**Packages**

In this post we will use the following packages:

| **Packages** | **Description** |
| --- | --- |
| tidyverse | Collection of packages (visualization, manipulation): ggplot2, dplyr, purrr, etc. |
| remotes | Installation from remote repositories |
| RQGIS3 | Interface between R and QGIS3 |
| sf | Simple Feature: import, export and manipulate vector data |
| ggtext | Improved text rendering support for ggplot2 |
| sysfonts | Load fonts in R |
| showtext | Use fonts more easily in R graphs |
| circular | Functions for working with circular data |
| geosphere | Spherical trigonometry for geographic applications |

In the case of the RQGIS3 package, it is necessary to install QGIS in OSGeo4W [here](https://www.qgis.org/es/site/forusers/download.html). I will explain the reason for using QGIS later.

# install the packages if necessary

if(!require("tidyverse")) install.packages("tidyverse")

if(!require("remotes")) install.packages("remotes")

if(!require("RQGIS3")) remotes::install\_github("jannes-m/RQGIS3")

if(!require("sf")) install.packages("sf")

if(!require("ggtext")) install.packages("ggtext")

if(!require("circular")) install.packages("circular")

if(!require("geosphere")) install.packages("geosphere")

if(!require("sysfonts")) install.packages("sysfonts")

if(!require("showtext")) install.packages("showtext")

# packages

library(sf)

library(tidyverse)

library(ggtext)

library(circular)

library(geosphere)

library(RQGIS3)

library(showtext)

library(sysfonts)

**Initial considerations**

Angles in vectorial lines are based on the angle between two vertices, and the number of vertices depends on the complexity, and therefore the resolution, of the vector data. Consequently, there can be differences in using different resolutions of a spatial line, either from the coast or from the river as in this example. A straight line is simply constructed with two points of longitude and latitude.

Related to this is fractality, an apparently irregular structure but that is repeated at different scales, known from coastlines or also from river. The most paradoxical feature is that the length of a coastline depends on the measurement scale, the smaller the measurement increment, the longer is the measured coastline.

There are two possibilities of obtaining the vertice angles. In the first one we calculate the angle between all consecutive vertices.

For example, imagine two points, Madrid (-3.71, 40.43) and Barcelona (2.14, 41.4).

What is the angle of a straight line between both cities?

bearingRhumb(c(-3.71, 40.43), c(2.14, 41.4))

## [1] 77.62391

We see that it is 77º, that is, northeast direction. But what if we go from Barcelona to Madrid?

bearingRhumb(c(2.14, 41.4), c(-3.71, 40.43))

## [1] 257.6239

The angle is different because we *move* from the northeast to the southwest. We can easily invert the direction to get the opposite angle.

# opposite angle of Barcelona -> Madrid

bearingRhumb(c(2.14, 41.4), c(-3.71, 40.43)) - 180

## [1] 77.62391

# opposite angle of Madrid -> Barcelona

bearingRhumb(c(-3.71, 40.43), c(2.14, 41.4)) + 180

## [1] 257.6239

The direction in which we calculate the angles is important. In the case of rivers, it is expected to be the direction of flow from origin to the mouth, however, a problem may be that the vertices, which build the lines, are not geographically ordered in the attribute table. Another problem may be that the vertices start at the mouth which would give the reverse angle as we have seen before.

However, there is an easier way. We can take advantage of the attributes of projected coordinate systems (Robinson projection, etc.) that include the angle between the vertices. We will use this last approach in this post. Still, we must pay close attention to the results as stated above.

**Preparation**

**Data**

Data attached on this repository..

**Import and project**

The first thing we do is to import, project the spatial lines and delete the third dimension *Z*, chaining the following functions: st\_read() helps us import any vector format, st\_zm() delete the dimension Z or M of a geometry and st\_transform() projects the vector data to the new projection in *proj4* format. All functions in the sf package start with st\_\* with reference to the spatial character, similar to *PostGIS*. In the same style as *PostGIS*, verbs are used as function names.

proj\_rob <- "+proj=robin +lon\_0=0 +x\_0=0 +y\_0=0 +ellps=WGS84 +datum=WGS84 +units=m no\_defs"

river\_line <- st\_read("RiverHRCenterlinesCombo.shp") %>%

st\_zm() %>%

st\_transform(proj\_rob)

## Reading layer `RiverHRCenterlinesCombo' from data source `C:\Users\xeo19\Documents\GitHub\blogR\_update\content\post\en\2020-07-24-river-flow-directions\RiverHRCenterlinesCombo.shp' using driver `ESRI Shapefile'

## Simple feature collection with 78 features and 6 fields

## geometry type: MULTILINESTRING

## dimension: XYZ

## bbox: xmin: -164.7059 ymin: -36.97094 xmax: 151.5931 ymax: 72.64474

## z\_range: zmin: 0 zmax: 0

## geographic CRS: WGS 84

**Extract the angles**

In the next step we have to extract the vertice angles. Unfortunately, as far as I know, it is not possible to extract the attributes with some function from the sf package. Although the function st\_coordinates() returns the coordinates, it does not include other attributes. Therefore, we must use another way, and that is the open software Quantum GIS in which we can find a tool to extract all the vertice attributes. We could import the vector data into QGIS Desktop and export the vertices from there, but it is also possible to access the QGIS tools from R directly.

For this, we need to have QGIS installed in OSGeo4W. The RQGIS3 package allows us to use very easily all the tools of the software in R. First we use the set\_env() function to define all the necessary QGIS paths and start the API with open\_app().

# paths to QGIS

set\_env()

## Trying to find QGIS in C:/OSGEO4~1

## $root

## [1] "C:/OSGeo4W64"

##

## $qgis\_prefix\_path

## [1] "C:/OSGeo4W64/apps/qgis"

##

## $python\_plugins

## [1] "C:/OSGeo4W64/apps/qgis/python/plugins"

##

## $platform

## [1] "Windows"

# start of QGIS Python

open\_app()

The find\_algorithms() function helps us to search for different QGIS tools. In addition the get\_usage() function specifies the way of usage with all the required parameters.

# search tools

find\_algorithms(search\_term = "vertices", name\_only = TRUE)

## [1] "native:extractspecificvertices"

## [2] "native:extractvertices"

## [3] "native:filterverticesbym"

## [4] "native:filterverticesbyz"

## [5] "native:removeduplicatevertices"

## [6] "saga:convertpolygonlineverticestopoints"

# usage of tool

get\_usage(alg = "native:extractvertices")

## Extract vertices (native:extractvertices)

##

## This algorithm takes a line or polygon layer and generates a point layer with points representing the vertices in the input lines or polygons. The attributes associated to each point are the same ones associated to the line or polygon that the point belongs to.

##

## Additional fields are added to the point indicating the vertex index (beginning at 0)

## the vertex’s part and its index within the part (as well as its ring for polygons)

## distance along original geometry and bisector angle of vertex for original geometry.

##

##

## ----------------

## Input parameters

## ----------------

##

## INPUT: Input layer

##

## Parameter type: QgsProcessingParameterFeatureSource

##

## Accepted data types:

## - str: layer ID

## - str: layer name

## - str: layer source

## - QgsProcessingFeatureSourceDefinition

## - QgsProperty

## - QgsVectorLayer

##

## OUTPUT: Vertices

##

## Parameter type: QgsProcessingParameterFeatureSink

##

## Accepted data types:

## - str: destination vector file

## e.g. d:/test.shp

## - str: memory: to store result in temporary memory layer

## - str: using vector provider ID prefix and destination URI

## e.g. postgres:… to store result in PostGIS table

## - QgsProcessingOutputLayerDefinition

## - QgsProperty

##

## ----------------

## Outputs

## ----------------

##

## OUTPUT:

## Vertices

In our case the tool to extract the vertices is simple and only has one input and one output. The function run\_qgis() executes a QGIS tool indicating the algorithm and its arguments. The advantage of using the algorithm directly from R is that we can pass objects of class sf (or sp) and raster that we have imported or created in R. As output we create a geojson, it could also be of another vector format, and we save it in a temporary folder. At the same time we indicate to import the result directly into R (load\_output = TRUE).

river\_vertices <- run\_qgis(alg = "native:extractvertices",

INPUT = river\_line,

OUTPUT = file.path(tempdir(), "rivers\_world\_vertices.geojson"),

load\_output = TRUE)

## $OUTPUT

## [1] "C:/Users/xeo19/AppData/Local/Temp/Rtmpa4DrCj/rivers\_world\_vertices.geojson"

Currently on Windows there seem to be problems with the *proj* library. In principle, if the function ends up creating the river\_vertices object, you should not worry.

**Selection**

Before continuing with the distribution estimation of the angles, we filter some rivers of interest. The functions of the tidyverse collection are compatible with the sf package.

river\_vertices <- filter(river\_vertices,

NAME %in% c("Mississippi", "Colorado",

"Amazon", "Nile", "Orange",

"Ganges", "Yangtze", "Danube",

"Mackenzie", "Lena", "Murray",

"Niger")

)

river\_vertices

## Simple feature collection with 94702 features and 11 fields

## geometry type: POINT

## dimension: XY

## bbox: xmin: -10377520 ymin: -3953778 xmax: 13124340 ymax: 7507359

## geographic CRS: WGS 84

## Warning in st\_is\_longlat(x): bounding box has potentially an invalid value range

## for longlat data

## Warning in st\_is\_longlat(x): bounding box has potentially an invalid value range

## for longlat data

## # A tibble: 94,702 x 12

## NAME SYSTEM name\_alt scalerank rivernum Length\_km vertex\_index vertex\_part

## \*

## 1 Nile 1 4 3344. 0 0

## 2 Nile 1 4 3344. 1 0

## 3 Nile 1 4 3344. 2 0

## 4 Nile 1 4 3344. 3 0

## 5 Nile 1 4 3344. 4 0

## 6 Nile 1 4 3344. 5 0

## 7 Nile 1 4 3344. 6 0

## 8 Nile 1 4 3344. 7 0

## 9 Nile 1 4 3344. 8 0

## 10 Nile 1 4 3344. 9 0

## # ... with 94,692 more rows, and 4 more variables: vertex\_part\_index ,

## # distance , angle , geometry

**Estimate the distribution**

To visualize the distribution we can use either a histogram or a density graph. But in the case of estimating the probability density function, we find a mathematical problem when applying it to circular data. For circular data we should not use the density() standard function of R since in our data a direction of 360º is the same at 0º, which would cause errors in this range of values. It is a general problem for different statistical metrics. More statistical details are explained in the circular package. This package allows you to define the characteristics of circular data (unit, data type, rotation, etc.) as an object class in R.

So what we do is to build a function that estimates the density and returns a table with the angles (x) and the density estimates (y). Since rivers have different lengths, and we want to see differences regardless of that, we normalize the estimates using the maximum value. Unlike the density() function, in which the smoothing bandwidth bw is optimized, here it is required to indicate it manually. It is similar to defining the bar width in a histogram.

dens\_circ <- function(x){

dens <- density.circular(circular(x$angle, units = "degrees"),

bw = 70, kernel = "vonmises",

control.circular = list(units = "degrees"))

df <- data.frame(x = dens$x, y = dens$y/max(dens$y))

return(df)

}

Finally, we estimate the density of each river in our selection. We use the split() function of R Base to get a table of each river in a list object. Then we apply our density estimation function to the list with the function map\_df() from the purrr package. The suffix \_df allows us to get a joined table, instead of a list with the results of each river. However, it is necessary to indicate the name of the column with the argument .id, which will contain the name of each river. Otherwise we would not know how to differentiate the results.

dens\_river <- split(river\_vertices, river\_vertices$NAME) %>%

map\_df(dens\_circ, .id = "river")

# results

head(dens\_river)

## river x y

## 1 Amazon 0.000000 0.2399907

## 2 Amazon 0.704501 0.2492548

## 3 Amazon 1.409002 0.2585758

## 4 Amazon 2.113503 0.2679779

## 5 Amazon 2.818004 0.2774859

## 6 Amazon 3.522505 0.2871232

**Visualization**

Now we only have to make the graph through the famous ggplot package. First we add a new font *Montserrat* for it use in this plot.

# font download

font\_add\_google("Montserrat", "Montserrat")

# use of showtext

showtext\_opts(dpi = 200)

showtext\_auto()

In the next step we create two objects with the title and the plot caption. In the title we are using an html code to color part of the text instead of a legend. You can use html very easily with the ggtext package.

# title with html

title <- "Relative distribution of river **flow direction** in the world"

caption <- "Based on data from Zeenatul Basher, 20180215"

The background grid that creates ggplot by default for polar coordinates did not convince me, so we create a table with x axis background lines.

grid\_x <- tibble(x = seq(0, 360 - 22.5, by = 22.5),

y = rep(0, 16),

xend = seq(0, 360 - 22.5, by = 22.5),

yend = rep(Inf, 16))

Next we define all the styles of the graph. The most important thing in this step is the element\_textbox() function of the ggtext package to be able to interpret our html code incorporated into the title.

theme\_polar <- theme\_minimal() +

theme(axis.title.y = element\_blank(),

axis.text.y = element\_blank(),

legend.title = element\_blank(),

plot.title = element\_textbox(family = "Montserrat",

hjust = 0.5,

colour = "white",

size = 15),

plot.caption = element\_text(family = "Montserrat",

colour = "white"),

axis.text.x = element\_text(family = "Montserrat",

colour = "white"),

strip.text = element\_text(family = "Montserrat",

colour = "white",

face = "bold"),

panel.background = element\_rect(fill = "black"),

plot.background = element\_rect(fill = "black"),

panel.grid = element\_blank()

)

Finally we build the graph: 1) We use the geom\_hline() function with different y intersection points to create the background grid. The geom\_segment() function creates the x grid. 2) We create the density area using the geom\_area() function. 3) In scale\_x\_continous() we define a negative lower limit so that it does not collapse at a small point. The labels of the eight main directions are indicated in the scale\_y\_continous() function, and 4) Finally, we change to a polar coordinate system and set the variable to create facets.

ggplot() +

geom\_hline(yintercept = c(0, .2, .6, .8, 1), colour = "white") +

geom\_segment(data = grid\_x ,

aes(x = x, y = y, xend = xend, yend = yend),

linetype = "dashed", col = "white") +

geom\_area(data = dens\_river,

aes(x = x, y = y, ymin = 0, ymax = y),

alpha = .7,

colour = NA,

show.legend = FALSE,

fill = "#011FFD") +

scale\_y\_continuous(limits = c(-.2, 1), expand = c(0, 0)) +

scale\_x\_continuous(limits = c(0, 360),

breaks = seq(0, 360 - 22.5, by = 22.5),

minor\_breaks = NULL,

labels = c("N", "", "NE", "", "E", "", "SE", "",

"S", "", "SW", "", "W", "", "NW", "")) +

coord\_polar() +

facet\_wrap(river ~ ., ncol = 4) +

labs(title = title, caption = caption, x = "") +

theme\_polar

## Warning: Ignoring unknown aesthetics: ymin, ymax