Basic spatial R workshop - 1

Tobias Andermann

5/24/2020

Introduction to spatial data in R

(This tutorial is inspired by this very useful spatial R tutorial written by Robert Hijmans.)

Dependencies:

```r
library(sp)
library(raster)
library(sf)
```

If you are completely new to R, have a look at this general R introduction tutorial. It is very lengthy but well written and easy to follow. You don't have to do all of it, but try to understand the basic R syntax and once you feel like you get a hang of it, you can go back to this spatial tutorial. Take your time understanding the basics of R programming, that way the spatial tutorial will make a lot more sense for you as well. It's okay if you are behind on the general course pace, the main point of this course is for you to get the most out of it, and not to copy-paste commands you don't really understand the purpose of.

Before we jump into working with real spatial data, let's first learn a bit more about the basic types of objects we will be working with. In general one can decide between two differnt types of spatial data: **vectors and rasters**.

Vectors are used to represent discrete objects with clear boundaries, e.g. sampling locations, rivers, roads, country borders etc.

Rasters are applied to represent continuous phenomena, or "spatial fields", e.g. elevation, temperature, or species diversity.

1. Vector data

Let's first go through the basic types of vector data that are used in spatial analyses, which are points, lines, and polygons.

First we create some fake data. Let's pretend we are creating data for 10 taxa with the names `A-J`. Let's first just create this list of fake taxon names. You can use the `LETTERS` default array and extract the 10 first elements of it as shown in the command below. We'll assign this to the variable `name` which will now contain the first 10 letters of the alphabet, which are going to be our taxon names.

```r
# create taxon names
name <- LETTERS[1:10]
name
```

```
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J"
```

For each of these taxa we have a sampling location (`sampling_sites`) and a body size measurement in cm (`body_size`). To create coordinate data, we define one array with longitude and another with lattitude values of each point (made up data). You can use the `cbind()` command to pair them up into coordinate pairs, and we'll store these coordinate pairs as a new variable called `sampling_sites`. Print the content of the `sampling_sites` to the screen to understand what our fake coordinate data look like.

```r
# generate sampling locations
longitude <- c(-116.7, -120.4, -116.7, -113.5, -115.5,
               -120.8, -119.5, -113.7, -113.7, -110.7)
latitude <- c(45.3, 42.6, 38.9, 42.1, 35.7, 38.9,
              36.2, 39, 41.6, 36.9)
# this command simply combines the two arrays longitude and latitude into a shared matrix
sampling_sites <- cbind(longitude, latitude)

# define body sizes of sampled individuals
body_size = c(11,15,17,19,22,12,21,14,9,18)
```

A good way of dealing with all these different data arrays (names, longitude, latitude, boday size) is to join these arrays into one data frame in order to keep it together and sorted.

```r
# join data in a single dataframe
wst <- data.frame(longitude, latitude, name, body_size)
wst
```

```
##    longitude latitude name body_size
## 1     -116.7     45.3    A        11
## 2     -120.4     42.6    B        15
## 3     -116.7     38.9    C        17
## 4     -113.5     42.1    D        19
## 5     -115.5     35.7    E        22
## 6     -120.8     38.9    F        12
## 7     -119.5     36.2    G        21
## 8     -113.7     39.0    H        14
## 9     -113.7     41.6    I         9
## 10    -110.7     36.9    J        18
```

We won't be working with this dataframe in this tutorial, but it's good to know how to store your data this way. Also it's handy to know how to extract individual data columns from a dataframe. We can do that by using square brackets `[]` and the index of the column (or sets of columns) that we want to extract. Within the square brackets, you can specify the lines and columns to extract, uisng the following indexing syntax `[lines,columns]`. E.g. if we want to extract the first two columns (and all lines) we can do it like this (the `,` separates the indices for line and columns, in this case we leave the lines part blank, which will lead to all lines being extracted):

```r
wst[,1:2]
```

```
##    longitude latitude
## 1     -116.7     45.3
## 2     -120.4     42.6
## 3     -116.7     38.9
## 4     -113.5     42.1
## 5     -115.5     35.7
## 6     -120.8     38.9
## 7     -119.5     36.2
## 8     -113.7     39.0
## 9     -113.7     41.6
## 10    -110.7     36.9
```

Likewise if you want ot extract the first 3 lines and all columns for these lines you can type:

```r
wst[1:3,]
```

```
##   longitude latitude name body_size
## 1    -116.7     45.3    A        11
## 2    -120.4     42.6    B        15
## 3    -116.7     38.9    C         17
```

Alternatively, you can also extract columns by their name by using the $ sign. E.g. if you want to extract the body size columns you can extract if like this:

```
wst$body_size
```

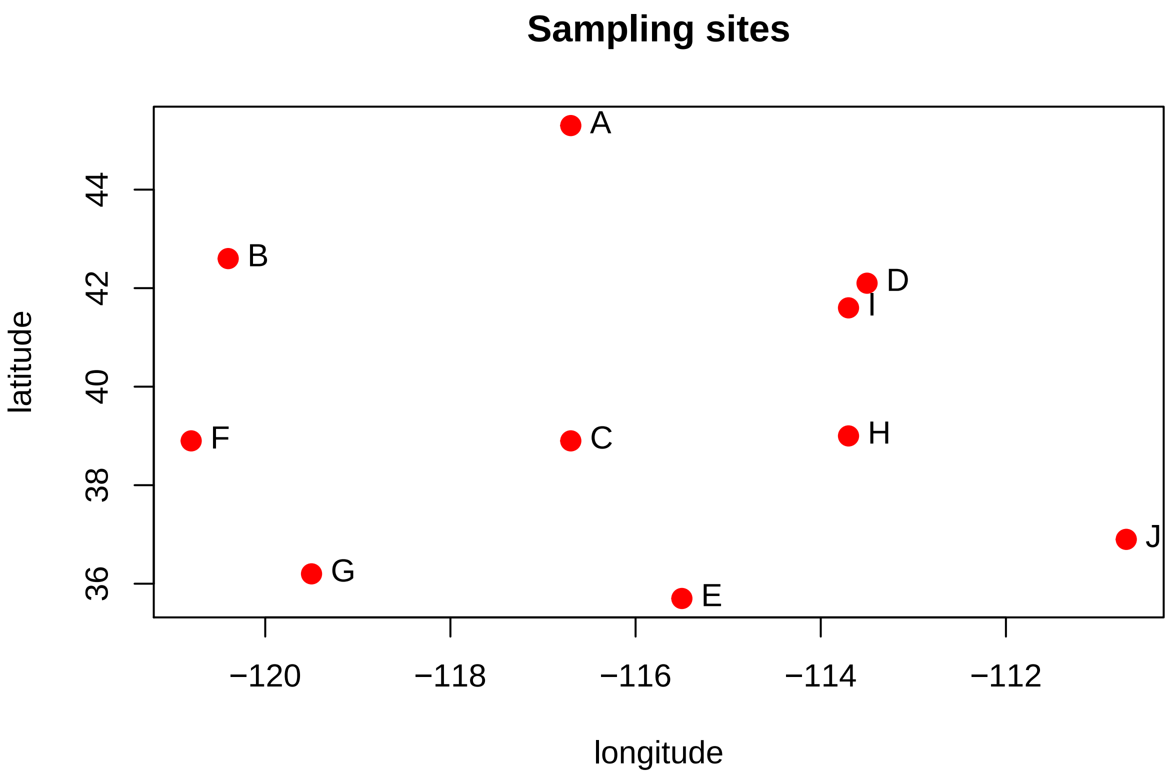
```
##  [1] 11 15 17 19 22 12 21 14  9 18
```

Points

Now let's plot the data **points** (coordinates) in a coordinate system. You plot in R by using the `plot()` function. The easiest way to plot our coordinate points is by using the `sampling_sites` object we created a little bit ago with the `cbind()` command. This object has the x and y coordinates (longitude and latitude) for each point already paired up and stored in the right format to plot. There are several settings you can use in the plot command, such as `cex` which alters the point size, `pch` which alters the shape of the points (see overview of available pch values here), and `col` where you can define the color of the points. Play around with these settings a bit if this is your first time plotting in R, it's good to know and fun to play with. The `main` argument in the plot command defines the title of the plot. You can add the names that we defined to each point in our `names` array by using the `text()` function as shown below.

General note on plotting: You can plot a variety of different types of objects in R using the `plot()` command. Everytime you call the `plot()` function it will overwrite the previous plot (unless you add `add=T` to the plotting command, in that case it will be added to the previous plot). In the following we will be using other plotting functions in combinations with the default `plot()`, such as e.g. the `text()`, `lines()`, `polygons()` functions. All of these functions will add information to existing plots without overwriting it, but they cannot be used alone without a preceeding `plot()` call.

```
plot(sampling_sites, cex=2, pch=20, col='red', main='Sampling sites')
# add names to plot
text(sampling_sites, name, pos=4)
```

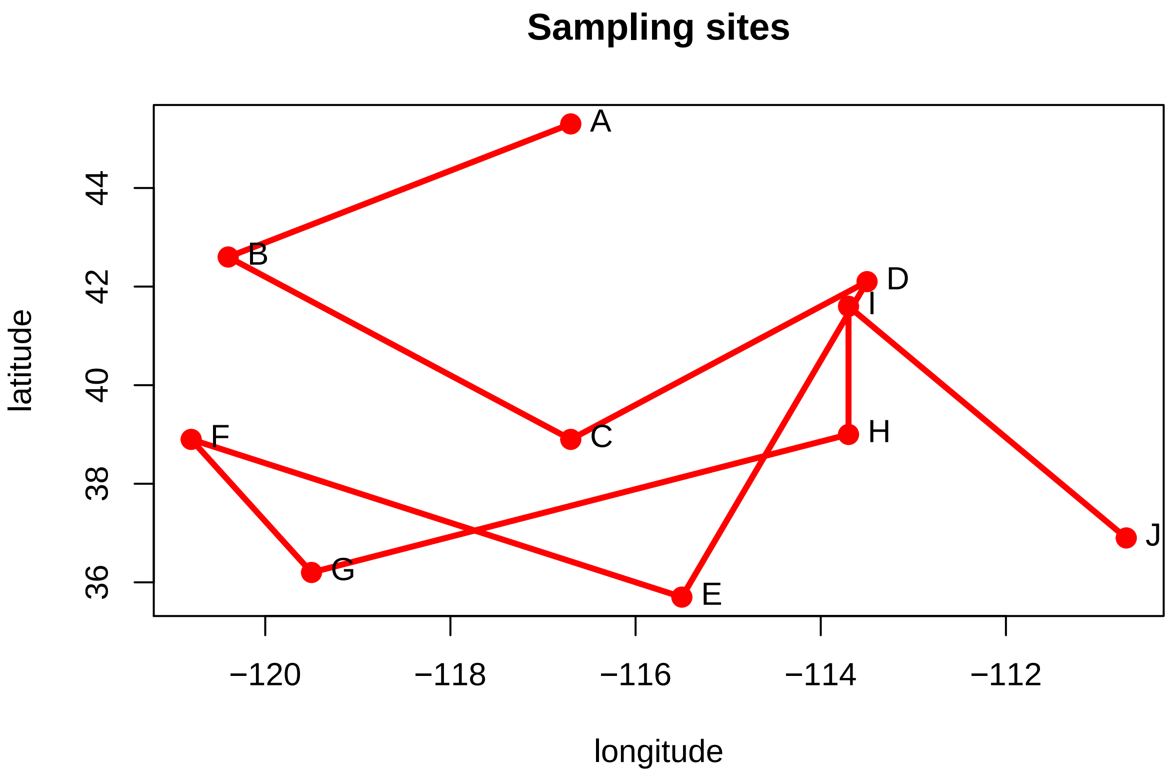


As you see in our plot we didn't plot a map, but instead a simple coordinate system. However the spatial relationships between the points are plotted in the correct ratios based on their latitude and longitude coordinates. In principle, plotting points on a map is just plotting in a regular coordinate system (as above) with a fancy background.

Lines

We can also plot **lines** connecting our dots from above, using the `lines()` function. Here we are only **plotting** our points as lines but later we will see how to define line objects, which besides the point coordinate information also carry the information in which order the points are connected. This can be used e.g. to visualize a time chronology of sampling.

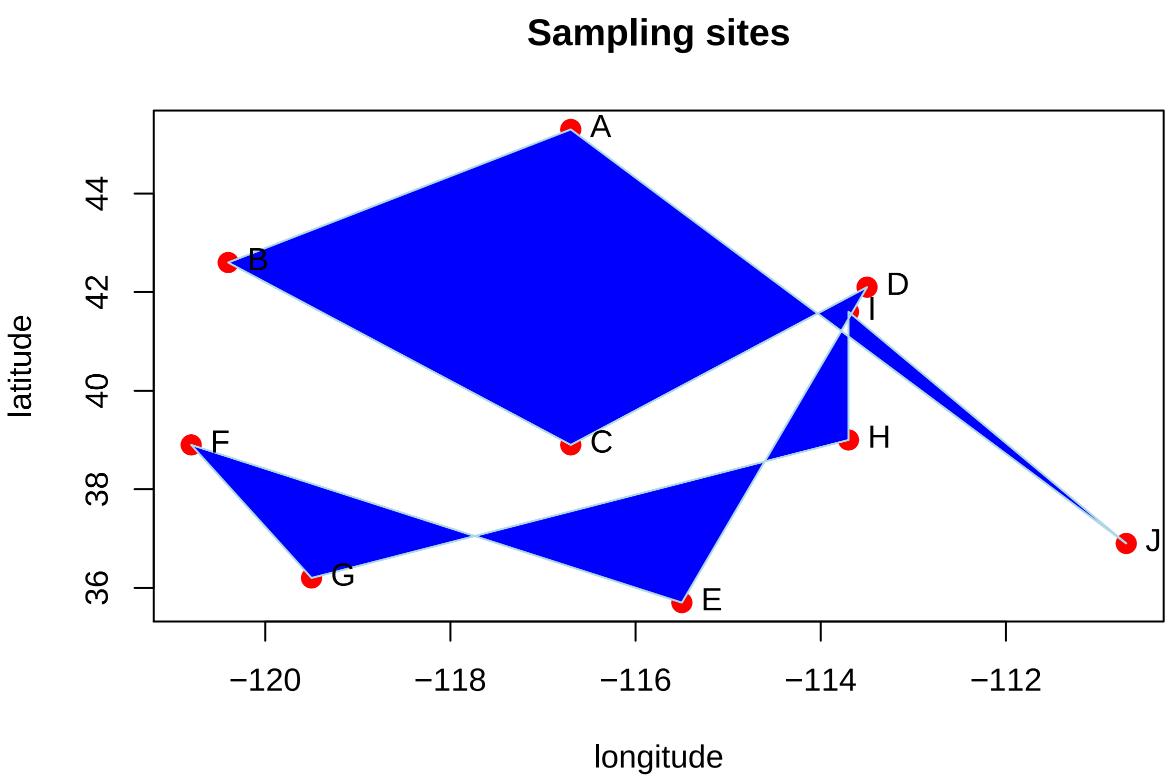
```
# first plot the points just as we did previously
plot(sampling_sites, cex=2, pch=20, col='red', main='Sampling sites')
# draw lines between data points
lines(sampling_sites, lwd=3, col='red')
text(sampling_sites, name, pos=4)
```



Polygons

We can also plot a **polygon** from the same data, using the `polygon()` function.

```
plot(sampling_sites, cex=2, pch=20, col='red', main='Sampling sites')
polygon(sampling_sites, col='blue', border='light blue')
text(sampling_sites, name, pos=4)
```



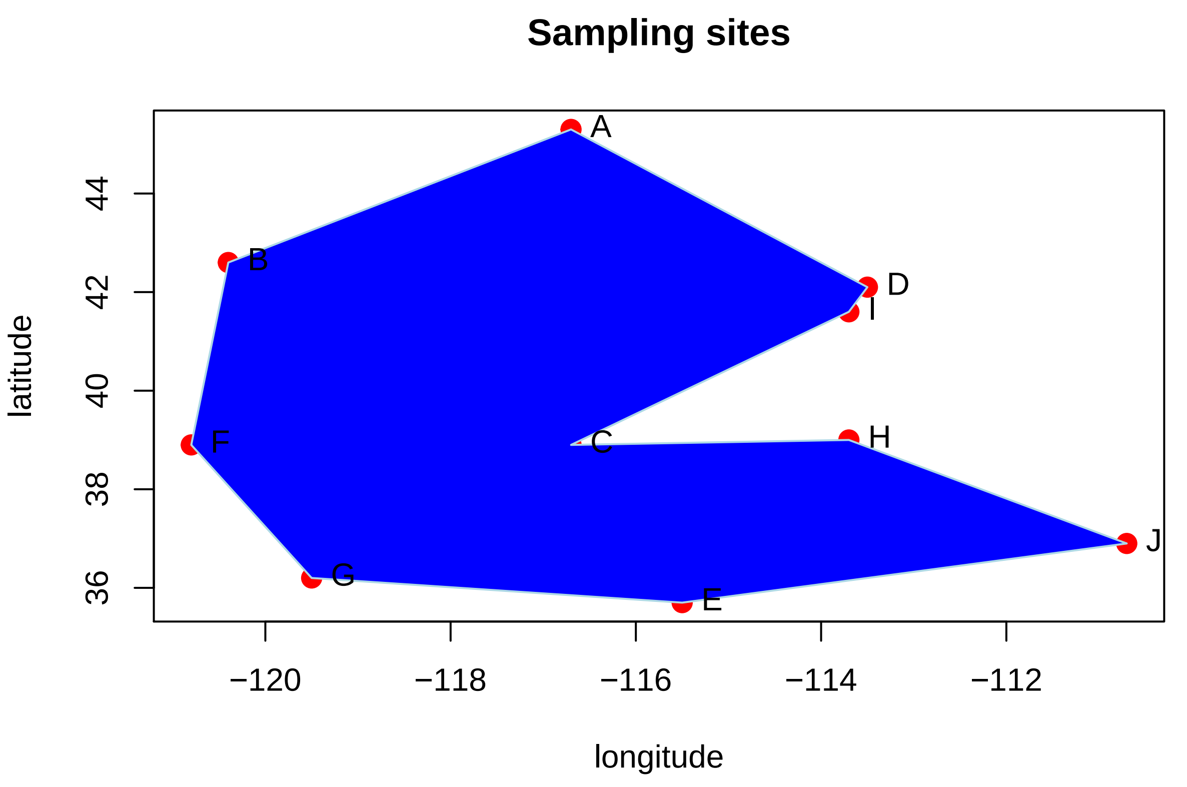
As you can see, also in case of polygons it matters in which order the points are stored. Let's reorder our data points and see how the polygon changes. For reordering an array we can just select the elements in the desired order by using their indices. For example for an array containing the 3 values 'a','b',and 'c' named `my_tiny_array` you can reorder the array having the 2nd element followed by the 3rd and then the 1st like this:

```
my_tiny_array = c('a','b','c')
my_tiny_array[c(2,3,1)]
```

```
## [1] "b" "c" "a"
```

Now we want to do the same for our coordinate array `sampling_sites`. Since this is a 2D array we need to specify rows and columns when selecting indices. Since we want to extract both columns and only reorder the lines in our `sampling_sites` array, we specify the new order of row indices followed by a comma.

```
reordered_sampling_sites = sampling_sites[ c(1,2,6,7,5,10,8,3,9,4), ]
plot(reordered_sampling_sites, cex=2, pch=20, col='red', main='Sampling sites')
polygon(reordered_sampling_sites, col='blue', border='light blue')
text(sampling_sites, name, pos=4)
```



Defining spatial objects

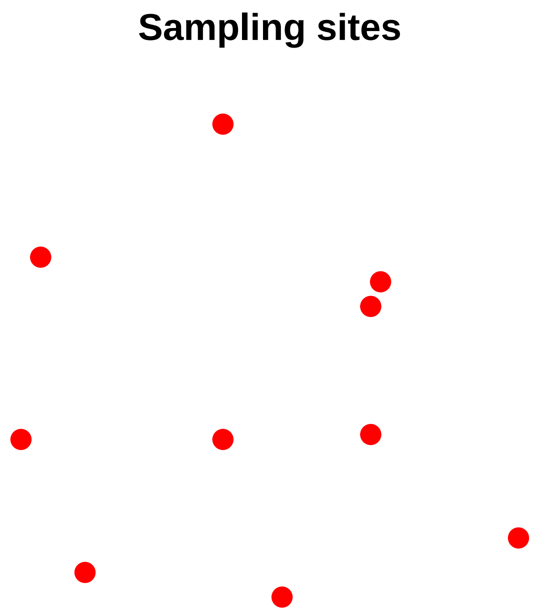
So far we have only worked with simple data points stored in arrays. However, different data object types exist that are specifically designed to handle spatial data. Some of the most commonly used objects are defined in the `sp` package. For vector data, the basic types are the `SpatialPoints()`, `SpatialLines()`, and `SpatialPolygons()`. These data objects only represent geometries. To also store attributes, additional data objects are available with these names plus 'DataFrame', for example, `SpatialPolygonsDataFrame()` and `SpatialPointsDataFrame()`. The advantage of this over e.g. just storing your coordinates in array form as we did so far, is that these data spatial objects have several useful functions, for example you can define the coordinate reference system of your coordinates in the spatial object, which makes it easy to later on transform your coordinates into another coordinate projection (you will see examples of that later on). Also most spatial functions you may be using to process or plot your data may require your data to be stored as spatial objects. Let's store our coordinates as a `SpatialPoints()` object and inspect the content and structure of this object with the `showDefault()` function.

```
library(sp)
pts <- SpatialPoints(sampling_sites)
# use the showDefault() function to view th content of the object (works for any type of object in R)
showDefault(pts)
```

```
## An object of class "SpatialPoints"
## Slot "coords":
##       longitude latitude
##  [1,]    -116.7     45.3
##  [2,]    -120.4     42.6
##  [3,]    -116.7     38.9
##  [4,]    -113.5     42.1
##  [5,]    -115.5     35.7
##  [6,]    -120.8     38.9
##  [7,]    -119.5     36.2
##  [8,]    -113.7     39.0
##  [9,]    -113.7     41.6
## [10,]    -110.7     36.9
##
## Slot "bbox":
##              min    max
## longitude -120.8 -110.7
## latitude    35.7   45.3
##
## Slot "proj4string":
## CRS arguments: NA
```

You see that besides the coordinate pairs we defined, this object contains information about the bounding box `bbox`, which defines the smallest rectangle containing all of your points. It also contains the currently empty slot `proj4string`, which is where you can store the coordinate reference system of your points, which we will do in a little bit. You will also notice when plotting the points that the plot will look a bit different than before (e.g. it's lacking the x and y axis). The reason for that is that the default plotting options for `SpatialPoints()` objects are different than those of simple numeric arrays (which makes sense because often when you want to plot points on a map, you may not wany the x-axis and y-axis to show).

```
plot(pts, cex=2, pch=20, col='red', main='Sampling sites')
```



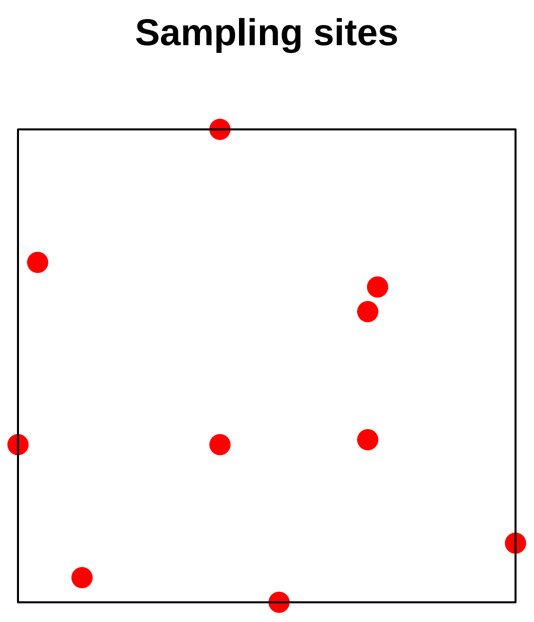
You can extract the additional values stored in the `SpatialPoints()` object, e.g. the coordinates of the bounding box, like this:

```
box = bbox(pts)
box
```

```
##              min    max
## longitude -120.8 -110.7
## latitude    35.7   45.3
```

Let's plot the bounding box with our coordinates to see what it looks like. You can use the `rect()` function to plot a simple rectangle given the x and y coordinates of two opposing corner points:

```
plot(pts, cex=2, pch=20, col='red', main='Sampling sites')
rect(box[1],box[2],box[3],box[4])
```



Let’s now use the SpatialPointsDataFrame() function to define a spatial object containing additional information about the samples (sample name and measured body size). For this we first store the data in a dataframe and then assign it to the SpatialPointsDataFrame() object together with the coordinates:

```
additional_data <- data.frame(sample_name=name, body_size=body_size)
ptsdf <- SpatialPointsDataFrame(pts, data=additional_data)
showDefault(ptsdf)
```

```
## An object of class "SpatialPointsDataFrame"
## Slot "data":
##    sample_name body_size
## 1            A        11
## 2            B        15
## 3            C        17
## 4            D        19
## 5            E        22
## 6            F        12
## 7            G        21
## 8            H        14
## 9            I         9
## 10           J        18
##
## Slot "coords.nrs":
## numeric(0)
##
## Slot "coords":
##       longitude latitude
##  [1,]    -116.7     45.3
##  [2,]    -120.4     42.6
##  [3,]    -116.7     38.9
##  [4,]    -113.5     42.1
##  [5,]    -115.5     35.7
##  [6,]    -120.8     38.9
##  [7,]    -119.5     36.2
##  [8,]    -113.7     39.0
##  [9,]    -113.7     41.6
## [10,]    -110.7     36.9
##
## Slot "bbox":
##              min    max
## longitude -120.8 -110.7
## latitude    35.7   45.3
##
## Slot "proj4string":
## CRS arguments: NA
```

Now we'll use a different function to create a spatial line object using the `spLines()` function. For this we first need to load the package `raster`. We'll also make up some new data, in order to have different data points than for the `SpatialPoints` object from above:

```
library(raster)
# make up new data
lon <- c(-116.8, -114.2, -112.9, -111.9, -114.2, -115.4, -117.7)
lat <- c(41.3, 42.9, 42.4, 39.8, 37.6, 38.3, 37.6)
lonlat <- cbind(lon, lat)
# store data as spatial lines object
lns <- spLines(lonlat)
lns
```

```
## class       : SpatialLines
## features    : 1
## extent      : -117.7, -111.9, 37.6, 42.9  (xmin, xmax, ymin, ymax)
## coord. ref. : NA
```

```
plot(lns)
points(lonlat)
```