HDAT9800 Visualisation and Communication of Health Data

Chapter 9

Drs James Farrow & Tim Churches

Advanced Shiny maps

Lazy evaluation using reactive functions

Map grids

leafletProxy

Leaflet callbacks

Lazy evaluation of maps

Recall we can write an *eventReactive* function that only invalidates on a certain event

Normally *renderLeaflet* will update on any *input$* references used

This can result in a great deal of unwanted calculation and slowness

We can replace the *input$* variables with reactive functions

We make those reactive functions invalidate on a button event

Before

```
output$map <- renderLeaflet({
  poly <- states[[input$State]]
  grid <- ... # grid calculations
  pal_fn <- colorNumeric(c("red", "white"),
                         c(0, input$maxd))
  leaflet() %>%
    addTiles() %>%
    addPolygons(poly, stroke=TRUE, fill=FALSE) %>%
    addPolygons(grid, color=pal_fn(grid$distance))
})
```

Either input gets changed the entire map gets redrawn

We only want it to be redrawn when an update button is pushed

After

Add an *actionButton* to the UI with input id `"update"`

```
state <- eventReactive(input$update, {input$State})
maxd <- eventReactive(input$update, {input$maxd})

output$map <- renderLeaflet({
  poly <- states[[state()]]
  grid <- ... # grid calculations
  pal_fn <- colorNumeric(c("red", "white"),
                         c(0, maxd()))
  leaflet() %>%
    addTiles() %>%
    addPolygons(poly, stroke=TRUE, fill=FALSE) %>%
    addPolygons(grid, color=pal_fn(grid$distance))
})
```

Map grids

To create heat maps we need a grid of points

The *sp::makegrid* function will make a random set or regular grid of points to fill a rectangular region, usually the bounding box of a polygon or collection of polygons

- random collection of points
- evenly spaced points using *cellsize* parameter
- (approximate) given number of points using the *n* parameter

Method

Create a regular grid of points

Crop the points to the polygon (or area) of interest (with possible buffer)

Calculate interesting attributes of the points

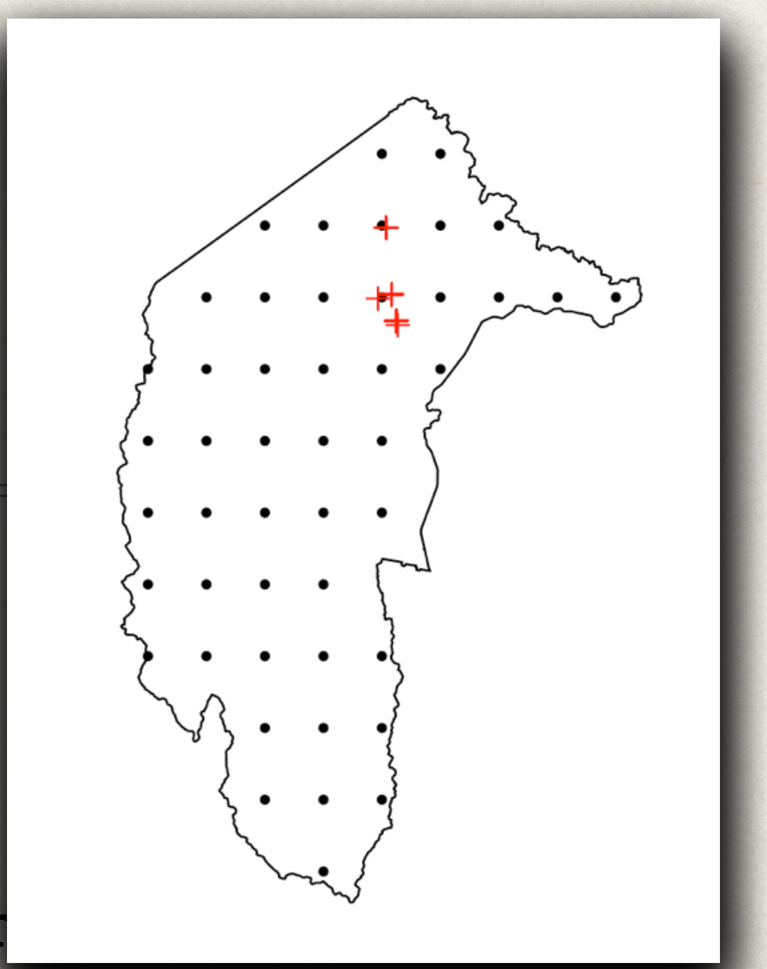
Create a polygon (square) for each grid point

(Calculate interesting attributes of the polygons if desired)

(Crop the polygons if desired)

Draw the polygons

Method



```
grid <- makegrid(poly, n = 1000)
cellsize <- sp::spDists(data.matrix(grid[1,]
                        data.matrix(grid[2,]

grid.points <- sp::SpatialPointsDataFrame(gr
    data.frame(n=1:nrow(grid)),
    proj4string = CRS(proj4string(poly)))
grid.points <- grid.points[poly, ]

distance.matrix <- sp::spDists(grid.points, features)
grid.points$furthest = apply(distance.matrix, 1, max)

grid.squares <- rgeos::gBuffer(grid.points,
    width = cellsize / 2,
    quadsegs = 1,
    capStyle = "SQUARE",
    byid = TRUE)
```

Distance matrix

	f1	f2	f3	f4	
p1	34	34	36	37	34
p2	52	54	60	61	55
p3	65	68	76	75	68
p4	22	20	21	21	20
p5	20	17	10	11	17

Distance matrix

	f1	f2	f3	f4	
p1	34	34	36	**37**	34
p2	52	54	60	**61**	55
p3	65	68	**76**	75	68
p4	**22**	20	21	21	20
p5	**20**	17	10	11	17

Distance matrix

	furthest
p1	37
p2	61
p3	75
p4	22
p5	20

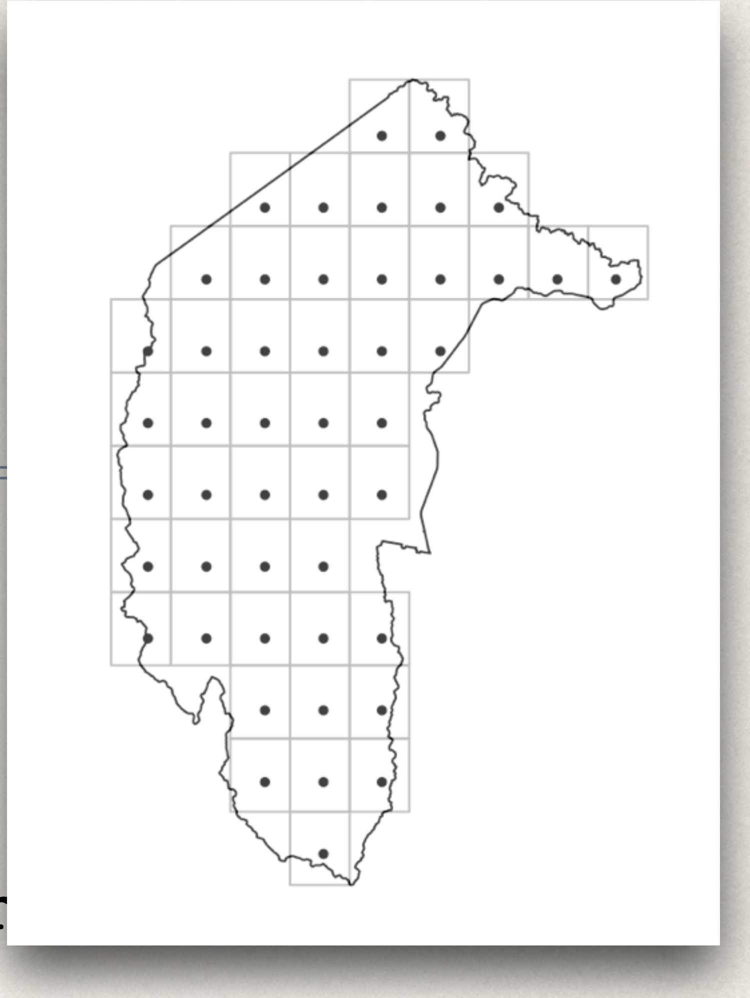
Method

```
grid <- makegrid(poly, n = 1000)
cellsize <- sp::spDists(data.matrix(grid[1,]
                        data.matrix(grid[2,]

grid.points <- sp::SpatialPointsDataFrame(gr
    data.frame(n=1:nrow(grid)),
    proj4string = CRS(proj4string(poly)))
grid.points <- grid.points[poly, ]

distance.matrix <- sp::spDists(grid.points, features)
grid.points$furthest = apply(distance.matrix, 1, max)

grid.squares <- rgeos::gBuffer(grid.points,
    width = cellsize / 2,
    quadsegs = 1,
    capStyle = "SQUARE",
    byid = TRUE)
```



Colour the polygons

This is exactly the same as the Chapter 7 population density example

- create a colour palette function
- draw the polygons and colour according to the value of furthest using the palette function

leafletProxy

A *leafletProxy* can be used in an observer to respond when the map changes

```
output$map <- renderLeaflet({
  # Use leaflet() here, and only include aspects
  # of the map that won't need to change dynamically
  # (at least, not unless the entire map is being
  # torn down and recreated).
  leaflet(quakes) %>% addTiles() %>%
    fitBounds(~min(long), ~min(lat),
              ~max(long), ~max(lat))
})
```

leafletProxy

```
# Incremental changes to the map (in this case, replacing
# the circles when a new colour is chosen) should be
# performed in an observer. Each independent set of things
# things that can change should be managed in its own
# observer.
observe({
  pal <- colorpal()

  leafletProxy("map", data = filteredData()) %>%
    clearShapes() %>%
    addCircles(radius = ~10^mag/10, weight = 1,
      color = "#777777", fillColor = ~pal(mag),
      fillOpacity = 0.7, popup = ~paste(mag)
    )
})
```

leafletProxy

```
# Use a separate observer to recreate the legend
# as needed

observe({
  proxy <- leafletProxy("map", data = quakes)

  # Remove any existing legend, and only if the legend is
  # enabled, create a new one.
  proxy %>% clearControls()
  if (input$legend) {
    pal <- colorpal()
    proxy %>% addLegend(position = "bottomright",
                        pal = pal, values = ~mag
    )
  }
})
```

Layer ids

Layer ids can be used to control and replace specific map elements

When adding an object with a layer id, existing objects with the same id are removed

Layer ids are a vector, one per object

- if you give a singleton all the objects have the same layer id and thus each will be remove as a subsequent one is added leaving only the final object

Layer ids must be unique by category

Layer ids

Category	Add function	Remove	Clear
tile	addTiles, addProviderTiles	removeTiles	clearTiles
marker	addMarkers, addCircleMarkers	removeMarker	clearMarker
shape	addPolygons, addPolylines, addCircles, addRectangles	removeShape	clearShapes
geojson	addGeoJSON	removeGeoJSON	clearGeoJSON
topojson	addTopoJSON	removeTopoJSON	clearTopoJSON
control	addControl	removeControl	clearControls

Leaflet callbacks

We can arrange to have *leaflet* tell R when the map view changes

As the user moves around or zooms in and out we can receive events

This allows us to update the map

We can also optimise our map drawing

Leaflet input events

`input$MAPID_OBJCATEGORY_EVENTNAME`

Clicking on a circle on *mymap* would update `input$mymap_shape_click`

The value is a list which includes

- lat — latitude
- lng — longitude
- id — the layer id (if any)

For GeoJSON events

- featureId
- properties

Leaflet input events

`OBJCATEGORY`

- marker
- shape
- geojson
- topojson

`EVENTNAME`

- click
- mouseover
- mouseout

Leaflet map events

`input$MAPID_click` — when map is clicked, named list

- `lat` and `lng`

`input$MAPID_bounds` — the currently visible map area, named list

- `north`, `east`, `south` and `west`

`input$MAPID_zoom` — an integer indicating *zoom level*

`input$MAPID_center` — a list of the centre of the map named list

- `lat` and `lng`

Optimising map drawing

The number of points (and thus the number of polygons) affects the render time

More points means more detail and the ability to zoom in

More points also means slower render times

A value of about 10000 for n is manageable but we can’t zoom in

We want to be able to calculate our points and polygons based on the map bounding box and any polygon in question

This way we don’t end drawing polygons outside the viewable area