Interactive visualisation of environmental data in decision support systems

BIO8068 Data visualisation in Ecology

# Introduction

You can use R to undertake many, but not all, of the most common GIS spatial analyses that you may encounter. This allows you to integrate your R code into shiny web-hosted applications, so that as well as simply displaying spatial data, you can start to develop something analogous to a **decision support system** or DSS. Originally DSS had to be hosted on dedicated computer clusters, and required their own unique interface. Now however they can be hosted on web-servers, making them more accessible. Key design decisions remain:

* Who is the end-user? Typically for a DSS the end-user is some sort of policy-maker, or decision-maker. Think back to your windfarm assignment in BIO8069.
* Who makes the final decision? The final decision as to where to place a windfarm will be made by elected councillors, advised by council planning officials, in consultation with environmental agencies.
* Who has access to the DSS? Council officials will receive reports and advice from private environmental consultancy companies, possibly hired by the windfarm developers. Representations will also be made by conservation bodies, local residents’ associations etc. Will they have access to the DSS?
* What data is in the DSS? Typically DSS contain a mixture of geographical, environmental, economic and social data. However, as these are drawn from different data sources they can be challenging to keep updated, or at the appropriate spatial scale.
* What will the DSS be able to model and predict? I have been involved in DSS where economists, ecologists and hydrologists all had to collaborate to construct the underlying models. Each discipline had its own assumptions, and unexpected incompatibilities, especially with regard to spatial and temporal scale, were constant challenges. All models are simplifications of reality. The DSS should be sufficiently constrained so as not to make predictions outside of its realistic limits of operation.
* How will DSS outputs be visualised? This is of critical importance. Graphical displays are easier to understand, but also easier to mis-interpret.

In this practical we will focus on some of the key geospatial tools you will find useful for building a DSS. Note that when you come to doing your assignment, you are definitely **not** expected to incorporate all these GIS tools into your simple DSS. Rather, it is to give you an understanding of some of the commonly used GIS analyses, such as those found in ArcGIS Toolbox, within R. In particular:

* raster vs vector conversion
* buffering
* clipping
* GIS raster overlay (equivalent to Raster Calculator)

Most of these steps are described very clearly in <https://geocompr.robinlovelace.net/index.html> Ironically, one key facility, common to all GIS (ArcGIS, QGIS, GRASS GIS) is not available in R, and that is Viewshed Analysis. I have therefore written a function for you to undertake this task, should you wish to implement it. To make it easier for you to understand the connections between GIS and creating a simple DSS, we’ll use the “Working with Elevation Data” practical from BIO8069 as a template. This has the advantage that you will be familiar with all the datasets, and it is simply understanding the implementation in R that matters. I have exported the relevant files for you in a format accessible by R. The following uses the same section and sub-section numbering and titles as in the BIO8069 practical, to make it easy for you to compare analyses.

# 1. Display of elevation data

You will display the following information

* **pan50m** - 50m resolution map of elevation in North Pennine hills
* **se82, se02** - Ordnance Survey backdrop rasters. We don’t need these but will use leaflet instead.
* **settlements** - vector (polygon) map of urban areas
* **wind\_turbines** - the viewshed function I have created is more limited than the one in ArcGIS, so we will adapt this map accordingly.

## 1.1 Display of elevation data - contours and shading

The original backdrop map was Ordnance Survey. For simplicity and speed in R, we can use the mapview package to visualise the data in RStudio, and it is easy to adapt these for leaflet if you want finer-grained control. Remember that mapview assumes your data will be in **latitude-longitude** format, EPSG 4326, whereas when working with your data to calculate distances etc. you will work with Ordnance Survey EPSG 27700. This is a bit irritating, as you will have to convert to latitude longitude for a **dynamic** display. If you are happy with a **static** display then Ordnance Survey is fine.

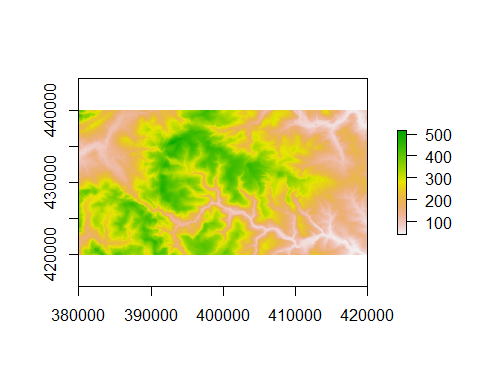
Begin by loading creating a new RStudio project for this practical. Within the project folder, create a subfolder to store the GIS data that you are going to import into R. I am calling mine gis\_data. The data are on Canvas as a zip file for download.

Load the relevant libraries as shown below. The rather weird options line is to suppress warnings about ellipsoid discarded or showSRID. These warnings have started to be displayed in the last 12 months due to upgrades to the geospatial system in R, which makes it more accurate, but not all packages are synchronised. However, it does not affect the functioning.

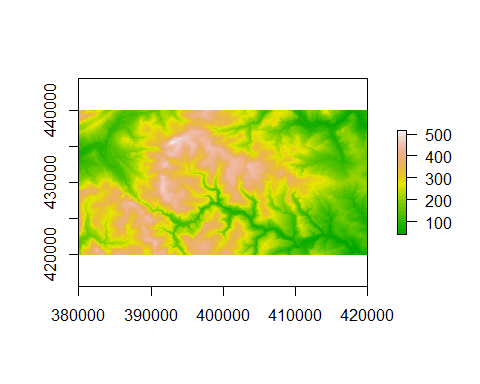
options("rgdal\_show\_exportToProj4\_warnings"="none")  
library(sf)  
library(raster)  
library(mapview)

If you receive errors that either library is not available, remember to install them with install.packages(). Now import and display the 50m resolution raster elevation map. This is available as a TIF format file (geotiff). Assuming your TIF format files are in gis\_data now load them into R:

# Import raster elevation data Ordnance Survey projection  
pan50m <- raster("gis\_data/pan50m.tif")  
plot(pan50m)

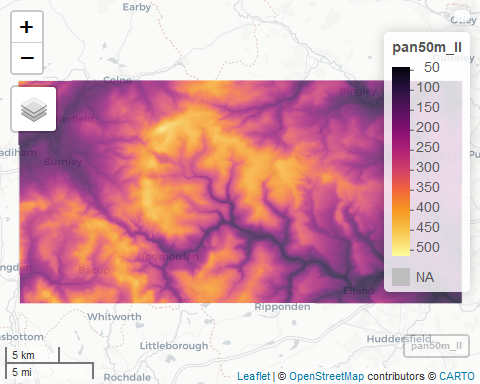


# You will notice that the default colours are the wrong way round, so we can  
# use the terrain.colors option to set low elevation to green, high to brown,   
# with 30 colour categories  
plot(pan50m, col=terrain.colors(30))



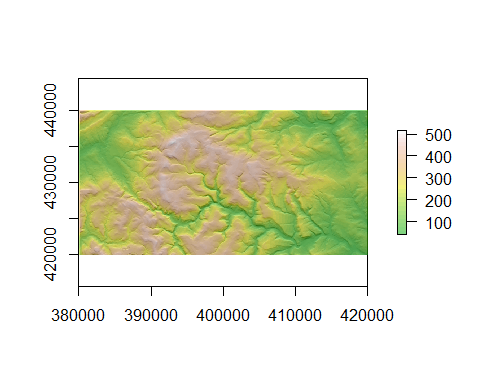
This provides a static map, using the OS projection system (note the coordinates on the horizontal and vertical axes) of the elevation in the North Pennines between Burnley and Bradford. To create a dynamic map that you can overlay onto several different baselayers, remember to convert to latitude-longitude first.

ll\_crs <- CRS("+init=epsg:4326") # 4326 is the code for latitude longitude  
pan50m\_ll <- projectRaster(pan50m, crs=ll\_crs)  
mapview(pan50m\_ll)



As maps created with mapview you can zoom and pan your map, display or hide the elevation data, show a backdrop with satellite, Open StreetMap etc. If you want to be really fancy, it is relatively straightforward to create a hillshade map from elevation data, which can also provide an attractive visualisation that highlights the structure of the elevation as if caught by sunlight.:

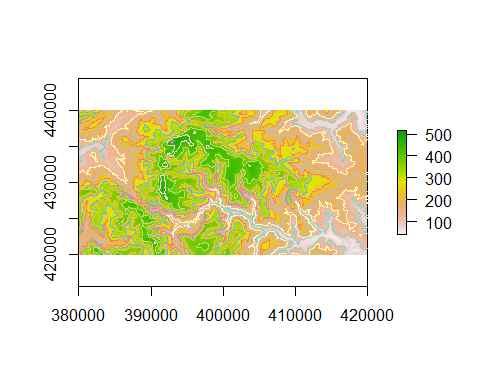
hs = hillShade(slope = terrain(pan50m, "slope"), aspect = terrain(pan50m, "aspect"))  
plot(hs, col = gray(0:100 / 100), legend = FALSE)  
# overlay with DEM  
plot(pan50m, col = terrain.colors(25), alpha = 0.5, add = TRUE)



## 1.2 Creation of contours from raster DTM

It is easy to create a contour map from your dem using rasterToContour. I’ve pushed the output through st\_as\_sf() to convert it from sp format to sf vector format, as the latter is becoming more widely used in R.

pan\_contours <- rasterToContour(pan50m) %>% st\_as\_sf()  
plot(pan50m)  
plot(pan\_contours, add=TRUE)



# 2. Add DTM-derived information to site data

## 2.1 Wind turbines dataset

We have two wind farms, with turbines at slightly different heights. In ArcGIS the height of the turbine is recorded in OFFSETA, and the height of the human observer in OFFSETB. The turbine data are in the wind\_turbine.shp shapefiles, which again I am assuming are in your gis\_data subfolder within your RStudio project. **Note** Although we usually refer to “a shapefile” it is actually a collection of files, with slightly different extensions (.prj, .sbx, .dbf, .shp). When you read the shapefile into RStudio, you only need to give the .shp filename, and it will automatically detect the others.

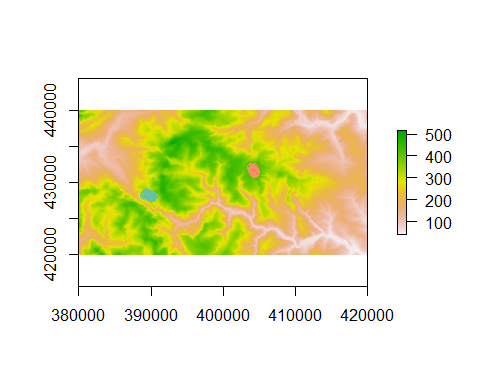
wind\_turbines <- st\_read("gis\_data/wind\_turbines.shp")

## Reading layer `wind\_turbines' from data source `C:\Temp\nras\BIO8068-GIS\gis\_data\wind\_turbines.shp' using driver `ESRI Shapefile'  
## Simple feature collection with 47 features and 4 fields  
## geometry type: POINT  
## dimension: XY  
## bbox: xmin: 388982.8 ymin: 427813 xmax: 404566.7 ymax: 432212  
## projected CRS: OSGB 1936 / British National Grid

print(wind\_turbines)

## Simple feature collection with 47 features and 4 fields  
## geometry type: POINT  
## dimension: XY  
## bbox: xmin: 388982.8 ymin: 427813 xmax: 404566.7 ymax: 432212  
## projected CRS: OSGB 1936 / British National Grid  
## First 10 features:  
## WF\_Name Turb\_ID OFFSETA OFFSETB  
## 1 Ovenden Moor, West Yorkshire OM1 54 1.5  
## 2 Ovenden Moor, West Yorkshire OM2 54 1.5  
## 3 Ovenden Moor, West Yorkshire OM3 54 1.5  
## 4 Ovenden Moor, West Yorkshire OM4 54 1.5  
## 5 Ovenden Moor, West Yorkshire OM5 54 1.5  
## 6 Ovenden Moor, West Yorkshire OM6 54 1.5  
## 7 Ovenden Moor, West Yorkshire OM7 54 1.5  
## 8 Ovenden Moor, West Yorkshire OM8 54 1.5  
## 9 Ovenden Moor, West Yorkshire OM9 54 1.5  
## 10 Ovenden Moor, West Yorkshire OM10 54 1.5  
## geometry  
## 1 POINT (403716.8 432174.4)  
## 2 POINT (403817.3 432094)  
## 3 POINT (403895.1 431981.4)  
## 4 POINT (403950.1 431862.1)  
## 5 POINT (403978.2 431756.2)  
## 6 POINT (403972.8 431619.4)  
## 7 POINT (404029.1 431543)  
## 8 POINT (404121.6 431425)  
## 9 POINT (404175.3 431325.8)  
## 10 POINT (404216.8 431206.5)

plot(pan50m)  
plot(wind\_turbines["WF\_Name"], add=TRUE)



As you have imported the data using st\_read the wind\_turbines object is of class sf, which gives a convenient display if printed in the R Console, showing both the coordinate reference system (Ordnance Survey GB) and the first 10 records, including OFFSETA and OFFSETB. If you wish, use st\_transform(wind\_turbines, 4326) to create a latitude-longitude version of your windfarm data, and view it interactively in mapview.

## 2.3 Calculate slope and 2.4 Calculate aspect

To create a slope and aspect raster maps the terrain function can be used. This is quite straightforward to implement in R. By default it calculates both slope and aspect in radians, so for consistency with ArcGIS I’ve added unit="degrees". The interpretation of the aspect values is identical to ArcGIS, i.e. values near 350, 360, 0, 5 degrees are north-facing, around 90 degrees east-facing etc.

dem\_slope <- terrain(pan50m, unit="degrees") # defaults to slope  
dem\_aspect <- terrain(pan50m, opt="aspect", unit="degrees")  
plot(dem\_slope)  
plot(dem\_aspect)

After enduring the sluggishness of ArcGIS, you might be pleasantly surprised by how quickly some of these commands run!

## 2.4 Add slope and aspect values to wind turbines attributes

Strictly-speaking this isn’t essential to calculate viewsheds, but I’ve included it here as it is straightforward in R should you require this type of facility. Basically, you’re going to transfer some of the information from the dem\_slope and dem\_aspect maps you’ve just created into two new columns in your wind\_turbines map, using the extract function. This simply creates a list of the extracted values at each wind turbine, and we can assign this to a new column using the wind\_turbines$slope <- or wind\_turbines$aspect <-

wind\_turbines$slope <- extract(dem\_slope, wind\_turbines)  
wind\_turbines$aspect <- extract(dem\_slope, wind\_turbines)

Print the first 10 rows of wind\_turbines in the RStudio Console window, and you will see the two additional columns. If you want to see e.g. 20 rows, simply issue print(wind\_turbines, n=20) in the Console.

# 3. Create a viewshed

## 3.1 Viewshed concept

Recall that a viewshed map shows you from where in the landscape an object, such as a wind turbine, can be seen. Whilst nearly all GIS have viewshed tools, unfortunately none of the spatial packages in R have a viewshed function, therefore I have created one for you, in the file LOS.R. Download this file from Canvas and save it in your RStudio project folder. When you add the command

source("LOS.R")

to your script several additional functions will be created, the key one being viewshed(). This takes the following arguments:

* dem = The name of the digital elevation map, assumed to be raster format and Ordnance Survey projection (EPSG 27700)
* windfarm = The name of the windfarm location, assumed to be sf point geometry format. Currently the viewshed function only accepts a single point as it would be too slow to run across multiple points. Therefore you will need to pick a single wind-turbine in the centre of each of your two windfarms in this example, and run viewshed twice. Only the geometry is required, so you may have to use st\_geometry(mapname)[[1] to just get the east and north coordinates (see example below).
* h1 = The height of the human observer, i.e. OFFSETA. Defaults to 1.75m if not given.
* h2 = The height of the wind turbine, i.e. OFFSETB. Defaults to 75m if not given.
* radius = Maximum radius for the viewshed, i.e. RADIUS2. Defaults to NULL if not given. The reliability of my viewshed() function seems poor at longer distances due to numerical rounding errors, so I would recommend that you always set a radius, probably maximum of 5000 to 10000 (5km to 10km)
* vector = Whether to return results as a points vector output. Defaults to raster if not given.

You might be wondering why OFFSETA and OFFSETB are not automatically collected from the wind\_turbines map. This is partly as coding it would be more complex, but also to give you maximum flexibility when creating a Decision Support System. For example, you might want the end-user to be able to vary the potential height of a turbine (within a range) to determine the effects on the viewshed.

## 3.2 Using the viewshed function: separate windfarms

The next steps guide you through creating separate viewsheds for the west (Lancashire) and east (Yorkshire) windfarms, and then merging together. As we will only pick a central point for one of the turbines at each windfarm, the viewshed will not be quite as accurate as that calculated in ArcGIS.

### 3.2.1 Viewshed for western area

First display the windfarms (as latitude-longitude) in mapview and identify the name of a wind-turbine roughly in the middle of the western windfarm:

# Convert to latitude-longitude; EPSG code 4326  
wind\_turbines\_ll <- st\_transform(wind\_turbines, 4326)  
mapview(wind\_turbines\_ll)

Zooming into the western windfarm, clicking on one of the points representing a trubine displays wind turbine “CC7” (feature ID 30). You can choose a different one if you prefer. Let’s now get it out of the wind\_turbines (Ordnance Survey) points map:

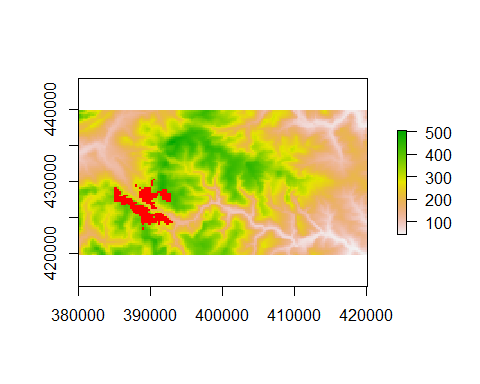
west\_windfarm <- dplyr::filter(wind\_turbines, Turb\_ID == "CC7")

Later we will look at how you can interactively click on a map to decide on where to place your turbine. The function I have written is not particularly efficient, so to save time we will aggregate up the elevation data to a coarser resolution using the aggregate function. Now you can calculate your viewshed; remember to source("LOS.R") to access the viewshed() function first. We only want to pass the windfarm geometry for the single point into the function, which we can extract with the st\_geometry function, taking the first entry by [[1]]. We can use a simpler approach if you are building a decision support system.

# Change to coarser 500m elevation map for speed  
pan500m <- aggregate(pan50m, fact=5) # fact=5 is the number of cells aggregated together  
  
# Extract just the geometry for a single mast, and pass to viewshed function.  
# Adding a 5km maximum radius  
# Takes 1 to 2 minutes to run viewshed depending on your PC  
west\_windfarm\_geom <- st\_geometry(west\_windfarm)[[1]]  
west\_viewshed <- viewshed(dem=pan500m, windfarm=west\_windfarm\_geom,  
 h1=1.5, h2=49, radius=5000)

## Loading required package: plyr

# Display results  
plot(pan500m)  
plot(west\_viewshed, add=TRUE, legend=FALSE, col="red")

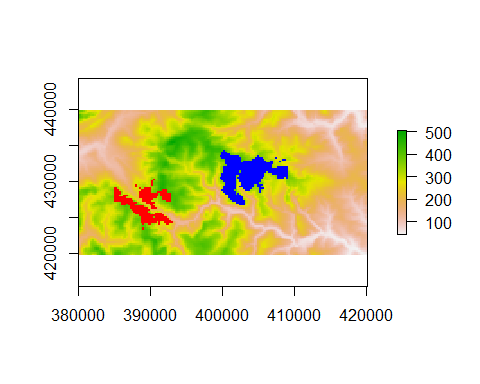


Due to the changes we have made (single wind turbine, and coarser digital elevation model) the output is not identical to the one from ArcGIS, but it is similar.

### 3.2.2 Viewshed for eastern area

Now we can repeat the process for the east (Yorkshire) windfarm, which has slightly taller turbines. Again, interactively look at the wind\_turbines\_ll map using mapview and select one of the turbines near the middle. This is a bit trickier as they are roughly in two parallel lines, but I’ve gone for “OM7” (which is record 7) at the Ovenden Moor site in Yorkshire.

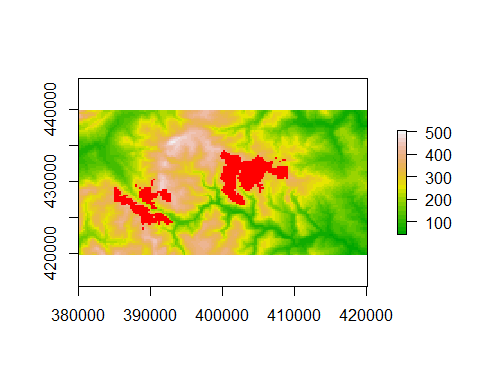
# Get the OM7 turbine  
east\_windfarm <- dplyr::filter(wind\_turbines, Turb\_ID == "OM7")  
  
# Extract geometry and calculate viewshed  
east\_windfarm\_geom <- st\_geometry(east\_windfarm)[[1]]  
east\_viewshed <- viewshed(dem=pan500m, windfarm=east\_windfarm\_geom,  
 h1=1.5, h2=54, radius=5000)  
  
# Display results  
plot(pan500m)  
plot(west\_viewshed, add=TRUE, legend=FALSE, col="red")  
plot(east\_viewshed, add=TRUE, legend=FALSE, col="blue")



### 3.2.3 Merge East and West viewsheds

When you create your decision support system you will only need to look at a single wind turbine at a single wind farm, but for completeness, we will now merge the two viewsheds into a single map. Before we can merge them together, however, we have to ensure that they have the same extent. If you plot each one separately, you’ll see one is based in Lancashire to the west, the other Yorkshire to the east. So initially, reset their extents to the full original digital elevation map, then merge:

west\_viewshed <- extend(west\_viewshed, pan500m) # could use 50m  
east\_viewshed <- extend(east\_viewshed, pan500m)  
both\_viewshed <- merge(west\_viewshed, east\_viewshed)  
plot(pan500m, col=terrain.colors(25))  
plot(both\_viewshed, legend=FALSE, add=TRUE, col="red")



## 3.3 Which settlements can see the viewshed?

Originally this had to be done in 4 stages in ArcGIS, but is actually somewhat simpler in RStudio. We’ll begin by importing the map of settlements, as a simple features polygon (vector) object:

settlements <- st\_read("gis\_data/settlements.shp")

## Reading layer `settlements' from data source `C:\Temp\nras\BIO8068-GIS\gis\_data\settlements.shp' using driver `ESRI Shapefile'  
## Simple feature collection with 74 features and 5 fields  
## geometry type: POLYGON  
## dimension: XY  
## bbox: xmin: 378510 ymin: 417730 xmax: 429686 ymax: 442570  
## projected CRS: OSGB 1936 / British National Grid

Check out the columns in the settlements object; if you wish, use st\_transform with EPSG 4326 to convert to settlements\_ll so that you can view it interactively via mapview.

### 3.3.1 Reclassify viewshed map into one class

In ArcGIS you were creating a separate viewshed for each turbine; this resulted in multiple classes automatically being created. In RStudio, you have simply created two viewsheds (east and west) then merged them into a single map both\_viewshed which already contains just one class. Therefore this step is not needed. Should you ever need to reclassify a raster map in R, use the reclassify function, which takes a map, and a simple matrix of reclassification codes or ranges.

### 3.3.2 Convert viewshed map into polygon map

Whilst my viewshed function has a vector output option, this is in the form of point data. If you actually want a polygon output, as here, you need to convert the raster to a polygon. We can do this via the rasterToPolygons function (watchTheCaptialisation!) but note that this returns an older sp object, so we will push it into st\_as\_sf to ensure we have the more modern sf object:

both\_viewshed\_poly <- rasterToPolygons(both\_viewshed) %>% st\_as\_sf()  
plot(both\_viewshed\_poly)  
print(both\_viewshed\_poly, n=5)

If you print out both\_viewshed\_poly you can see that it actually contains over 800 entries. Look closely at the map: each raster cell has been turned into its own polygon, which we do not want. Every entry in the table has the same value in the column headed layer.

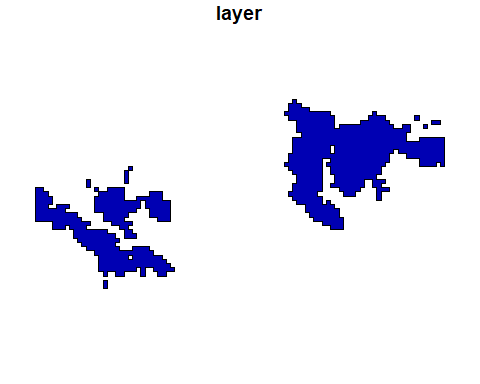
### 3.3.3 Dissolve viewshed polygons into a single polygon

There are actually two ways of doing this. First, is to add dissolve=TRUE to the rasterToPolygons function. However, this requires installation of an additional library, called rgeos which sometimes causes problems. If you are able to install rgeos then simply use:

both\_viewshed\_poly <- rasterToPolygons(both\_viewshed, dissolve=TRUE) %>% st\_as\_sf()  
plot(both\_viewshed\_poly)  
print(both\_viewshed\_poly, n=5)

An alternative is to group and summarise based on the layer column, using functions from dplyr which you probably already have installed:

both\_viewshed\_poly <- both\_viewshed\_poly %>%   
 dplyr::group\_by(layer) %>%  
 dplyr::summarize()  
plot(both\_viewshed\_poly)



print(both\_viewshed\_poly, n=5)

## Simple feature collection with 1 feature and 1 field  
## geometry type: MULTIPOLYGON  
## dimension: XY  
## bbox: xmin: 384725 ymin: 423275 xmax: 408975 ymax: 434525  
## CRS: +proj=tmerc +lat\_0=49 +lon\_0=-2 +k=0.9996012717 +x\_0=400000 +y\_0=-100000 +ellps=airy +units=m +no\_defs  
## # A tibble: 1 x 2  
## layer geometry  
## <dbl> <MULTIPOLYGON [m]>  
## 1 1 (((388975 423525, 388975 423275, 388725 423275, 388725 423525, 388725 4~

Now you can see there is only one polygon (of type MULTIPOLYGON) with no sub-divisions into small squares within the main map.

### 3.3.4 Clip the settlements map with the dissolved viewshed map

There are several methods of clipping sf data in R, depending on whether you want to keep items inside, outside, both, neither etc. of the pairs of maps. Here you are using the settlements as the original features, and the dissolved viewshed as the equivalent of your “cookie cutter”, so you only keep features where both occur:

settlements\_my\_viewshed <- st\_intersection(settlements, both\_viewshed\_poly)

Error in geos\_op2\_geom("intersection", x, y) : st\_crs(x) == st\_crs(y) is not TRUE

The error message indicates that the coordinate reference systems are not quite the same, and this can be a problem in complex analyses where you have done several conversions. Print in the RStudio Console information about the settlements map and you will see:

projected CRS: OSGB 1936 / British National Grid

However the equivalent line for both\_viewshed\_poly gives:

CRS:+proj=tmerc +lat\_0=49 +lon\_0=-2 +k=0.9996012717 +x\_0=400000 +y\_0=-100000 +ellps=airy +units=m +no\_defs

This is also Ordnance Survey British National Grid, but expressed as WKT (well-known text) format. To get both\_viewshed\_poly to behave, simply push it through st\_transform with EPSG 27700, and run the st\_intersection function again:

both\_viewshed\_poly <- st\_transform(both\_viewshed\_poly, 27700)  
settlements\_my\_viewshed <- st\_intersection(settlements, both\_viewshed\_poly)

## Warning: attribute variables are assumed to be spatially constant throughout all  
## geometries

Print out the results of your analysis to the RStudio Console, and display the maps - you will see that there are only a tiny number of settlements within 5 km of the windfarms. This probably was one of the reasons why these two sites were chosen, so that visual impact is minimal. If you wish, convert your settlements\_my\_viewshed map to a latitude-longitude version, and overlay in mapview for interactive display.

# A walkthrough example of the viewshed function in a Shiny app

This can be done by using the observeEvent() function and getting it to look for a click event. The syntax is a little strange, in that if your leaflet map is called map, the observeEvent will look for something called map\_click. It is easiest to store the results of map\_click in another R object, so that you can check the coordinates. I have created a very simple Shiny app which you are free to adapt and modify. This is available on Canvas to download, and can also be viewed at <https://naturalandenvironmentalscience.shinyapps.io/viewshed/> As you can see, it is a very simple app (no explanatory text!). You merely click on the map, and after about 15 seconds the viewshed is displayed.

Whilst I have added comments to the app.R script for this Shiny app, the following explains some of its features in more detail:

library(shiny)  
library(sf)  
library(raster)  
library(leaflet)  
library(leafem)  
library(rgdal)  
source("LOS.R")

Quite a lot of libraries are required, namely shiny, sf and raster as you would expect. As the maps will be shown interactively to allow pan and zoom, both leaflet and leafem are needed. The rgdal library needs to be explicitly added for successful deployment onto the ShinyApps.io webserver. Finally, the last line of the startup includes a source() call to the LOS.R script, to provide access to the viewshed and ancilliary functions. The LOS.R script is in the same folder as the main app.R script.

pan50m <- raster("www/pan50m.tif")  
pan500m <- aggregate(pan50m, fact=5) # aggregate to 500m grid for speed  
ll\_crs <- CRS("+init=epsg:4326")  
pan50m\_ll <- projectRaster(pan50m, crs=ll\_crs)

These 4 lines read in the TIF file containing the elevation, make it a coarser resolution of 500m to speed things up, and reprojects the 50m version to latitude longitude which is required by leaflet. Whilst the higher resolution 50m lat-lon version will be displayed onscreen for the user, the coarser 500m resolution will be used for calculations. The reprojection to EPSG 4326 is latitude longitude, which we store in ll\_crs for re-use later.

ui <- fluidPage(  
 leafletOutput(outputId = "map")  
)

The user-interface ui function is tiny, and all we have specfied is a leaflet output, which we have named map. Later on, you’ll see reference to map\_click; if you decided to give the ui an outputID = "elevation") then later you would need to change the code to elevation\_click etc. All the next code is within the server function:

output$map <- renderLeaflet({  
 leaflet() %>%  
 setView(lng = -2, lat=53.75, zoom=11) %>%  
 addRasterImage(pan50m\_ll, colors=terrain.colors(25))  
 })

This sets up leaflet map to be on roughly the right area (North Pennines) and then adds the elevation map with terrain colours. The zoom option changes the overall startup zoom when the application first runs. Note that setView uses longitude and latitude.

observeEvent(input$map\_click, {  
 coord <- input$map\_click  
 lng <- coord$lng   
 lat <- coord$lat

This is the start of a long observeEvent section of code, so look at the indentation and brackets to ensure you understand how long it is. It actually finishes on **Line 52** quite near the end of the code. These 4 lines detect when the elevation map has been clicked, grabs the coordinates, and then stores the longitude and latitude in two R variables.

if(length(c(lng,lat))==2){

Another block of indented code begins here. It is just to check that the length of the latitude and longitude is 2 numbers long. When the Shiny app first begins, there is not a latitude and longitude stored (as you haven’t clicked the map yet), so this if statement avoids error messages.

turbine\_pt\_ll <- data.frame(lat = lat, lng = lng) %>%   
 st\_as\_sf(coords = c("lng", "lat")) %>%   
 st\_set\_crs(4326)  
 turbine\_pt\_os <- st\_transform(turbine\_pt\_ll, crs=27700)  
 turbine\_pt\_os <- st\_geometry(turbine\_pt\_os)[[1]] # Only want geometry

These lines get everything ready for calculating the viewshed, by storing the latitude and longitude in a sf points object, converting it to Ordnance Survey, and extracting the geometry. The first three lines creates your turbine mast at latitude longitude (4326), then st\_transform converts to OS. The last line just gets the coordinates via st\_geometry, as my viewshed function is very simple in what it expects.

viewshed\_5km\_os <- viewshed(dem=pan500m, windfarm=turbine\_pt\_os,  
 h1=1.5, h2=50, radius=5000)  
 viewshed\_5km\_ll <- projectRaster(viewshed\_5km\_os, crs=ll\_crs)

Now it is simply a case of calling the viewshed function, remembering to use pan500m and not pan50m as the latter would take 15 minutes to run, setting it to 5 km radius, 50m turbine height and 1.5m viewer. We have to re-project back to latitude-longitude for display of the results in leaflet.

leafletProxy("map") %>%   
 addRasterImage(viewshed\_5km\_ll, color="red")

Now we can add our viewshed onto the original map. The leafletProxy function is useful in that the original map is not redrawn, so you can click multiple times to see different viewsheds on the same map.

Obviously this is very simple, and a better version would allow you to choose the radius of the viewshed, height of the turbine, include explanatory text for the user etc.