Spatial analysis in R

Hans Gerritsen

2023-06-02

Introduction

There are many vignettes and cheat sheets on using R for mapping and spatial analysis. This one is intended to summarise some of the things that we may wish to do with fisheries data, including fishing surveys, observer data and vessel monitoring data.

Packages

There are many packages in R that deal with spatial data, mapping etc. The rgdal and rgeos packages are due to retire by the end of 2023 the sf (simple features) package combines the functionality of these. It can be hard to get your head around the various types of objects but probably worth the effort if you want to work wit spatial objects R.

For basic maps, you can use the mapdata package, or use your own shapefiles/polygons. You can use base R plotting but ggplot can also do a nice job.

For grid data, the raster package is very good. It has actually be superseded with the terra package but I haven't made the switch yet.

```
library(sf) # all things spatial
library(maps) # for base R type maps
library(mapdata) # map data
library(ggplot2) # ggplot can also be used for mapping
library(gridExtra) # multiple ggplot maps on one page
library(dplyr) # for data wrangling
```

Points, lines, polygons

Points

Lets make up some different data sets: start with the most basic: points. Here we generate a random set of sampling locations, roughly in the area around Ireland. Some will be on land but we deal with that later.

```
set.seed(1) # this makes the random selecton reproducible
points_df <- data.frame(id=paste('station',1:100)
                        ,lon=runif(100,-12,-4)
                        ,lat=runif(100,50,56)
                        ,catch=rgamma(100,1,1))
points_sf <- st_as_sf(points_df,coords=c('lon','lat'),crs=4326)
## crs=4326 is WGS84, e.g. https://epsg.io/4326
```

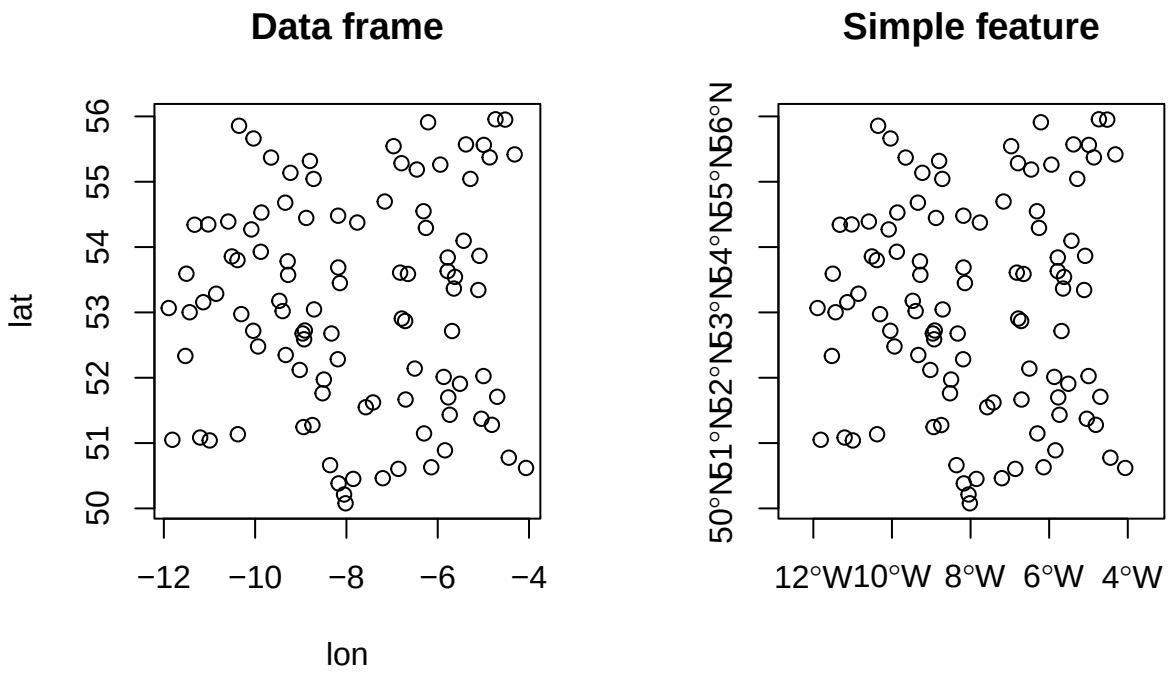
This is what the simple feature dataframe looks like:

```
head(points_sf)
```

```
## Simple feature collection with 6 features and 2 fields
## Geometry type: POINT
## Dimension:     XY
## Bounding box:  xmin: -10.38654 ymin: 51.27925 xmax: -4.734338 ymax: 55.9561
## Geodetic CRS:  WGS 84
##          id       catch                   geometry
## 1 station 1 0.157548219 POINT (-9.875931 53.92834)
## 2 station 2 0.159290251 POINT (-9.023009 52.11918)
## 3 station 3 0.617986490 POINT (-7.417173 51.62156)
## 4 station 4 0.144259023  POINT (-4.734338 55.9561)
## 5 station 5 0.004453169 POINT (-10.38654 53.80096)
## 6 station 6 1.350694652 POINT (-4.812883 51.27925)
```

In order to plot the simple feature using base R we extract the geometry using st_geometry. Note that the axes look different from just plotting a normal data frame.

```
par(mfrow=c(1,2))
plot(points_df[,c('lon','lat')],main='Data frame')
plot(st_geometry(points_sf),axes=T,main='Simple feature')
```




```
par(mfrow=c(1,1))
```

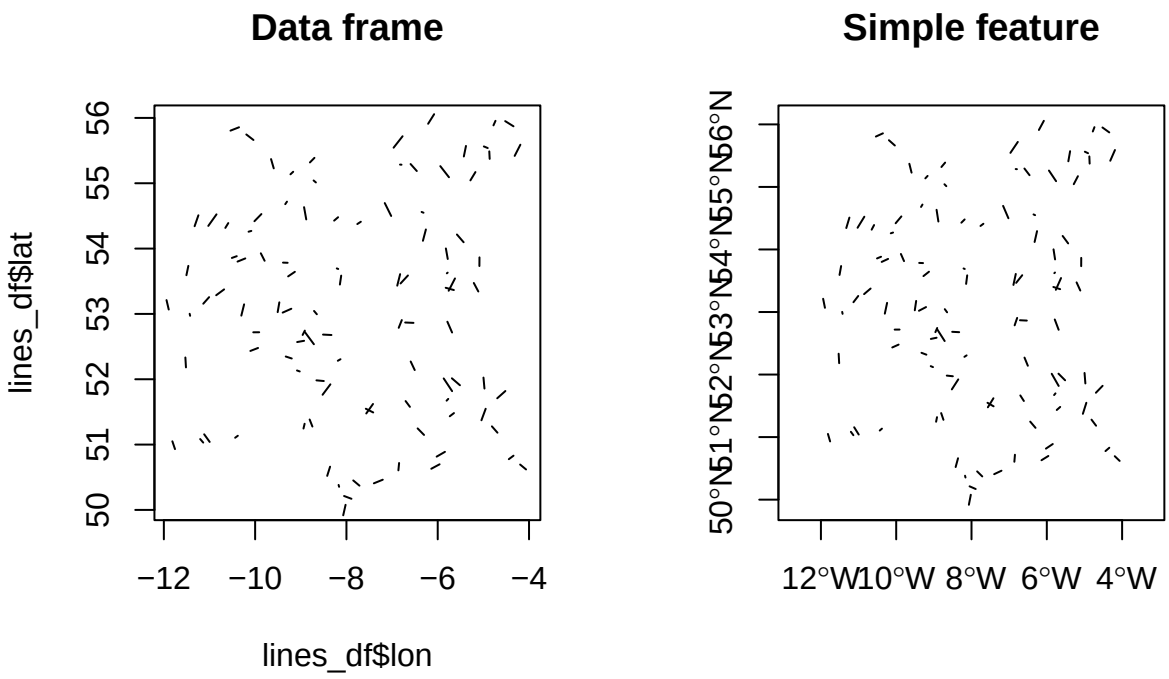
Lines

To demonstrate working with lines, we will make up a tow track for each sampling point. First we make a list of start and end positions. This is converted into a linestring, then combined into a multilinestring and finally a simple features dataframe. There are a number of ways make such a data frame but I found the approach below to be the most intuitive.

```
## use jitter to create a second location close to each sampling location
lines_df <- points_df
lines_df$lon1=jitter(lines_df$lon,1000)
lines_df$lat1=jitter(lines_df$lat,1000)

## make a list, each element has the start and end position of one line
lines_list <- split(lines_df[,c('lon','lon1','lat','lat1')],lines_df$id)
## make each element of the list into a matrix of coordinates
coords_list <- lapply(lines_list,function(x) matrix(as.numeric(x),ncol=2))
## make each element of the list into a linestring
lines_ls <- lapply(coords_list,st_linestring)
## add a coordinate system (more about that later)
lines_sfc <- st_as_sfc(lines_ls,crs=4326)
## turn into a simple features data frame so we can add attributes
## in this case this the only attribute is the station id
lines_sf <- st_as_sf(data.frame(id=names(lines_ls),lines_sfc))
```

And plot the result

```
par(mfrow=c(1,2))
plot(lines_df$lon,lines_df$lat,main='Data frame',type='n')
segments(lines_df$lon,lines_df$lat,lines_df$lon1,lines_df$lat1)
plot(st_geometry(lines_sf),axes=T,main='Simple feature')
```




```
par(mfrow=c(1,1))
```

Polygons

Polygons are often used for plotting coastlines, survey strata, closed areas etc. We can use a coastline file from mapdata to illustrate. It comes as a normal dataframe.

```
coast <- map_data('worldHires') %>%
  subset(region %in% c('Ireland','UK','Isle of Man','Wales'))
head(coast)
```

```
##              long      lat group   order region subregion
## 1014550 -5.363343 36.15054    89 1014550     UK Gibralter
## 1014551 -5.366380 36.15140    89 1014551     UK Gibralter
## 1014552 -5.369989 36.15140    89 1014552     UK Gibralter
## 1014553 -5.373313 36.15140    89 1014553     UK Gibralter
## 1014554 -5.376693 36.15140    89 1014554     UK Gibralter
## 1014555 -5.380016 36.15140    89 1014555     UK Gibralter
```

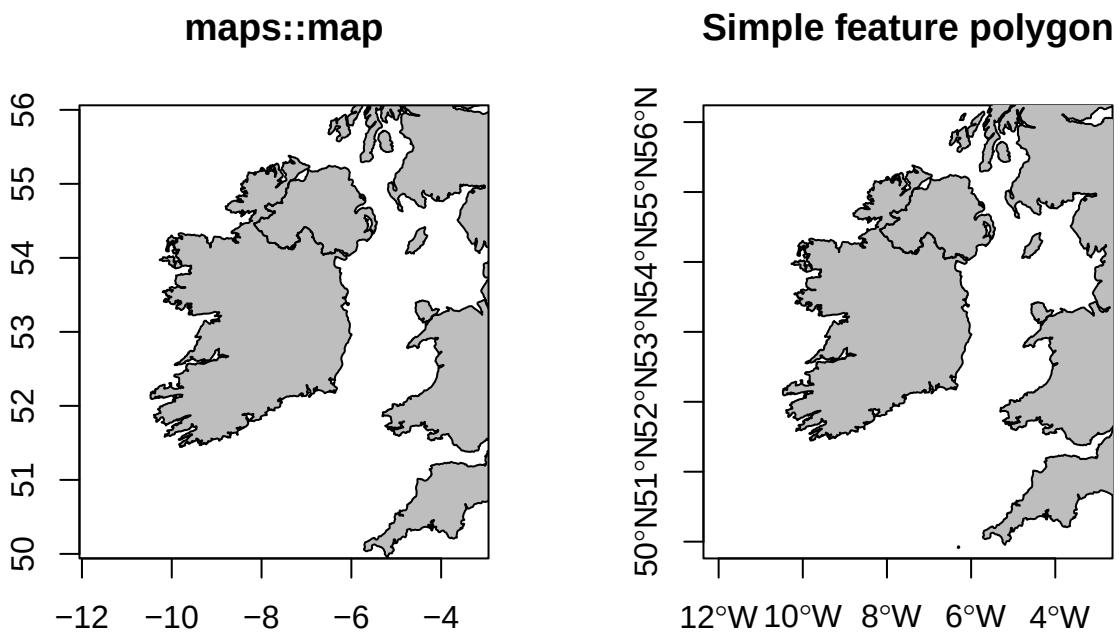
Making a dataframe of positions in to a polygon object is similar to the approach we took for lines.

```
## first split each country, island etc using the group column in the dataframe
poly_list <- split(coast[,c('long','lat')], coast$group)
## make each element of the list into a matrix of coordinates
coords_list <- lapply(poly_list, as.matrix)
## make each element of the list into a polygon
poly_st <- lapply(coords_list,function(x) st_polygon(list(x)))
```

```
## add a coordinate system (more about that later)
poly_sfc <- st_sfc(poly_st,crs=4326)
## turn into a simple features data frame so we can add attributes
attr <- unique(coast[,c('group','region','subregion')])
## the polygons are not neccesarily in the same order so be careful
i <- match(names(poly_st),attr$group)
poly_sf <- st_as_sf(data.frame(attr[i,],poly_sfc))
```

And plot the result

```
par(mfrow=c(1,2))
map('worldHires',xlim=c(-12,-3),ylim=c(50,56),col='grey',fill=T)
axis(1); axis(2); box(); title('maps::map')
plot(st_geometry(poly_sf),axes=T,xlim=c(-12,-3),ylim=c(50,56),col='grey'
     ,main='Simple feature polygon')
```




```
par(mfrow=c(1,1))
```

Reading and saving shapefiles

We often use shapefiles to read in spatial data. This is very straightforward with sf. Shapfiles are loaded in as simple feature dataframes.

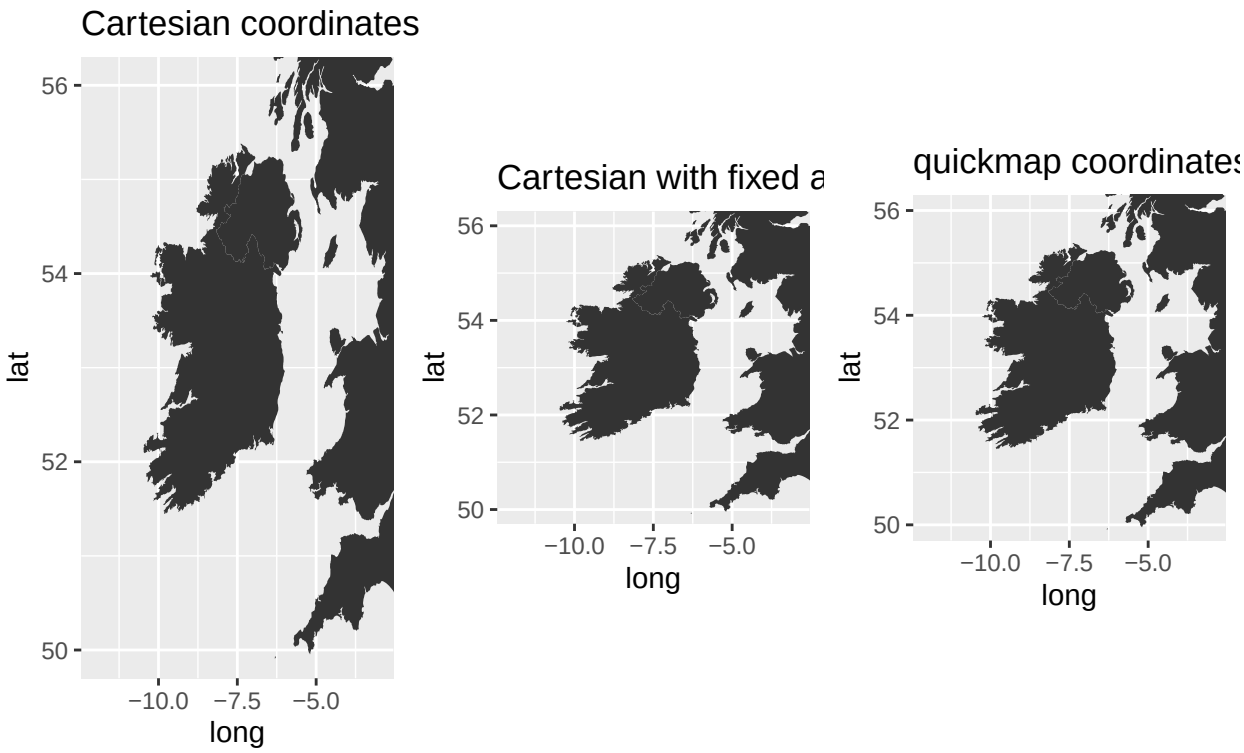
```
st_write(poly_sf,'coast.shp',append=F) # append = F means overwrite existing file
my_coast <- st_read('.','coast')
```

Plotting maps

The map function from the maps package sets the aspect ratio of the plot to an appropriate value (i.e. the map is not squashed or stretched) as long as the area you are plotting is reasonably small (i.e. smaller than the size of a continent). We have seen this in the polygon example above. We have also used the base R plot function used directly to create maps. In this case is is useful to fix the aspect ratio. Around Ireland asp=1.5 is approximately correct as 1 degree latitude is approximately 1.5 times as long as 1 degree longitude.

You can do something similar in ggplot. You can use the default (cartesian) coordinate system but this can result in stretched maps: usually maps will not have the correct aspect ratio unless you specify it yourself using coord_fixed. However there are also specific mapping coordinate functions like coord_quickmap.

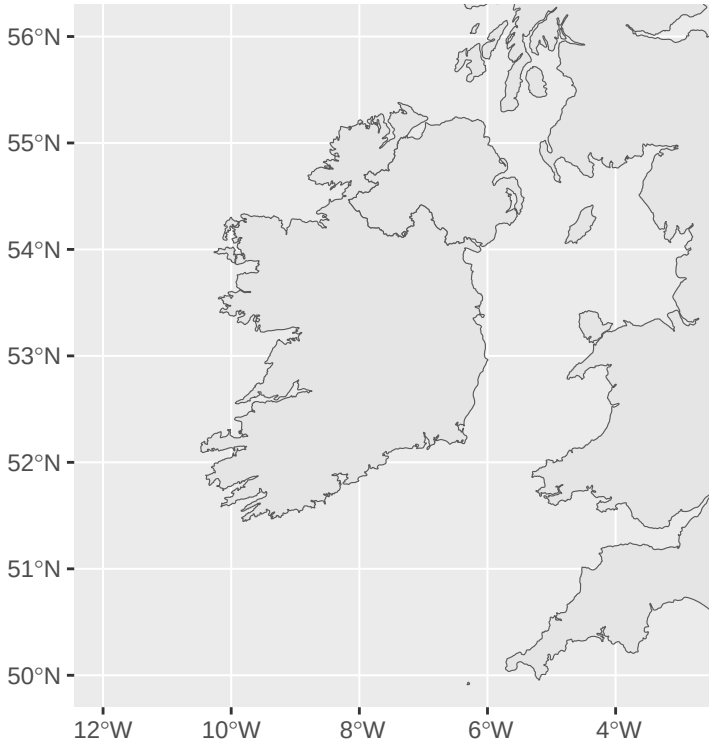
```
g1 <- ggplot(coast, aes(long, lat, group=group)) +
  geom_polygon() +
  coord_cartesian(xlim=c(-12,-3),ylim=c(50,56)) +
  ggtitle('Cartesian coordinates')

g2 <- ggplot(coast, aes(long, lat, group=group)) +
  geom_polygon() +
  coord_fixed(xlim=c(-12,-3),ylim=c(50,56),ratio=1.5) +
  ggtitle('Cartesian with fixed aspect ratio')

g3 <- ggplot(coast, aes(long, lat, group=group)) +
  geom_polygon() +
  coord_quickmap(xlim=c(-12,-3),ylim=c(50,56)) +
  ggtitle('quickmap coordinates')

grid.arrange(g1,g2,g3,nrow=1)
```



If you are working with sf objects, you need to use the coord_sf coordinate system. Note that you do not

need to specify x and y aesthetics, it uses the geometry column in the sf object. If you do want to specifiy it explicitly it would be aes(geometry=geometry) because in this case, the geometry column is called geometry (but it can be called anything).

```
ggplot(my_coast) +
  geom_sf() +
  coord_sf(xlim=c(-12,-3),ylim=c(50,56))
```



Coordinate Reference System

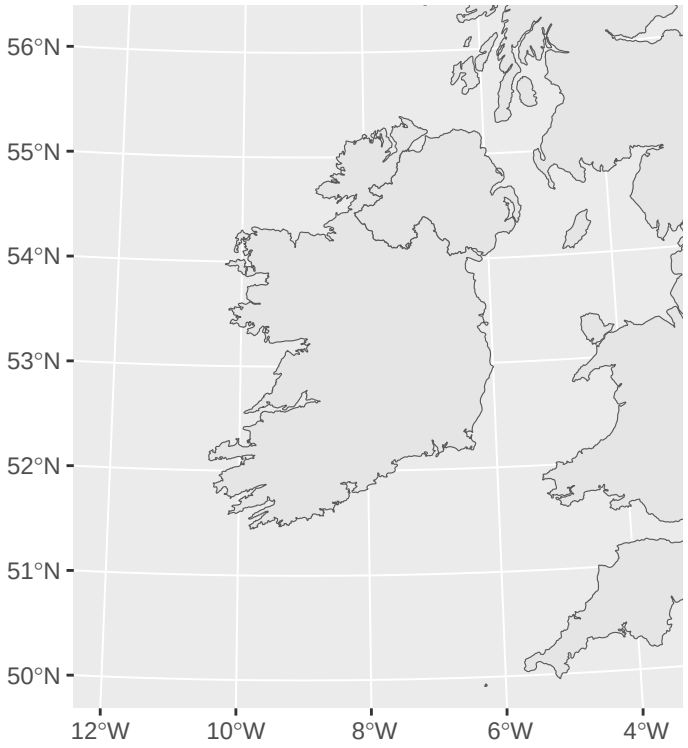
If you are using decimal degrees longitude and latitude, most likely the coordinate reference system (CRS) is WGS84, which has EPSG code 4326 (https://epsg.io/4326). If you load a shapefile, it should already have information on the CRS. It may be useful to use a projected coordinate system. Around Ireland UTM29N is often used (EPSG 32629). When you plot this, the axes will still be in degees (even though the units are now in meters). Also note the grid lines are not exactly parallel anymore.

```
st_crs(my_coast) # the current CRS is 4326, which is WGS 84.
```

```
## Coordinate Reference System:
##   User input: WGS 84
##   wkt:
## GEOGCRS["WGS 84",
##     DATUM["World Geodetic System 1984",
##         ELLIPSOID["WGS 84",6378137,298.257223563,
##             LENGTHUNIT["metre",1]]],
##     PRIMEM["Greenwich",0,
##         ANGLEUNIT["degree",0.0174532925199433]],
```

```
##     CS[ellipsoidal,2],
##         AXIS["latitude",north,
##             ORDER[1],
##             ANGLEUNIT["degree",0.0174532925199433]],
##         AXIS["longitude",east,
##             ORDER[2],
##             ANGLEUNIT["degree",0.0174532925199433]],
##     ID["EPSG",4326]]
```

```
coast_utm <- st_transform(my_coast, crs=32629) # transform to UTM 29N
ggplot(coast_utm) +
  geom_sf() +
  coord_sf(xlim=c(285016,873950),ylim=c(5542944,6222336))
```




```
## note that limits now need to be in UTM units
```

Points in polygon (spatial union)

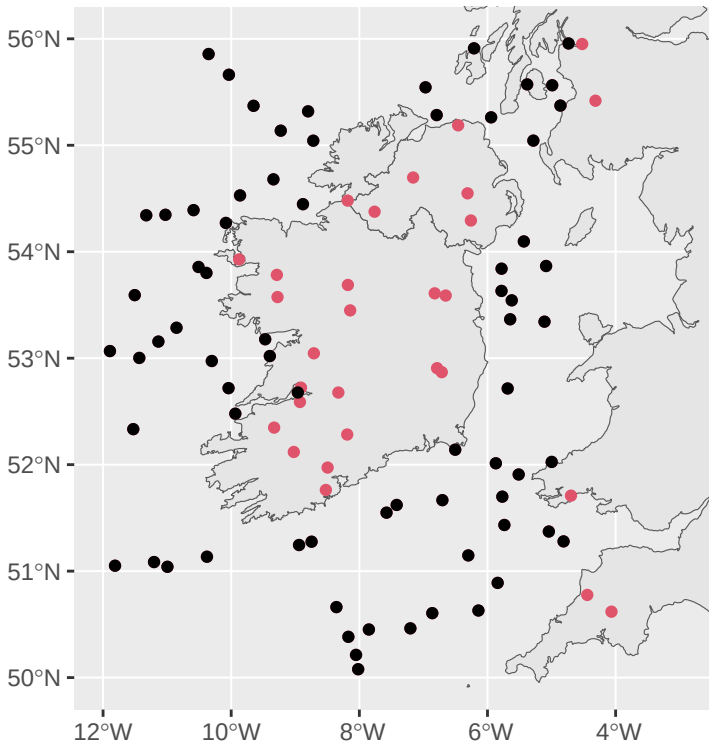
We often need to identify points inside polygons. E.g. to identify which stratum a haul was in or to see which VMS points were inside a proposed MPA. Our random survey locations resulted in some positions on land; we can do a spatial join and assign the polygon attributes to the points. Now each point that is inside a polygon (i.e. on land) has the attributes like group, region etc of the polygon. We can use this to remove any points on land.

```
points_join <- st_join(points_sf,my_coast)
head(points_join)
```

```
## Simple feature collection with 6 features and 5 fields
## Geometry type: POINT
## Dimension:     XY
## Bounding box:  xmin: -10.38654 ymin: 51.27925 xmax: -4.734338 ymax: 55.9561
## Geodetic CRS:  WGS 84
##          id       catch group  region subregion                    geometry
## 1 station 1 0.157548219   568 Ireland      <NA> POINT (-9.875931 53.92834)
## 2 station 2 0.159290251   568 Ireland      <NA> POINT (-9.023009 52.11918)
## 3 station 3 0.617986490    NA    <NA>      <NA> POINT (-7.417173 51.62156)
## 4 station 4 0.144259023    NA    <NA>      <NA>  POINT (-4.734338 55.9561)
## 5 station 5 0.004453169    NA    <NA>      <NA> POINT (-10.38654 53.80096)
## 6 station 6 1.350694652    NA    <NA>      <NA> POINT (-4.812883 51.27925)
```

We can use this to filter out points on land; in the plot below the red points are the ones that we have removed.

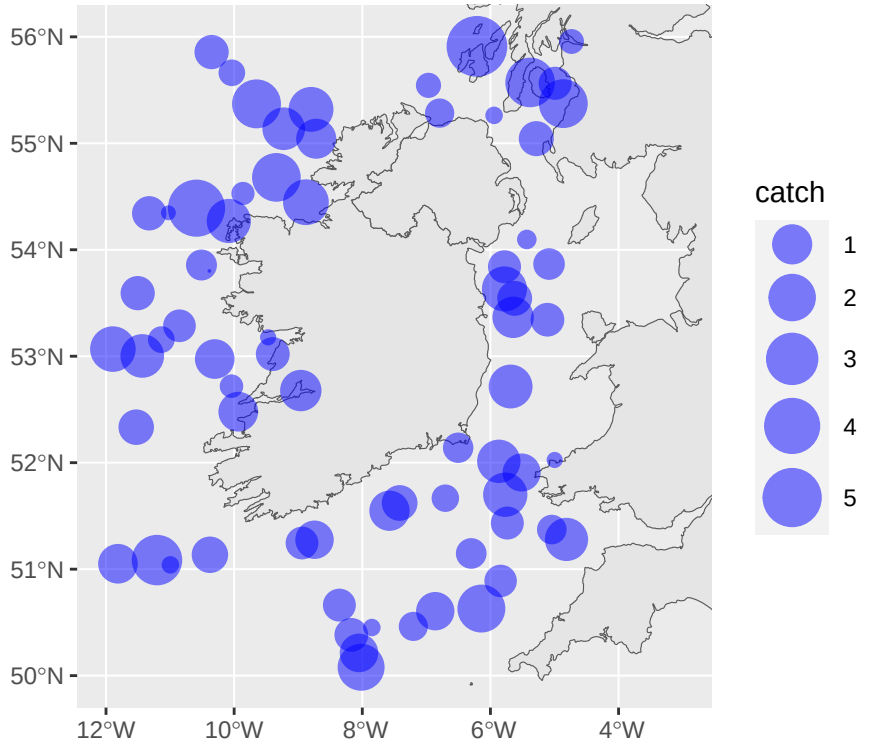
```
points_filter <- subset(points_join,is.na(group))
ggplot() +
  geom_sf(data=my_coast) +
  geom_sf(data=points_sf,col=2) +
  geom_sf(data=points_filter,col=1) +
  coord_sf(xlim=c(-12,-3),ylim=c(50,56))
```



Bubble maps

We often like to display our catch data as a bubble map showing the area of each circle in proportion to the size of the catch. This is how to do that.

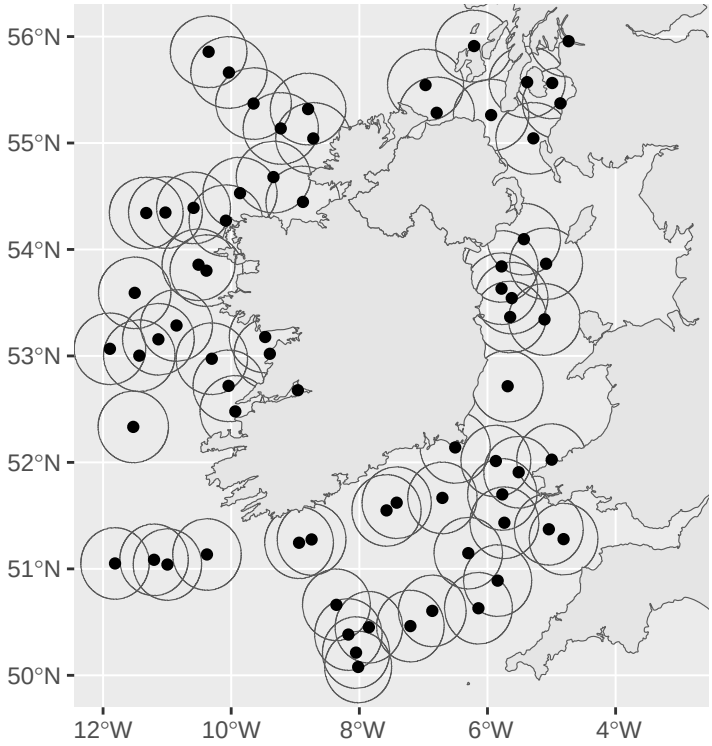
```
ggplot() +
  geom_sf(data=my_coast) +
  geom_sf(aes(size=catch),data=points_filter,col='blue',alpha=0.5) +
  scale_size(range=c(0,10),trans='sqrt') +
  coord_sf(xlim=c(-12,-3),ylim=c(50,56))
```



Buffer, union, area

There are many spatial operations you can do, just to illustrate a couple we will add a buffer around each station, here arbitrarily 20nm. Even though the points_filter object has a coordinate system of WGG 84 (in degrees longitude, latitude) it correctly applies the buffer on a scale of meters - so it must do some sort of projection. Note 1 nautical mile = 1852 meters.

```
points_buffer <- st_buffer(points_filter,dist=1852*20)
ggplot() +
  geom_sf(data=points_buffer,fill=NA) +
  geom_sf(data=my_coast) + geom_sf(data=points_filter) +
  coord_sf(xlim=c(-12,-3),ylim=c(50,56))
```

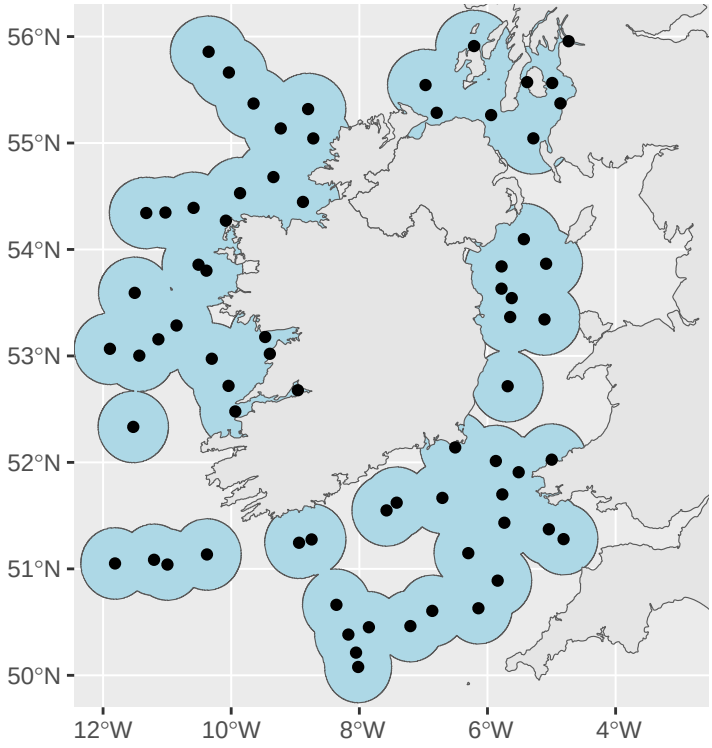


We could use this as a rough way to estimate the area covered by the survey. If we arbitrarily say that each station accounts for an area of 20nm the coverage of the survey is then the blue area below. The function st_area will calculate the area of the polygon (in m2 in this case). Note that for some reason the points_union object is not valid but this can be fixed with st_make_valid.

```
points_union <- st_union(points_buffer)
st_area(st_make_valid(points_union)) # survey area in m2
```

```
## 1.85024e+11 [m^2]
```

```
ggplot() +
  geom_sf(data=points_union,fill='lightblue') +
  geom_sf(data=my_coast) + geom_sf(data=points_filter) +
  coord_sf(xlim=c(-12,-3),ylim=c(50,56))
```

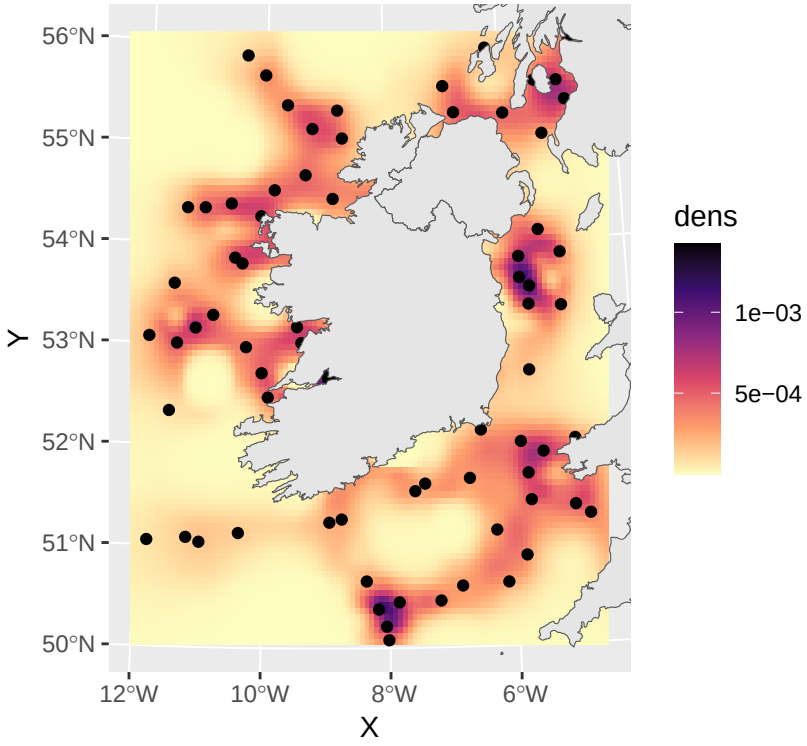


Raster

Now that we are on the subject, lets see if we can use another way to estimate the survey coverage. The locfit package can be used to estimate densities. This is also a nice way to introduce the raster library. Unfortunately is not very compatible with ggplot and coord_sf (but see the rasterVis library) so we have to flick between rasters and dataframes a bit.

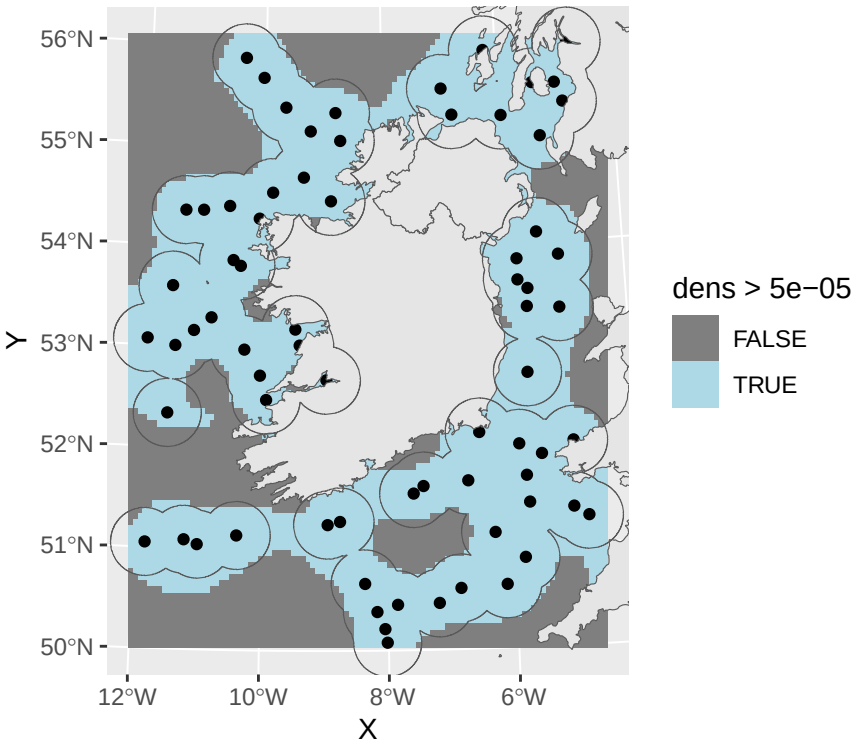
```
library(locfit)
library(raster)
## transform utm29N so we are working in km, not degrees
points_utm <- st_transform(points_join,32629)
## and back to a normal dataframe
points_coords <- st_coordinates(points_utm) %>% as.data.frame()
## set up prediction grid:
## the spatial extent of the grid needs to be in UTM
## this way we use the lat and long to get the extent of the grid in UTM
ext <- st_as_sf(data.frame(x=c(-12,-4),y=c(50,56)),coords=1:2,crs=4326) %>%
  st_transform(32629) %>% st_coordinates() %>% t() %>% extent()
## projection
crs <- '+proj=utm +zone=29 +datum=WGS84 +units=m +no_defs +type=crs' # same as EPSG 32629
## raster object (handy for later and also handy for generating a prediction grid)
pred_raster <- raster(ncol=100,nrow=100,ext=ext,crs=crs)
## grid locations back into a dataframe
pred_grid <- coordinates(pred_raster) %>% as.data.frame %>% rename(X=x,Y=y)
## density estimate is based on nearest neighbours, here we use 5
nn <- 5
nn_prop <- nn/nrow(points_utm)
## fit the model
```

```
fit_density <- locfit(~lp(X,Y,nn=nn_prop,deg=1,h=0),data=points_coords)
## predict
dens <- predict(fit_density,newdata=pred_grid) * nrow(points_utm)*1e6 # units are stations per km2
pred <- data.frame(pred_grid,dens=dens)

## and plot the result
ggplot() + geom_raster(aes(X,Y,fill=dens),data=pred) +
  scale_fill_viridis_c(option='A',direction=-1) +
  geom_sf(data=points_filter) +
  geom_sf(data=st_transform(my_coast,32629)) +
  coord_sf(crs=32629,xlim=range(pred_grid$X),ylim=range(pred_grid$Y))
```



Again, make an arbitrary choice: if the density of stations is more than 5e-5 per km then we consider the area to be part of the survey coverage. We can compare to the buffer approach. The two methods agree quite well.

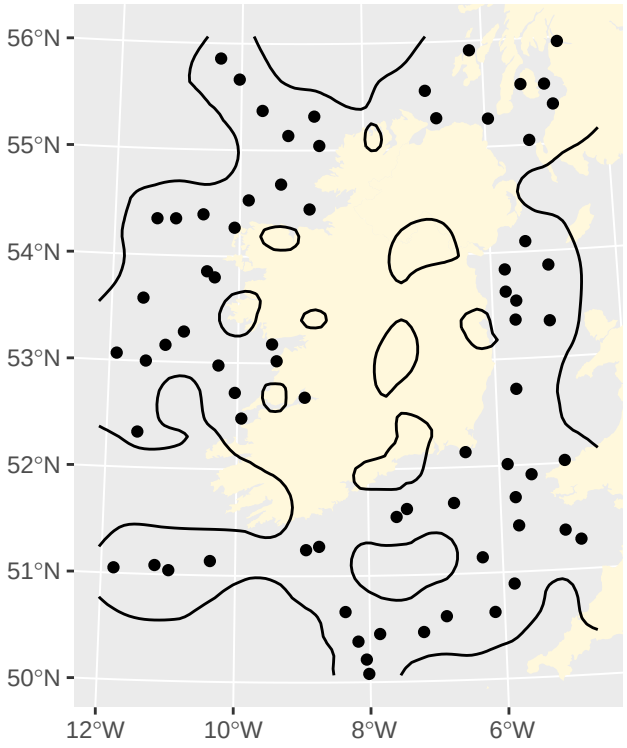
```
ggplot() + geom_raster(aes(X,Y,fill=dens>5e-5),data=pred) +
  scale_fill_manual(values=c(NA,'lightblue')) +
  geom_sf(data=points_filter) +
  geom_sf(data=st_transform(my_coast,32629)) +
  geom_sf(data=points_union,fill=NA) +
  coord_sf(crs=32629,xlim=range(pred_grid$X),ylim=range(pred_grid$Y))
```



Raster to polygon

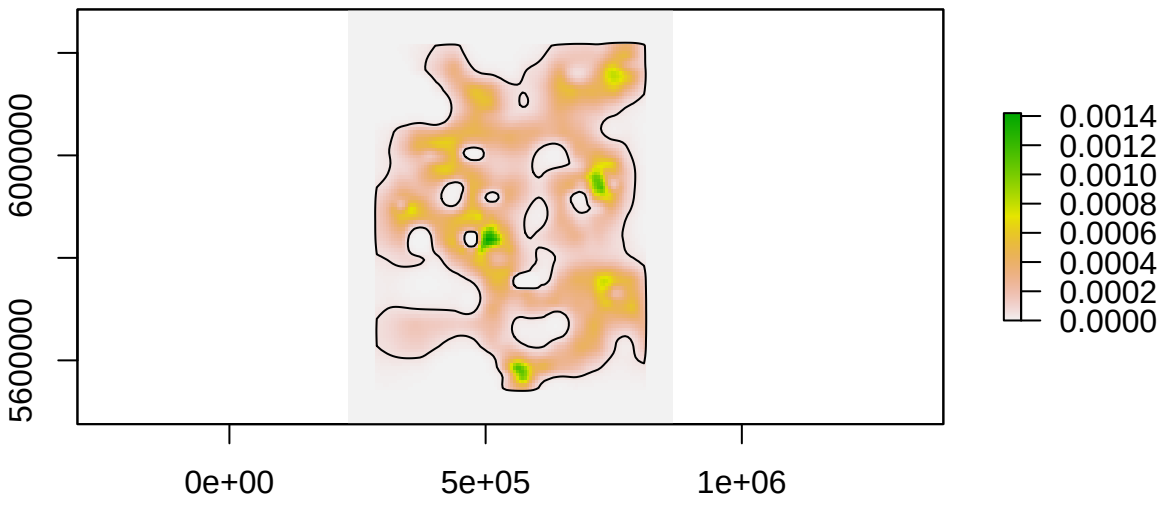
If you want to convert the model prediction to a polygon, you need to go back to a raster object.

```
pred_raster <- rasterize(pred_grid,pred_raster,dens)
# identify the contour
contour1 <- rasterToContour(pred_raster,levels=5e-5)
# make it into sf object
contour2 <-st_as_sf(contour1)

ggplot() +
  geom_sf(data=st_transform(my_coast,32629),col=NA,fill='cornsilk') +
  geom_sf(data=contour2) +
  geom_sf(data=points_filter) +
  coord_sf(crs=32629,xlim=range(pred_grid$X),ylim=range(pred_grid$Y))
```



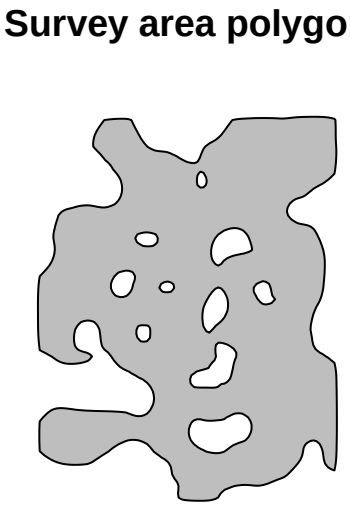
We are not quite there because we just have a bunch of lines, not a polygon. The sensible approach would be to increase the prediction area so the entire area is captured and the contour lines join up. However in some cases it might be useful to restrict the area. Here is a way to do that: firstly we extend the prediction grid and fill it with zeros so the edges join up when we run the rasterToContour function again.

```
## extend by 10 rows and 10 columns, fill with zero
extend_raster <- extend(pred_raster,c(10,10),value=0)
## contour again
contour1 <- rasterToContour(extend_raster,levels=5e-5)
plot(extend_raster)
plot(contour1,add=T)
```



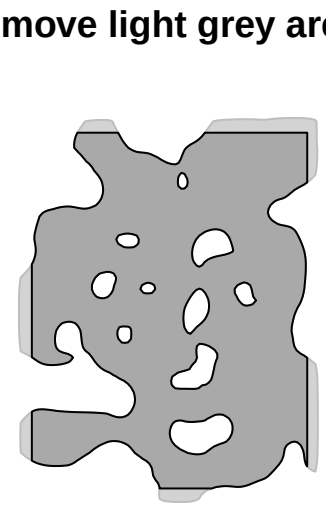
Now we can turn it into a polygon; it correctly identifies the holes

```
contour2 <- contour1 %>% st_as_sf() %>% st_make_valid()  %>% st_cast('POLYGON')
plot(st_geometry(contour2),col='grey',main='Survey area polygon')
```



Tidy up the edges (they are a bit raggedy), use a rectangular polygon as "cookie cutter".

```
box <- st_as_sf(data.frame(x=c(-11.7,-4.3),y=c(50.2,55.8)),coords=1:2,crs=4326) %>%
  st_transform(32629) %>% st_bbox() %>% st_as_sfc()
contour3 <- st_intersection(contour2,box)
plot(st_geometry(rbind(contour2,contour3)),
     ,col=c('lightgrey','darkgrey')
     ,border=c('grey','black')
     ,main='Remove light grey areas')
```



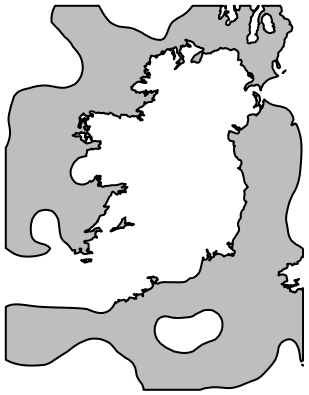
Take out the areas on land and we are done

```
contour4 <- st_difference(contour3,st_union(st_transform(my_coast,32629)))
st_area(contour4)
```

```
## 150878379654 [m^2]
```

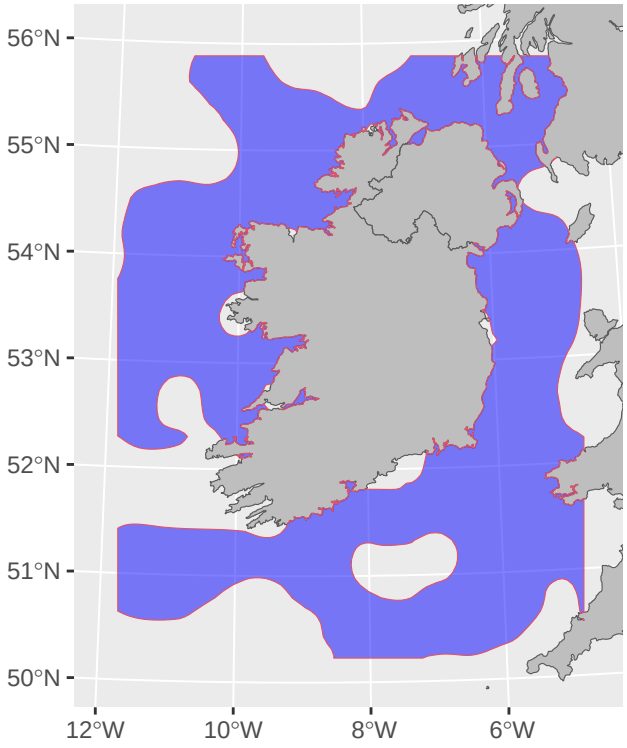
```
plot(st_geometry(contour4),col='grey',main='Remove land')
```

Remove land



The final result in ggplot.

```
ggplot() +
  geom_sf(data=st_transform(my_coast,32629),fill='grey') +
  geom_sf(data=contour4,fill='blue',alpha=0.5,col=2) +
  coord_sf(crs=32629,xlim=range(pred_grid$X),ylim=range(pred_grid$Y))
```


56°N
55°N
54°N
53°N
52°N
51°N
50°N
12°W
10°W
8°W
6°W