University of Exeter

GEOM184 - Open Source GIS

Practical 6

Interactive maps using R

27th February 2025

Welcome to the sixth and last practical for GEOM184 - Open Source GIS. Today we will continue working with R and focusing specifically on designing a ShinyApp that is representative of our large wood in rivers problem, and that we can effectively publish at the end of this session.

****Important**: we will re-use some of the data from practicals 1,2 and 5, including raster maps of heatmaps or other of the indicators that you have developed. Therefore, it is strongly recommended that each one of these is saved as an independent file, either as a GeoTIFF or a shapefile, since we will not repeat spatial analysis within the app development.**

Once again, today we will use RStudio.

There might be an element that you may be unsure on how to export as raster from last week, so you can quickly try the following:

```
dens <- MASS::kde2d(st_coordinates(lw_points)[,1], st_coordinates(lw_points)[,2], n = 100, h = c(0.01, 0.01))

# Convert to raster (old format)
r <- raster(dens) #note we need to use the 'raster' library

# Save as a GeoTIFF
writeRaster(r, "heatmap_raster.tif", overwrite = TRUE)
```

1 Part A - Introduction and creating the app structure

A ShinyApp can be defined by a single code, but since it merges different operations (e.g., definition of the user interface, back-end operations, etc.) it might be more appropriate to create a sub-set of codes that can be recalled within the main app code. A common approach is to create the main app code `App.R`, and then call within it three other separate scripts: textcolorpurple`Global.R` where all data are stored, textcolorpurple`Server.R` where any operation is performed, and textcolorpurple`UI` which defines how the user interface will look.
Therefore, within RStudio, create four scripts, each one of them renamed as per the above suggested nomenclature. @For consistency with last week, we will continue working using the Isonzo example. However, if you are working with either Exe or Torridge please do edit the code accordingly.

One thing to notice, is that we will work on this practical incrementally, i.e., we will build our app in the most basic format and then add information and functionalities at every step.

1.1 The main script - App.R

In this script we will start as we would do with any other R script: we input our key-information and we load any package and library that is required.

```
#################################################################
## Practical 6 for GEOM184 - Open Source GIS ##
## 27/02/2025 ##
## Creating a ShinyApp ##
```

```
## App.R ##
## code by Diego Panici (d.panici@exeter.ac.uk) ##
#####################################################################


# Load packages ----
library(shiny)
library(leaflet)
library(sf)
library(raster)
library(ggplot2)
library(ggiraph)
library(RColorBrewer)
library(terra)
library(leafem)


options(shiny.maxRequestSize = 1000 * 1024^2)
```

None of these lines should look un-familiar to you. Note the last line: we are telling Shiny to increase the maximum allowed size of files loaded within the app to 1000MB, normally capped at 5MB.

Task 1

Can you install the packages that are not currently on your R environment?

As a quick overview, `shiny` is the package by which we create the app; `leaflet` is used to arrange graphics and maps; `geojsonio` will be needed to represent some geographical features and their non-spatial attributes (for example, contours); `ggiraph` is used to create interactive graphs (another alternative would be `plotly`).

Next, we add the main app structure. This is extremely simple:

```
# Run global script containing all your relevant data ----
source("Global.R")

# Define UI for visualisation ----
source("UI.R")

ui <- navbarPage("Instream large wood on the River Isonzo", id = 'nav',
                 tabPanel("Map",
                          div(class="outer",
                              leafletOutput("map", height = "calc(100vh - 70px)")
                          )
                 )
)
```

```
# Define the server that performs all necessary operations ----
server <- function(input, output, session){
  source("Server.R", local = TRUE)
}
```

What we are observing here is that, first, we are fetching the data we need for our analysis and keep it in the `Global.R` script. We will cover that very shortly. Then, the UI line creates a Shiny user interface with a navigation bar titled *Instream large wood on the River Isonzo* whilst the next line adds a tab panel labelled *Map* to the UI and sources the UI code from the script `UI.R` that we created earlier (soon to be completed). Next, the server: it defines the server function that handles user inputs, processes data, and generates outputs (the input, output, session argument is standard in initialising your server). As before, the server function is sourced from the `Server.R` script.

The final step, is to provide a command to run the app! To do so, a very simple line is required:

```
# Run the application ----
shinyApp(ui, server)
```

However, at this stage we are not yet able to run the app. Rather, we keep this structure in place when the three other R scripts are completed.

1.2 The Global.R script

Here, we store all of our data, that we will then re-use within the server. So, let's begin populating this part, by first loading any data that we have at hand:

```
######################################################################
## Practical 6 for GEOM184 - Open Source GIS ##
## 27/02/2025 ##
## Creating a ShinyApp ##
## Global.R ##
## code by Diego Panici (d.panici@exeter.ac.uk) ##
######################################################################



# G1 Load large wood, river, and bridge data ----
lw_points <- st_read("LW_Isonzo.shp")
river <- st_read("RiverIsonzo.shp")
bridges <- st_read("Bridges_Isonzo.shp")

#Convert vectors to CRS 4326
lw_points <- st_transform(lw_points, crs = 4326)
river <- st_transform(river, crs = 4326)
bridges <- st_transform(bridges, crs = 4326)
```

Note the use of the header. Other than that, we have done the following simple steps:

1. Loaded the large wood, river and bridges vectors;

2. Reprojected on EPSG:4326 (this is the Shiny default, so we stick with it);

That's it. Nothing else to do here.

1.3 The UI.R script

It is now turn to define the user interface. This is also a relatively straightforward task. However, we first need to think how we want to display the interface. For our most basic task, we can think of a main map. Thus, let's do the following:

```
#####################################################################
## Practical 6 for GEOM184 - Open Source GIS ##
## 27/02/2025 ##
## Creating a ShinyApp ##
## UI.R ##
## code by Diego Panici (d.panici@exeter.ac.uk) ##
#####################################################################


div(class="outer",
    leafletOutput("map", height = "calc(100vh - 70px)")
)
```

`div(class="outer"` is a standard division container for UI, so we leave it like that. The lines with `leafletOutput` will create an outputs with a *Leaflet* map with an adaptive height based on the viewport's vertical height minus 70 pixels.

1.4 The Server.R script

We now need totell R/Shiny of the operations that we need to do. At this stage, we are just trying to visualise our map in an app, so our server function can look like:

```
#####################################################################
## Practical 6 for GEOM184 - Open Source GIS ##
## 27/02/2025 ##
## Creating a ShinyApp ##
## Server.R ##
## code by Diego Panici (d.panici@exeter.ac.uk) ##
#####################################################################

# S1 Render leaflet map ----
# Leaflet map output
  output$map <- renderLeaflet({

    leaflet() %>%

      setView(lng=13.533545, lat=45.850065, zoom=11.3) %>%
```

```
    addProviderTiles (providers$OpenStreetMap, group = "Colour")

  })
```

A couple of things to notice: the %>% is used whenever we require to include additional information or additional layers to a leaflet environment (as it's our main map) - think about it as the + for ggplot. Then, we use `output$map` for it to be called by the UI, as we defined a `map` output in the UI. We have also told Shiny to fetch the OpenStreetMap as a basemap. Also, note the `setView`: this is centred on the Isonzo area, so if you are focused on Torridge or Exe you will need to adjust it to the a point central to the catchment, and perhaps change the zoom level.

It would be interesting, at this stage, to just run the `App.R` script to see what the output may look like. Go to your app script, select all lines and hit `Ctrl+Enter` - do not click on `Run App`, as this is an ongoing issue with Shiny due to folder path definition, see more info here: here.

Task 2

If the app building has been successful, you should be able to see just a map centred on your area of interest. Check it captures the whole of your area, that you can zoom in and out, that the title is correct, that you are in the tab *Map*.

1.5 Adding data to the map

We now want to start visualising data to the map. In the first instance, let's just add river and bridges to the map. In the `Server.R` script, add after the provider tile the following lines, bearing in mind to append %>% at the end on the provide tile function:

```
    addPolylines(data = river, color = "blue", weight = 2, opacity = 0.8, group = "River") %>%
    addCircles(data = bridges,
               color = "black",
               fillColor="purple",
               fillOpacity=0.8,
               weight = 2,
               radius = 50,
               group = "Bridges")
```

The code is rather self-explanatory, but in principle we added the river shape as a line, and the bridges as circles, where we have customised colour, filling colour, transparency, etc. Exit the app (if you hadn't done it already), and run it again. Change any of the above customisable values if you want.

2 Part B - Include interactive elements

2.1 Clusters and bridges

We can now start including interactive elements so that we are not just seeing static lines and circles. The first thing to do, is to include large wood elements. We may want, for example, these large wood elements to pop up when clicked on and display key information, such as the type of large wood, the satellite imagery that was used to track it, and the clustering it belongs to. To do the following, first we add clusters to our `Global.R` script:

```
clusters <- st_read("IsonzoClusters.shp")
```

Task 3

Can you now re-project the cluster vector?

Lastly, within the same script we create a palette that will change colour according to the cluster number:

```
# Dynamically generate colours based on number of unique clusters
num_clusters <- length(unique(clusters$CLUSTER_ID))
pal_clusters <- colorFactor(palette = colorRampPalette(brewer.pal(12, "Paired"))(num_clusters), domain = clu
```

The above code does the following: first, generates a dynamic number of clusters (so that it will adapt to actual number within your dataset), and then creates a palette based on the number of clusters available.

Important: note that we used `CLUSTER_ID` as attribute of the `clusters` vector. If you have called this differently, then it needs to be edited.

Important: the following part assumes that the vector clusters contains all information that have been stored for large wood elements. If you have generated your clusters following steps in practicals 2 or 5, this should be the case, and in your attributes there should be `Imagery` and `Type`. If this is not the case, then it means they need to be added to the attribute table.

Next, we need to edit our server again. This time, as we want to be able to interact by clicking on the point, we need to add an *observer* which is outside `output$map`:

```
# Add popups for large wood points
  observe({
    leafletProxy("map") %>%
      clearMarkers() %>%
      addCircleMarkers(data = clusters,
                       fillColor = ~pal_clusters(CLUSTER_ID),
                       color = "black",
                       weight = 1,
                       radius = 5,
                       stroke = TRUE,
```

```
                    fillOpacity = 0.7,
                    popup = ~paste("<b>Type:</b>", Type,
                                   "<br><b>Imagery used:</b>", Imagery,
                                   "<br><b>Cluster ID:</b>", CLUSTER_ID),
                    group = "Large Wood")
})
```

There are a few things to consider here: first, the `observe` function dynamically updates the Leaflet map in response to changes in the Shiny app, continuously monitoring the reactive environment and executing the enclosed code whenever relevant data changes. Next, instead of recreating the map from scratch, `leafletProxy()` updates the existing map. Next, `clearMarkers()` remove any previously added markers. Finally, `addCircleMarkers`, uses the clusters dataset, then assigning colours to each marker based on their cluster ID, and other commands up to `fillOpacity` define the appearance. Interestingly, the next part is what our pop up will show when clicked: note that the text within quotation is what will appear in the pop up, and it uses HTML notation to express bold font `<b>`, and line break `<br>`. Other formatting is also available. The thing to notice here is that it recalls the information contained in each one of the attributes assigned to your cluster vector: if they have a different name, or are not there, then you will need to edit or remove it from the pop up.

Now save it, and run it again, and see how the circles can now be clicked on and a pop up appears.

Task 4

Can you now try and do the same for bridges so that when clicked on a pop up will show their name, although you want to keep their colour black (so do not include `fillColor`) and want to have a smaller radius? Hint: you may need to remove the bridges from the `output$map` part of the server.

2.2 Nearest distance lines

It is now time to add the nearest distance lines to our map. To do this, you just need to follow the same steps as before, making sure to update both `Global.R` and `Server.R`.

Task 5

Can you add the nearest distance between large wood and bridges to your script, as an additional element only (not an interactive element, unless you wish to)? Add the polylines as a black line and with 0.8 opacity. Remember to add the vector and transform the coordinates.

2.3 Add heatmap

Since the heatmap is a raster, the process to include this within Shiny is slightly different. First, we load the file in `Global.R`, reproject it, and set a palette for our heatmap:

```
heatmap <- rast("Heatmap.tif")
heatmap <- project(heatmap, crs(river))
```

```
pal_heatmap <- colorNumeric(palette = "inferno", domain = na.omit(values(heatmap)), na.color = "transpare
```

Note that in the code above we have set NA values as transparent.

Then, we edit the Server, as an additional line to the `output$map` function:

```
addRasterImage(heatmap, colors = pal_heatmap, opacity = 0.7, group = "Heatmap")
```

Now run the app, how is it looking?

2.4 Other rasters

We can now add as many other rasters as we like, including flow accumulation, slope, aspect.

Task 6

Add one by one the rasters that you have created so far and check how they look like and how informative they are.

2.5 Create an image query for rasters

An interesting feature is to create an image query for raster layers that provides a numerical value when you hover your cursor over it. This might be a helpful feature to visualise the values that you have generated. First, we need to do something counter-intuitive: we need to convert in our general script any of the rasters that we want to include with the image query using the `raster` package. So, for example:

```
heatmap <- raster(heatmap)
```

Now, in the server, we can add the following within the `output$map` instance:

```
addImageQuery(
      heatmap,
      layerId = "Heatmap",
      prefix = "Value: ",
      digits = 2,
      position = "topright",
      type = "mousemove", # Show values on mouse movement
      options = queryOptions(position = "topright"), # Remove the TRUE text
      group = "Heatmap"
    ) %>%
```

Note the pipe operator %>%. Now run the app, and check that by hovering on the heatmap you can see numerical values in the top-right.

We might want to go even a step further, and be able to control our layers and turn them on and off. In the server, we add the line:

```
addLayersControl(
      baseGroups = c("Colour"),
```

```
    overlayGroups = c("River", "Bridges", "Nearest distance", "Large Wood", "Heatmap"),
    options = layersControlOptions(collapsed = FALSE)
  )
```

This enables us to tick or untick any layer we want to exclude from our analysis.

Task 7

Can you now add all the other layers such as flow accumulation, slope, etc? Are they switchable on and off?

2.6 Add the large wood catchers

Task 8

Finally, can you add the large wood catchers as a vector (whether you designed them as lines, points, or polygons), include a pop up for text information, and make them controllable from the layer control?

3 Part C - Publish your app

This last step is relatively simple and ShinyApp can be published in different ways, see info here: `https://shiny.posit.co/r/getstarted/shiny-basics/lesson7/`. Whilst you can use any of the options (and if you have a website of yours, you are welcome to use it), the best option would be a GitHub upload - just be mindful of the maximum size limit (100MB), in which case you may need to resize your file (e.g., lower resolution resampling). You can also try and experiment by using Posit's own hosting server. For the purpose of your assessment, the GitHub option is required.

4 Conclusions

That's it.

this is the end of the practicals for GEOM184 Open Source GIS. By now, you should be proficient in using the GIS tools that we have been exploring in the last few weeks, for a well-rounded use and understanding. I hope you have enjoyed working with different tools and that this will be useful for your future career.

Good luck with your assessment!