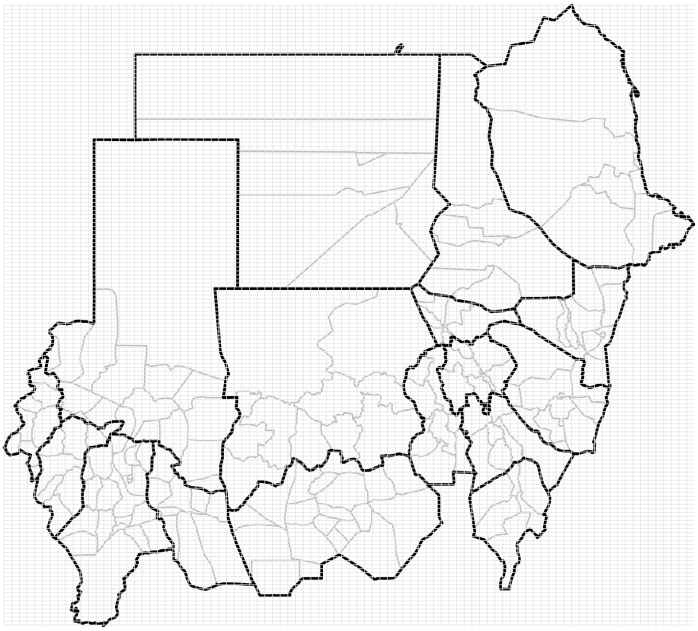
Compared to the **SpatialPolygonsDataFrame**, **grid12kmSudan** has a slot for lines (as expected) but doesn't have the **plotOrder** slot. If we look back at what we've learned regarding **plotOrder**, it is understandable why this is not included in a **SpatialLinesDataFrame** object. The **plotOrder** is determined by the area of the shapes in the object. However, **SpatialLinesDataFrame** objects does not have a value for area because it is just composed of lines, not polygons.

A closer look at the slot **grid12kmSudan@lines**, we notice that the data structure is much simpler than that of polygons. Each line is simply defined by a set of 2 points.

Let us now add the grid12kmSudan object as another layer on our map. We should put it on top of sudan02 and sudan01 with a thin line width (< 1) and a light colour. Here is the command that will produce this.

```
> sudan02Limits <- bbox(sudan02)
> plot(sudan02, xlim = sudan02Limits[1,], ylim =
  sudan02Limits[2,], border = "gray")
> plot(sudan01, xlim = sudan02Limits[1,], ylim =
  sudan02Limits[2,], lwd = 2, add = TRUE)
> plot(grid12kmSudan, xlim = sudan02Limits[1,], ylim =
  sudan02Limits[2,], lwd = 0.5, col = "gray90", add = TRUE)
```

This command produces the following map:



Now let us learn about **points** data.

Earlier, we have created an object called **`villages`**. This object contains village data for the whole of Sudan with their x and y geographical coordinates.

If we check the class of the object villages

```
> class(villages)
```

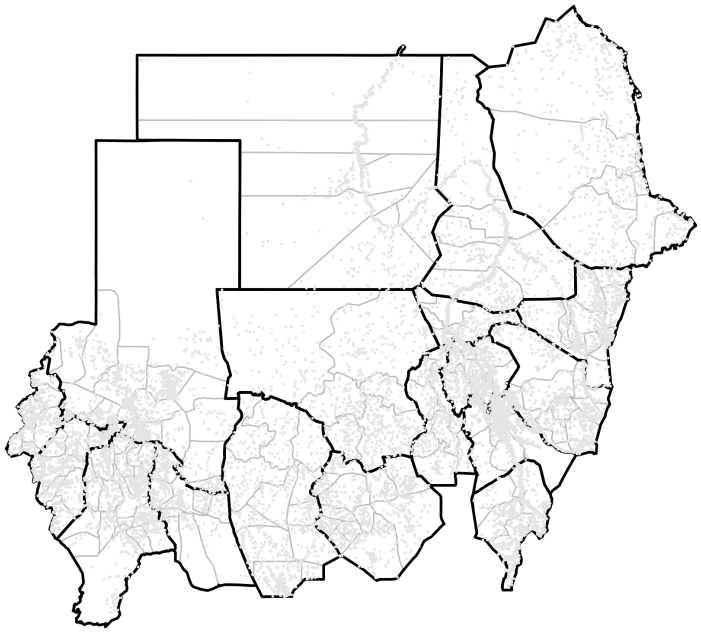
we find out that this object is a **`data.frame`** and not an **`sp`** class object.

However, this object is still map data because it contains geographic information which in this case is the x and y coordinates of the villages of Sudan.

This is the only information we need to be able to plot points data on the map. We can do this by calling the following commands:

```
> plot(sudan02, xlim = sudan02Limits[1,], ylim =
  sudan02Limits[2,], border = "gray")
> plot(sudan01, xlim = sudan02Limits[1,], ylim =
  sudan02Limits[2,], lwd = 2, add = TRUE)
> plot(grid12kmSudan, xlim = sudan02Limits[1,], ylim =
  sudan02Limits[2,], lwd = 0.5, col = "gray90", add = TRUE)
> points(villages$X, villages$Y, pch = 20, cex = 0.2, col =
  "gray90")
```

This produces the following map.



In the commands above, we learn a new function called **`points()`** and some new graphical parameters that are useful in plotting points.

The **`points()`** function basically instructs R to plot a series of points given the specified coordinates. The kind of character that is drawn to represent the point is determined by the parameter called **`pch`** (**p**oint **ch**aracter) with default being a hollow circle.

The most commonly used point characters are specified by the following numeric values:

In this case, we used **`pch = 20`** which is a solid point that is 2/3 the size of **`pch = 19`**.

The other graphical parameter that we learn here is cex (**c**haracter **ex**pansion) which determines the size of the point character and has a default value of **`cex = 1`**.

By now you would have noticed that when we used the **`points()`** function, we didn't have to add an argument to add the points on to the current graphical device. By default, the **`points()`** function will draw the points on to the current graphical device. In fact, **`points()`** will give an error message and will not plot anything if there is no open graphical device as you see below.

```
> points(villages$X, villages$Y, pch = 20, cex = 0.2, col =
  "gray90")

Error in plot.xy(xy.coords(x, y), type = type, ...) :
plot.new has not been called yet
```

Let us now learn how to add other map components onto the current map. These may include

labels	Either a numeric or character identifier that names the different features on the map
legend	Guide to what the shapes and symbols on the map refer to or mean
scale	The scale of a map is the ratio of a distance on the map to the corresponding distance on the ground.
compass	Indicates where the north, east, west and south directions are on the map

First, let us try adding labels to our map.

For **sudan01** object which contains a map of Sudan divided into states, we would like to be able to label each state accordingly. We would like this label to be placed at the centre of each state polygon as this is based on standard mapping conventions. To do this, we will use two functions that we have not learned yet. These are **coordinates()** and **text()**.

The **coordinates()** function is a function made available in the **sp** package. This function returns the centroid of the polygon or polygons in a map object. The **coordinates()** function returns a matrix containing the x and y coordinates of the centroid of the polygons in the map object. If there is only 1 polygon in the map object, then the resulting matrix is a single row.

The **text()** function on the other hand adds text into the current plotting device based on a specific x and y coordinates which for the purpose of labelling will be the centroid provided by the **coordinates()** function.

Additional parameters required for **text()** are the labels which is a character vector specifying the text to be written. In this case, we need the names of the states. We can easily get this from the data slot in **sudan01**. We use the parameter pos to specify where and how far the text will be written in relation to the x and y coordinates specified. The **pos** parameter can take values of 1, 2, 3, or 4 which places the text at the bottom, left, top or right of the centroid respectively. The **offset** parameter determines how far from the centroid the text will be written. The **cex** parameter sets the size of the text to be written.

We apply this function to label the states by giving the following commands:

```
> sudan01Centroids <- coordinates(sudan01)
> plot(sudan01)
> text(sudan01Centroids[,1], sudan01Centroids[,2], labels =
  sudan01@data$State, pos = 3, offset = 0.2, cex = 0.65)
```

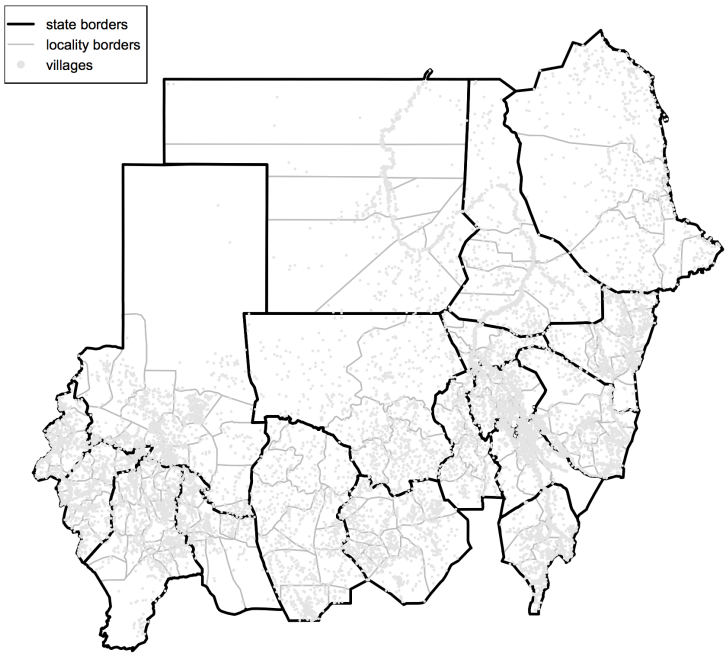
This results in the following map:

Now we shall add a legend on the map.

We use the **`legend()`** function for this. Following are the commands to create a legend for the map of **`sudan01`**, **`sudan02`** and villages.

```
> plot(sudan02, border = "gray")
> plot(sudan01, lwd = 2, add = TRUE)
> points(villages$X, villages$Y, pch = 20, cex = 0.2, col =
  "gray90")
> legend(topleft,
         y.intersp = 1.2,
         legend = c("state borders", "locality borders",
                    "villages"),
         lty = c(1, 1, NA),
         lwd = c(2, 1, NA),
         pch = c(NA, NA, 20),
         col = c("black", "gray", "gray90"),
         cex = 0.65,
         pt.cex = 1,
         bty = "o",
         bg = "white"
         )
```

This produces the following map:

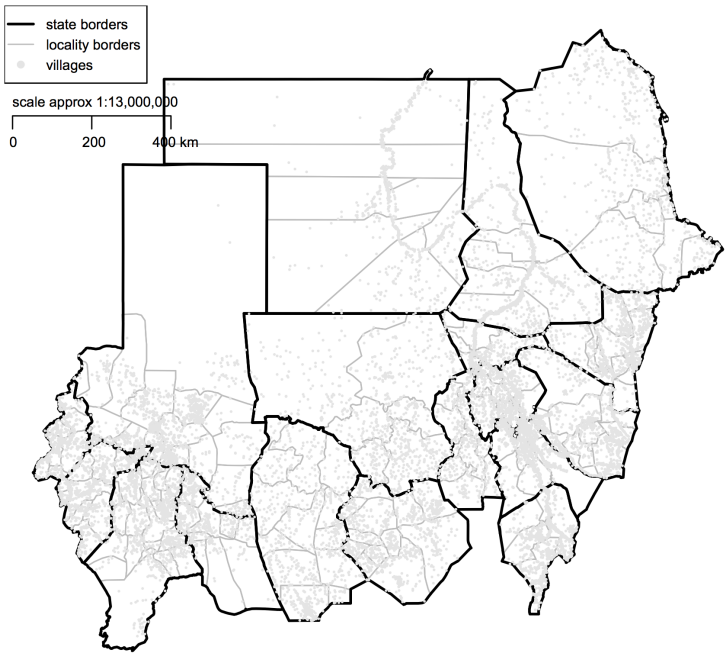


Lastly, we add a scale to our map.

We use the **map.scale()** function for this. We need to install and load the library **maps** to use this function. We add the scale on the map through the following commands:

```
> plot(sudan02, border = "gray")
> plot(sudan01, lwd = 2, add = TRUE)
> points(villages$X, villages$Y, pch = 20, cex = 0.2, col =
  "gray90")
> legend(topleft,
         y.intersp = 1.2,
         legend = c("state borders", "locality borders",
                    "villages"),
         lty = c(1, 1, NA),
         lwd = c(2, 1, NA),
         pch = c(NA, NA, 20),
         col = c("black", "gray", "gray90"),
         cex = 0.65,
         pt.cex = 1,
         bty = "o",
         bg = "white"
         )
> map.scale(x = sudan01Limits[1,], y = sudan01Limits[2,],
            ratio = TRUE, cex = 0.65)
```

This produces the following map:



Exercise 3: Manipulating shapefile data

In this exercise, we will use what we have learned about the structure and features of shapefile data in manipulating the data and creating new map objects.

By now you would have noticed that **sudan01** and **sudan02** shapefile objects contain data for the whole of Sudan. This is useful for creating maps for the whole country. However, there are times when we would like to create maps of the specific states only or assign colours for certain states differently rather than a single colour for all states.

To do these maps does not require new shapefiles for each of the state. All this needs is some manipulation of the full country map dataset. This is possible because the data structure of shapefiles contains all the data required to map all the features of the overall map separately.

First, let us try to create a map of **sudan01** and **sudan02** with the states and the localities coloured differently from each other.

A colour palette should be created which would provide colours for each of the states. This can be done using the following commands:

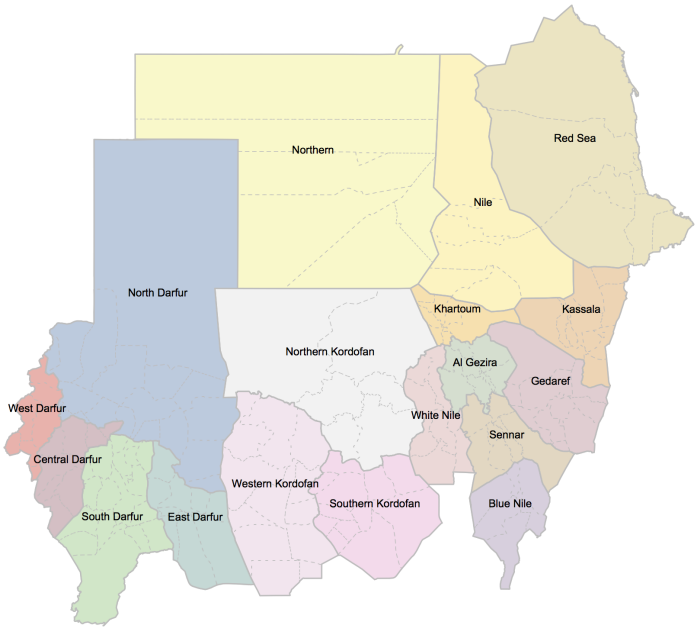
```
> createPalette <- colorRampPalette(colors = c("#FBB4AE",
  "#B3CDE3", "#CCEBC5", "#DECBE4", "#FED9A6", "#FFFFCC",
  "#E5D8BD", "#FDDAEC", "#F2F2F2"), space = "Lab")
> sudanCol <- createPalette(18)
```

What this command uses the **colorRampPalette()** function to create a colour palette of 18 unique colours (one for each state). These colours were generated from a base set of 9 colours which in turn was chosen using a qualitative colour scheme from **ColorBrewer 2.0** (see http://www.colorbrewer2.org).

Once we have a colour palette to use, we now plot the map and use the colour palette to specify the parameter **col**. This is done as follows:

```
> plot(sudan01, border = sudanCol, col = sudanCol)
> plot(sudan02, lwd = 0.5, lty = 3, border = "gray", add = TRUE)
> plot(sudan01, lwd = 1, border = "gray", add = TRUE)
> text(coordinates(sudan01)[,1], coordinates(sudan01)[,2],
  labels = sudan01@data$State, pos = 3, offset = 0.2, cex = 0.2)
```

The resulting map is:

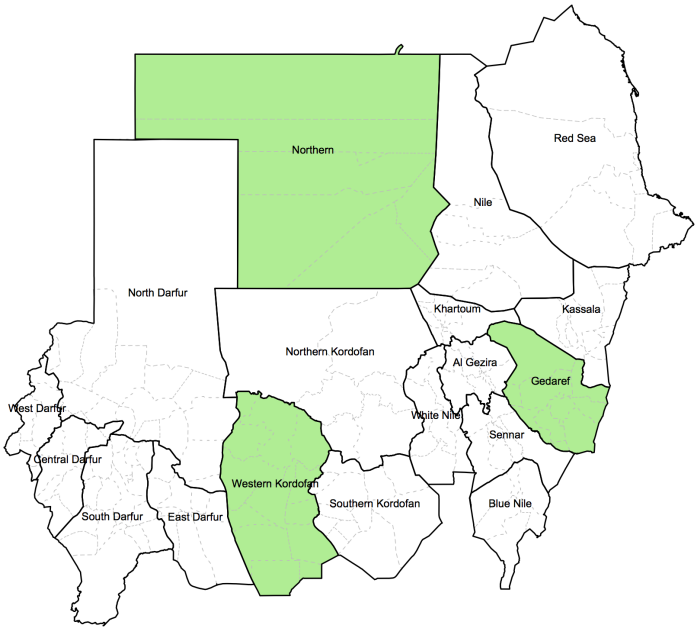


Now, let us try to specify a colour to just a handful of the states while the others remain without colour. This is particularly useful when you just want to highlight specific states in the map for presentation purpose or for emphasis.

For this map, we will need to use the **`sudan01@data`** slot to steer our selection of the states of interest which we can then specify a colour for. For example, if we want to show the full Sudan map and Gedaref, Northern and Western Kordofan coloured light green, we call the following commands:

```
> plot(sudan01, lty = 0, col = ifelse(sudan01@data$State %in%
  c("Gedaref", "Northern", "Western Kordofan"), "lightgreen",
  NA))
> plot(sudan02, lwd = 0.5, lty = 3, border = "gray", add = TRUE)
> plot(sudan01, lwd = 1, add = TRUE)
> text(coordinates(sudan01)[,1], coordinates(sudan01)[,2],
  labels = sudan01@data$State, pos = 3, offset = 0.2, cex = 0.5)
```

The resulting map is:



There are practical uses for knowing how to manipulate map data so as to change the colour of the components polygons either based on a qualitative criteria (such as different colours for each state to distinguish them from each other) or on numerical data (such as in creating a choropleth map).

The following example of simulated CMAM coverage results per state illustrates this usefullness.

We first simulate the data as follows:

```
> states   <- as.vector(sudan01@data$State)
> coverage <- c(95, 56, 73, 55, 19, 97, 60, 2, 46, 31, 70, 23,
                71, 66, 38, 78, 74, 51)

> exData        <- data.frame(states, coverage)
> names(exData) <- c("states", "coverage")
```

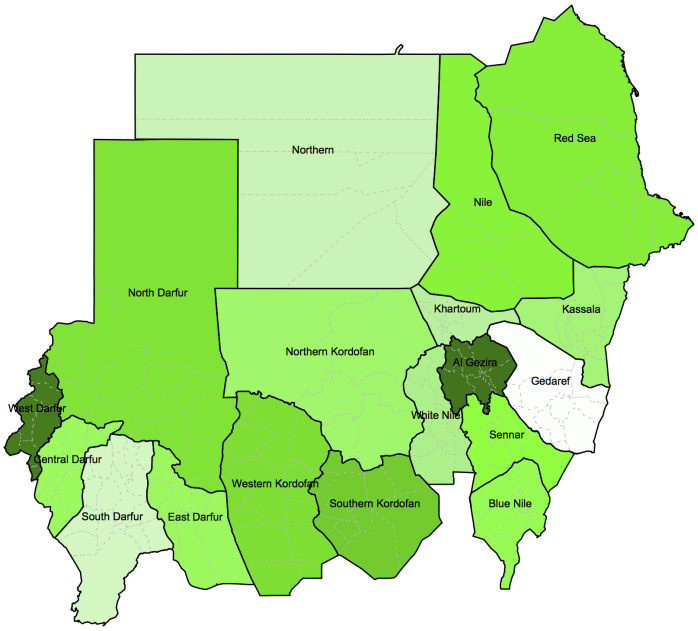
The exData object gives us the simulated of CMAM coverage by state. Then we specify a colour palette as follows:

```
> createMapPalette <- colorRampPalette(colors = c("white",
  "lightgreen", "green", "darkgreen"), space = "Lab")
> mapPalette <- createMapPalette(101)
```

We now can map this data using the folowing commands:

```
> plot(sudan01, lty = 0, col = mapPalette[exData$coverage])
> plot(sudan02, lwd = 0.5, lty = 3, border = "gray", add = TRUE)
> plot(sudan01, lwd = 1, add = TRUE)
> text(coordinates(sudan01)[,1], coordinates(sudan01)[,2],
  labels = sudan01@data$State, pos = 3, offset = 0.2, cex = 0.5)
```

This results in the following map:

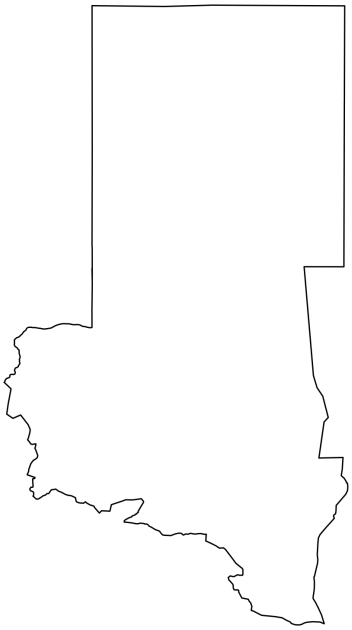


Finally, we learn how to create maps for each state separately using the sudan01 map dataset.

To begin with, let's try to create and map a map dataset for North Darfur. We call the following commands:

```
> nDarfur <- subset(sudan01, sudan01@data$State == "North
  Darfur")
> plot(nDarfur)
```

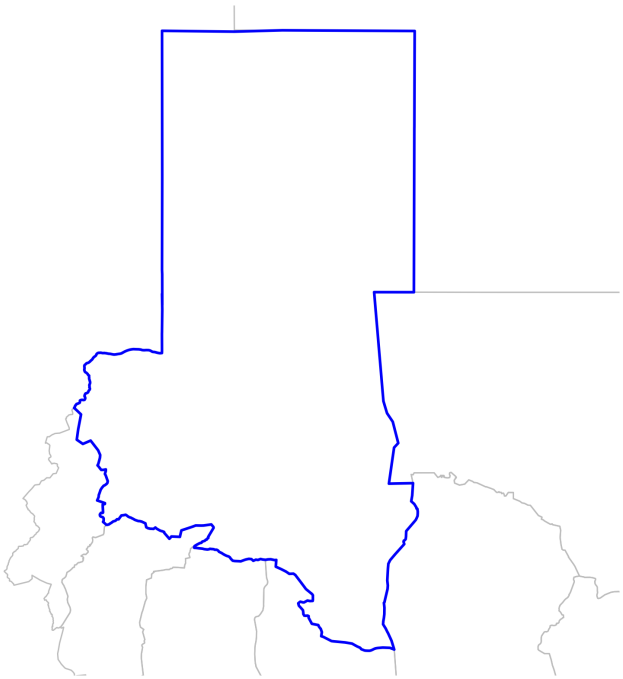
To produce the following:



However, this map will look better if we can also show the nearby states and their boundaries. We can do this by plotting **sudan01** but this time specifying a **xlim** and **ylim** parameters using the **bbox()** of the map object for North Darfur. The following commands illustrate this:

```
> nDarfur <- subset(sudan01, sudan01@data$State == "North
  Darfur")
> ndLimits <- bbox(nDarfur)
> plot(sudan01, xlim = ndLimits[1,], ylim = ndLimits[2,], border
  = "gray")
> plot(nDarfur, xlim = ndLimits[1,], ylim = ndLimits[2,], border
  = "blue", add = TRUE)
```

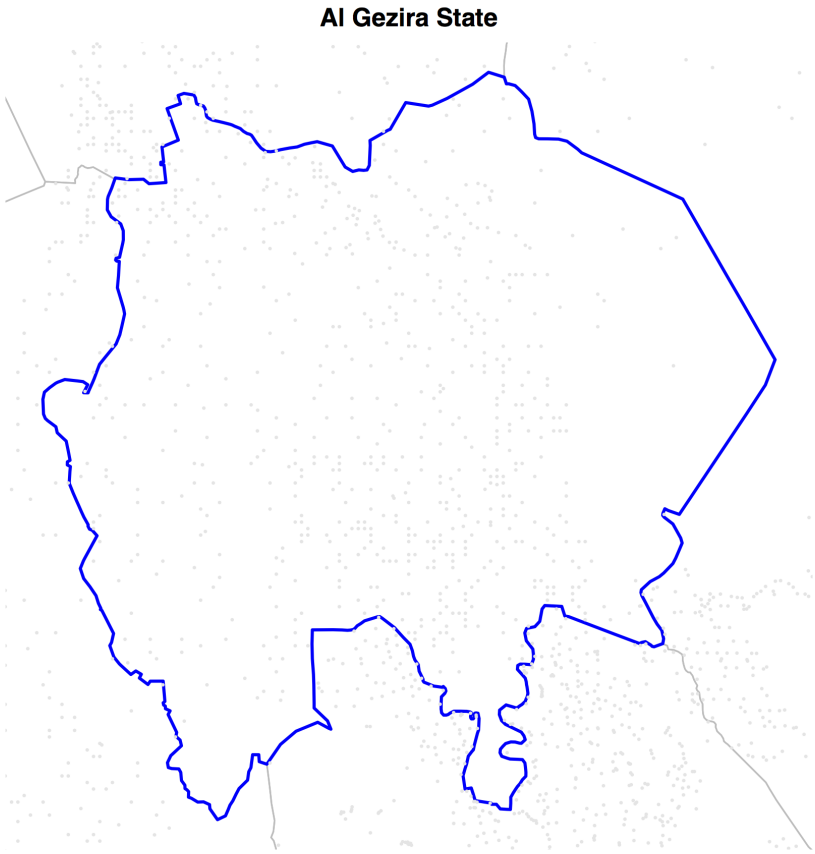
The resulting map is:



We can apply the same approach to get the map for each of the states. However, we can also create a looping function that will create and plot the maps for us automatically. This can be done as follows:

```
> stateList <- as.vector(sudan01@data$State)

> for(i in 1:nrow(sudan01@data))
    {
    a <- subset(sudan01, sudan01@data$State == stateList[i])
    aLimits <- bbox(a)

    x11(height = 6, width = 6, pointsize = 10)
    par(mar = c(2,2,2,2))

    plot(sudan01, xlim = aLimits[1,], ylim = aLimits[2,], border
    = "gray")
    plot(a, xlim = aLimits[1,], ylim = aLimits[2,], border =
    "blue", add = TRUE)

    points(villages$X, villages$Y, pch = 20, cex = 0.3, col =
    "gray90")

    title(main = stateList[i], cex = 2)
    }
```

The commands above will create a plot of the individual state maps on separate graphics device which can be saved one by one. An example of the maps is below.



The other approach is to plot each of the individual state map on the same graphics device. This can be done using the following commands:

```
> stateList <- as.vector(sudan01@data$State)

> x11(height = 18, width = 9, pointsize = 10)
> par(mar = c(2,2,2,2)); par(mfrow = c(6,3))

> for(i in 1:nrow(sudan01@data))
    {
    a <- subset(sudan01, sudan01@data$State == stateList[i])
    aLimits <- bbox(a)

    plot(sudan01, xlim = aLimits[1,], ylim = aLimits[2,], border
    = "gray")
    plot(a, xlim = aLimits[1,], ylim = aLimits[2,], lwd = 2,
    border = "blue", add = TRUE)

    points(villages$X, villages$Y, pch = 20, cex = 0.3, col =
    "gray90")

    title(main = paste(stateList[i], "State", sep = " "),
    cex = 2)
    }
```

The resulting map is: