GEODE USER'S MANUAL[1]

2021-03-31

Contents

[1] A note on this version of this manual: as much as possible, formatting has been adjusted to accommodate PDF style; however, line and paragraph formatting of R source code may not exactly match what is used in an actual R session. The html version of this manual is preferred for copying-and-pasting R code directly into an R session.

Geode User's Manual
Author: Michael C. Otterstatter

1 Background

1.1 Purpose

Geode is a software tool for mapping and analysis of geospatial data. It was designed in consultation with partners across the Canadian provincial and territorial cancer agencies.

Development of geode was motivated by a need for local information on cancer screening. The tool is intended to support the work of jurisdictions as they map and analyze cancer screening data, but is flexible enough to allow for a wide variety of geospatial work. The tool supports mapping and analysis of any geospatial boundary data available via shapefiles, any point locations identifiable as x-y positions (e.g., longitude and latitude), and any measures that can be assigned to regions (e.g., population size and density, age and sex distribution, counts and rates of disease or health conditions, socio-economic indicators, environmental health measures, etc.).

Users of this tool are expected to have a basic understanding of the characteristics, limitations and use of geospatial data and the basic methods used in the analysis of such data.

1.2 System requirements

The geode package was written in the freely available, open source software **R** (https://cran.r-project.org/). The tool is intended to be used with R version 4.0.0 (April 2020) or later and RStudio (https://rstudio.com/) version 1.2.1335 or later. *Earlier versions of R will not support all of the functionality of this tool and must be updated prior to installation.*

It should be noted that R is an 'in-memory' software, such that all data work is carried out in RAM. This has the benefit of enhanced performance, but can be an important consideration when working with large datasets. Users will need to ensure that their machines running R have sufficient RAM to accommodate the spatial datasets they wish to analyze.

A number of R packages are used by geode, including **sf**, **tmap**, **ggplot2**, and **spdep**. The appropriate versions of all required packages will be installed if you follow the instructions provided here. However, users must have read/write access to a local or network folder for installing and updating R packages.

This tool is designed to be compatible with the essential data manipulation functions of R, particularly those from the **tidyverse** packages **dplyr** and **tidyr**, so that it integrates seamlessly with R workflows.

1.3 Downloading and installing

In order to get started with geode, you must first install the package on your computer. Note that installation may take only a few minutes (if you already have most of the required R libraries), but could take half-an-hour or more (e.g., if you are starting from a newly downloaded version of R).

Installation can either be done directly in your R session, by downloading the package from GitHub:

```
install.packages("remotes")
remotes::install_github("cpacc/geode")
```

Or, you can install from the geode zip file following these two steps:

First, save the zip file to your computer and unzip using e.g., WinZip, 7-Zip, etc.

Second, update the filepath shown below (`my_libpath`) so that it points to the location on your computer where the geode source file ("geode_0.1.1.tar.gz") is saved. Run the updated lines of code in your RStudio console and geode will be installed.

```
# specify current location of geode source file on your computer
my_libpath <- "C:/Users/Michael/Documents/R/working dir/geode_prototype/R"
package_name <- "geode_0.1.1.tar.gz"


# install geode
install.packages(paste(my_libpath, package_name, sep = "/"),
                 repos = NULL, type = "source")
```

Regardless of whether you install from Github or from the zip file, this process will also install other R packages ('dependencies') used by geode from an online repository. It may take several minutes to install all dependencies and you may be asked i) if you want to update certain packages that have more recent versions available (recommended, but probably not essential – this could take a while if many packages need updating), and ii) if you wish to install from sources the packages which need compilation (recommended and probably essential in order to get the appropriate version of the required packages).

Note that installation of geode and its dependencies only needs to be done the first time you use geode, or if there are any updates released to the package itself.

After installation, you should see something like the following in your RStudio console window:

```
* installing *source* package 'geode' ...
** using staged installation
** R
** byte-compile and prepare package for lazy loading
** help
*** installing help indices
```

```
  converting help for package 'geode'
    finding HTML links ... done
    geo_calculate                           html
    geo_detect                              html
    geo_distance                            html
    geo_export                              html
    geo_import                              html
    geo_plot                                html
    pipe                                    html
** building package indices
** testing if installed package can be loaded from temporary location
*** arch - i386
*** arch - x64
** testing if installed package can be loaded from final location
*** arch - i386
*** arch - x64
** testing if installed package keeps a record of temporary installation path
* DONE (geode)
```

Now that it's installed, you only need to load the geode package for use when you start a new R session.

```
library(geode)
```

Before working with your own data, it is strongly recommended that you try the example in Chapter 2 (see *2.2 Getting started*) and some of the examples found later in Chapter 4.

1.4 Help files

If geode installed and loaded correctly, you will be able to access the help files supporting the available functions. Each help file contains a brief description of the function itself and an overview of the syntax needed to use the function properly. Help files also contain simple examples of how a function may be used.

For example, try running the help commands below in the console window of RStudio – help text should appear in the Help window at bottom right of your RStudio session.

Access the function for importing spatial data:

```
?geo_import
```

Access the function for plotting maps:

```
?geo_plot
```

Access the function for calculating spatial statistics/generating statistical maps:

```
?geo_calculate
```

Access the function for identifying spatial clusters:

```
?geo_detect
```

Access the function for calculating and mapping proximity:

```
?geo_distance
```

Access the function for exporting data:

```
?geo_export
```

Complete help documentation is also included in Chapter 5 of this manual.

1.5 Boundary files and sample data sets

The geode package can work with any geospatial data based on shapefiles (.shp) and corresponding attribute data from comma separated value files (.csv). The examples provided in this user's manual make use of publicly available geospatial datasets, as described below.

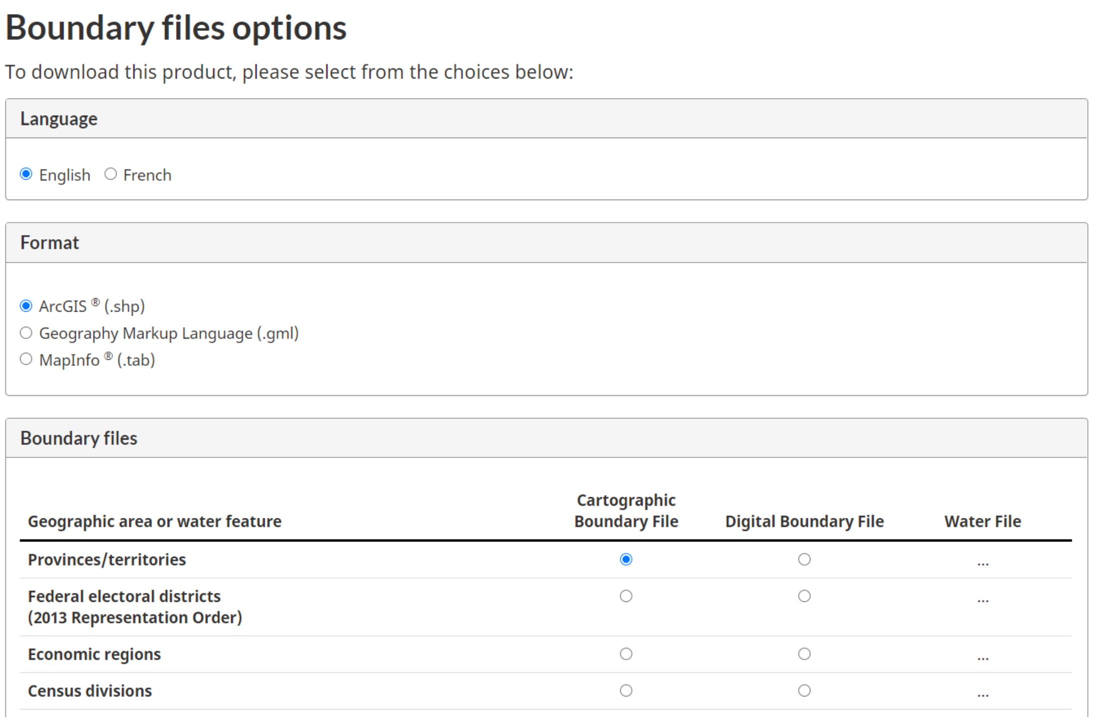
A wide range of geographic data are freely available from Statistics Canada. A helpful reference page, containing links to geographic boundary files, maps, attribute data and references, can be found at:

https://www12.statcan.gc.ca/census-recensement/2016/geo/index-eng.cfm

In particular, geographic boundary files from the Canadian Census are available at:

https://www12.statcan.gc.ca/census-recensement/2011/geo/bound-limit/bound-limit-eng.cfm

There are two ways to obtain these boundary files. First, users can select a census year and then use the point-and-click options to define a boundary file for download. In geode we exclusively use shapefiles (.sph) to define geographic boundaries – the Statistics Canada website identifies these as "ArcGIS" files.


Boundary files options

To download this product, please select from the choices below:

Language

[✓ English] French

Format

[✓ ArcGIS ® (.shp)]
Geography Markup Language (.gml)
MapInfo ® (.tab)

Boundary files

Geographic area or water feature	Cartographic Boundary File	Digital Boundary File	Water File
Provinces/territories	●	○	...
Federal electoral districts (2013 Representation Order)	○	○	...
Economic regions	○	○	...
Census divisions	○	○	...

After clicking 'Continue,' a download link will appear allowing you to save the selected boundary file as a zip folder (there are also instructions provided on how to open zip files). Once you have saved and unzipped the file, it can be directly used by geode: simply specify the folder path for the shapefile in the `geo_import()` function.

Alternatively, users can download and unzip boundary files directly in R. For example, imagine we have used the point-and-click options on the above Statistic Canada webpage to select a shapefile (format 'ArcGIS') for the Canadian Provinces/territories. When we click 'Continue' a download link appears for our file: 'lpr_000b16a_e.zip (ZIP version, 27,960.0 kb).' In Windows, we can right-click on this link and copy the link address so that it can be pasted into our R script for further use.

In the example code below, we use this link address as our 'source_path' and also define the 'destination_path' on our computer where the zip file should be saved. It is then simply a matter of using the R functions `download.file()` and `unzip()` to retrieve the shapefile from the Statistics Canada website and unzip the folder so that we can import the boundaries using geode.

```r
source_path <- "https://www12.statcan.gc.ca/census-
recensement/2011/geo/bound-limit/files-fichiers/2016/lpr_000b16a_e.zip"


# change to your download folder path
destination_path <- "C:/Users/Michael/Documents/R/data/lpr_000b16a_e.zip"

download.file(url = source_path, destfile = destination_path)

unzip(zipfile = destination_path)
```

Large shapefiles (e.g., dissemination blocks for all of Canada) may take several minutes to download. Fortunately, you only need to download and unzip a shapefile once.

Many of the example datasets used in this manual are from the Statistics Canada website. In particular, geographic boundary data files were obtained from:

https://www12.statcan.gc.ca/census-recensement/2011/geo/bound-limit/bound-limit-2016-eng.cfm

and corresponding geographic attributes were obtained from:

https://www12.statcan.gc.ca/census-recensement/2011/geo/ref/att-eng.cfm

Specific data for British Columbia was obtained from DataBC at

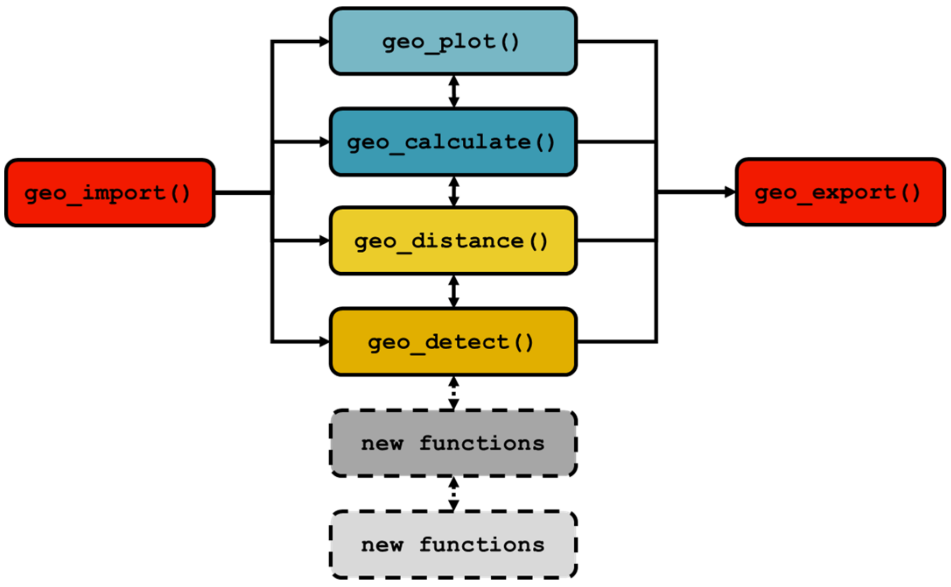
https://catalogue.data.gov.bc.ca/dataset

2 Using geode

In this chapter, we briefly review the structure of the **geode** software tool and then work through an initial example to help new users get started.

2.1 Structure of the tool

The **geode** tool is designed as a set of compatible functions that can be used individually or together to map and analyze geospatial data.



In the current version the core functions are

`geo_import()` for importing geographic boundary data and attribute data
`geo_plot()` for plotting maps
`geo_calculate()` for calculating (and mapping) spatial statistics
`geo_distance()` for running distance-based proximity analysis
`geo_detect()` for detecting significant spatial clusters
`geo_export()` for exporting spatial and attribute data to other file formats.

The design of **geode** is flexible and allows the easy addition of new functions in subsequent versions of the tool.

Important details on each function can be found in Chapter 3, and in the associated help documentation in Chapter 5, and through the available help files in R. Users should review these details to ensure a full understanding of geode output.

2.2 Getting started

After installing the **geode** package, it is recommended that new users work through the following example in order to become acquainted with the basic usage of the **geode** tool. This example works through several functions quickly and is only meant to give users a general sense of how the tool works, before delving into the technical details of each function.

The example provides a brief introduction to three of the core functions: `geo_import()`, `geo_plot()` and `geo_calculate()`.

Supporting information, such as details of the specific syntax for each function, is found in the help files as described below.

Note that in this example only the **geode** package is loaded by default. Hence, any functions that are used from other R packages are always called explicity using the syntax package_name::function_name. For example, the filter function from the tidyverse package **dplyr** is called in the script using `dplyr::filter()`.

Users are free to load other packages to support additional data manipulation if desired.

2.2.1 Setup

First, begin by loading the geode package using the R function `library()`.

```
library(geode)
```

It is good practice to define any needed folder paths (e.g., input or output locations) or data files at the beginning of your script. Here, we define an input directory and the names of two data files. These files are provided with the **geode** package, but originate from freely available Statistic Canada datafiles.

Note that in R, all file and folder paths must be specified using forward slashes '/.'

```
# location of input data (modify as needed for your computer)
indir <- "C:/Users/Michael/Documents/R/working dir/geode_prototype/data"

# names of data files to be imported
shape_file_da <- "geode_data_spatial_da.shp"
source_file_pop <- "geode_data_pop_da.csv"
```

2.2.2 Importing data with `geo_import()`

To begin, see the help file for `geo_import()`, which describes it's usage, arguments and further details. Whenever you have questions about the use of this function, call the help file.

```
?geo_import
```

Try importing the spatial data file we defined above ('shape_file_da') using the following code. Shapefiles can be optionally simplified during import, which creates a smaller, more efficient, working file (but note that simplified files are usually only appropriate for experimental use).

```r
# shapefile not simplified on import
my_dat <- geo_import(path = paste(indir, shape_file_da, sep = "/"),
                     filetype = 'spatial',
                     simplify = FALSE)

# shapefile simplified on import
my_dat_simple <- geo_import(path = paste(indir, shape_file_da, sep = "/"),
                     filetype = 'spatial',
                     simplify = TRUE)
```

After importing, inspect the variables present in the data. Geographic boundary data should always contain a 'geometry' column.

```r
dplyr::glimpse(my_dat)
```

How big are the imported shapefiles? This is important to know, as all imported files are brought into memory and may affect the performance of your computer if RAM is limited. Check using the R function `object.size()`.

```r
object.size(my_dat) # original DA file ~ 250MB
object.size(my_dat_simple) # simplified DA file ~ 75MB
```

Now, import the attribute datafile we defined above ('source_file_pop'). Attribute data contains characteristics of the spatial units, such as population size.

```r
my_pop <- geo_import(path = paste(indir, source_file_pop, sep = "/"),
                     filetype = 'attribute')
```

Inspect variables in attribute data.

```r
dplyr::glimpse(my_pop)
```

2.2.3 Creating maps with `geo_plot()`

To begin, see the help file for `geo_plot()`, which covers it's usage, arguments and further details. Whenever you have questions about the use of this function, call the help file.

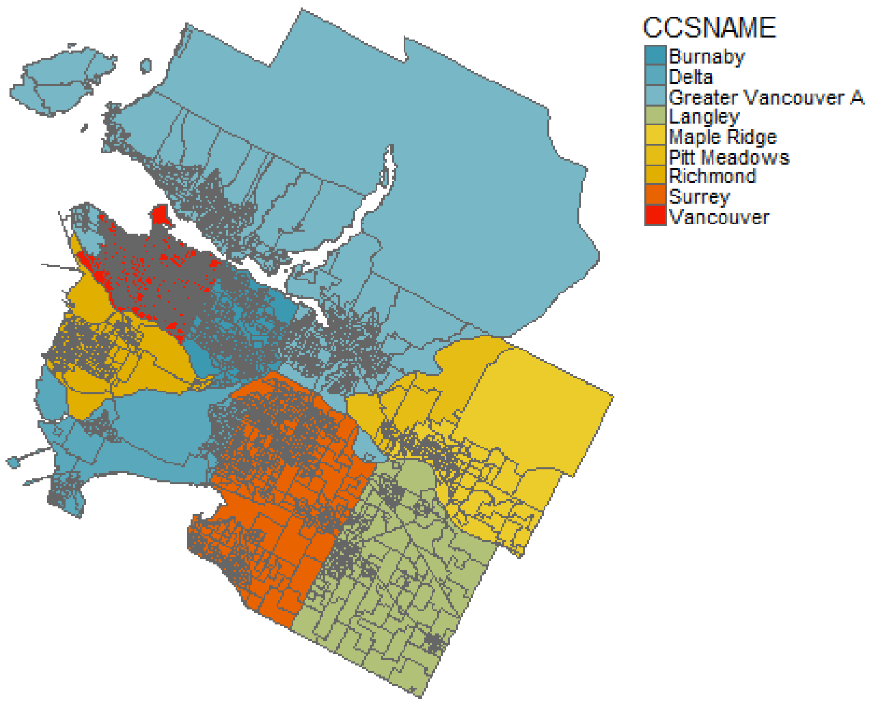
```r
?geo_plot
```

Subset the data for plotting. Here we filter to include only the Census Division (variable CDNAME) of Greater Vancouver.

```r
my_plot_dat <- my_dat %>%
  dplyr::filter(CDNAME == 'Greater Vancouver')
```

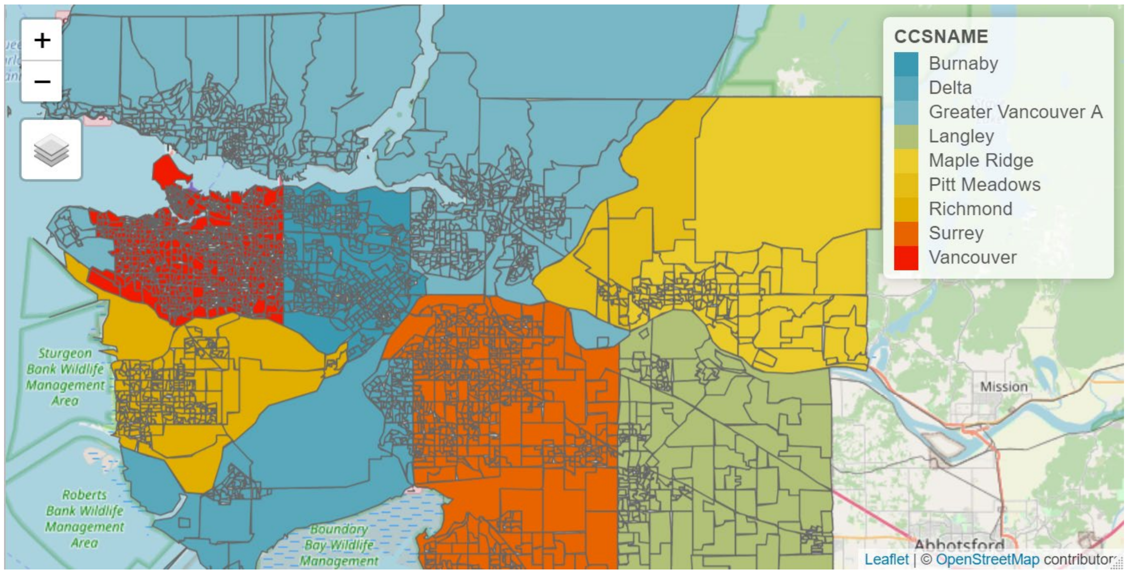
Generate a static map using the default settings (i.e., specify only required arguments)

```
geo_plot(data = my_plot_dat,
         geography_col = CCSNAME,
         plot_type = 'choropleth')
```



Generate an interactive map using the default settings (i.e., specify only required arguments). With interactive maps, users can turn background layers on/off, scroll in any direction, zoom in/out, and hover over areas to obtain further information. (Note that these background layers are loaded from an online repository, hence users must have an active internet connection).

```
geo_plot(data = my_plot_dat,
         geography_col = CCSNAME,
         plot_type = 'choropleth',
         interactive = TRUE)
```

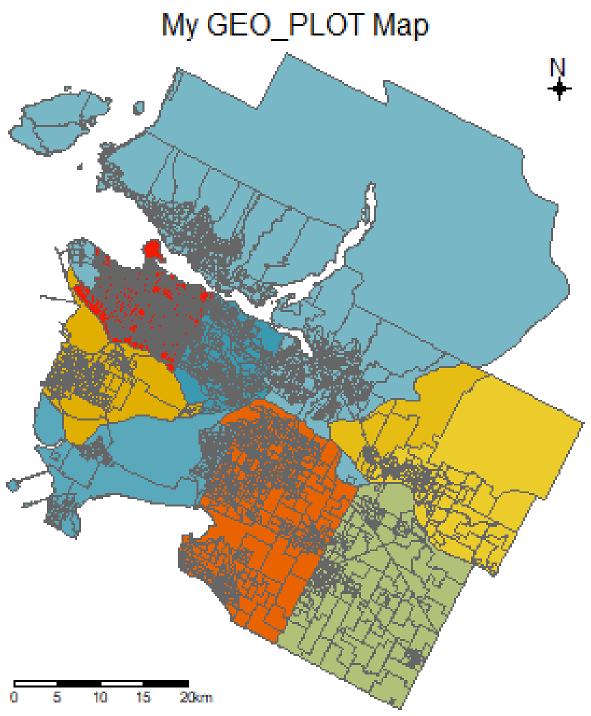

CCSNAME
Burnaby
Delta
Greater Vancouver A
Langley
Maple Ridge
Pitt Meadows
Richmond
Surrey
Vancouver
Sturgeon Bank Wildlife Management Area
Roberts Bank Wildlife Management Area
Boundary Bay Wildlife Management
Mission
Abbotsford
Leaflet | © OpenStreetMap contributors

Also note that the information displayed when hovering over a region in an interactive map can be specified using the `hover_id` argument. By default, **geode** displays the name/value contained in the column specified by the `geography_col` argument. However, any column in the input dataset can be specified by `hover_id`, giving users the option to display, for example, the regional population size or event rate of interest.

If more visibility of the underlying reference layer is desired, the shading of the choropleth map can be adjusted using the `transparency =` argument, with values between 0 (transparent) and 1 (opaque).

Next, generate a static map including optional settings (e.g., include plot title, scale and compass, but drop legend).

```
geo_plot(data = my_plot_dat,
  geography_col = CCSNAME,
  plot_type = 'choropleth',
  plot_title = 'My GEO_PLOT Map',
  legend_title = 'none' ,
  scale_bar = TRUE,
  compass = TRUE,
  interactive = FALSE)
```



Finally, export the plot for later use. First create the plot as an R object 'my_plot.'

```
my_plot <- geo_plot(data = my_plot_dat,
                    geography_col = CCSNAME,
                    plot_type = 'choropleth',
                    plot_title = 'My GEO_PLOT Map',
                    legend_title = 'none' ,
                    scale_bar = TRUE,
                    compass = TRUE,
                    interactive = FALSE)
```

Then, export as an image file:

```
# specify output location and file name for plot
outdir <- "C:/Users/Michael/Documents/R/working dir/output"
outfile <- "my_plot.png" # note file extension .png

# exporting as an image using png() function
# 4" x 6" image, resolution 300 dpi
png(filename = paste(outdir, outfile, sep = "/"),
    height = 4, width = 6, units = "in", res = 300)

print(my_plot)
dev.off()
```

Here we used the `png()` function to export a .png file. Other available functions, with similar syntax, can be used to export images: `bmp()`, `jpeg()` and `tiff()`.

Or, export as a pdf:

```
# specify output location and file name for plot
outdir <- "C:/Users/Michael/Documents/R/working dir/output"
outfile <- "my_plot.pdf" # note file extension .pdf

pdf(file = paste(outdir, outfile, sep = "/"),
    width = 8.5, height = 11) # 8.5" x 11" output size

print(my_plot)
dev.off()
```

For additional information and options available when exporting plots as images or pdf, see the corresponding help files `?png` or `?pdf`.

A note about map colour styles: When generating maps, users can select from a range of predefined colour palettes or define their own (see style argument under `?geo_plot`). The above examples were generated using the built-in 'zissou' style, but users are encouraged to examine other colour schemes or to create their own.

2.2.4 Generating spatial statistics with `geo_calculate()`

To begin, see the help file for `geo_calculate()`, which covers it's usage, arguments and further details. Whenever you have questions about the use of this function, call the help file.

```
?geo_calculate
```

Import the provided BC data with Dissemination Areas (DAs) and population sizes:

```
bc_dat <- geo_import(path = paste(indir, 'geode_data_bc_da_pop.shp', sep =
"/"), filetype = 'spatial')
```

Subset data to only DAs in Census Division (variable CDNAME) of Central Okanagan:

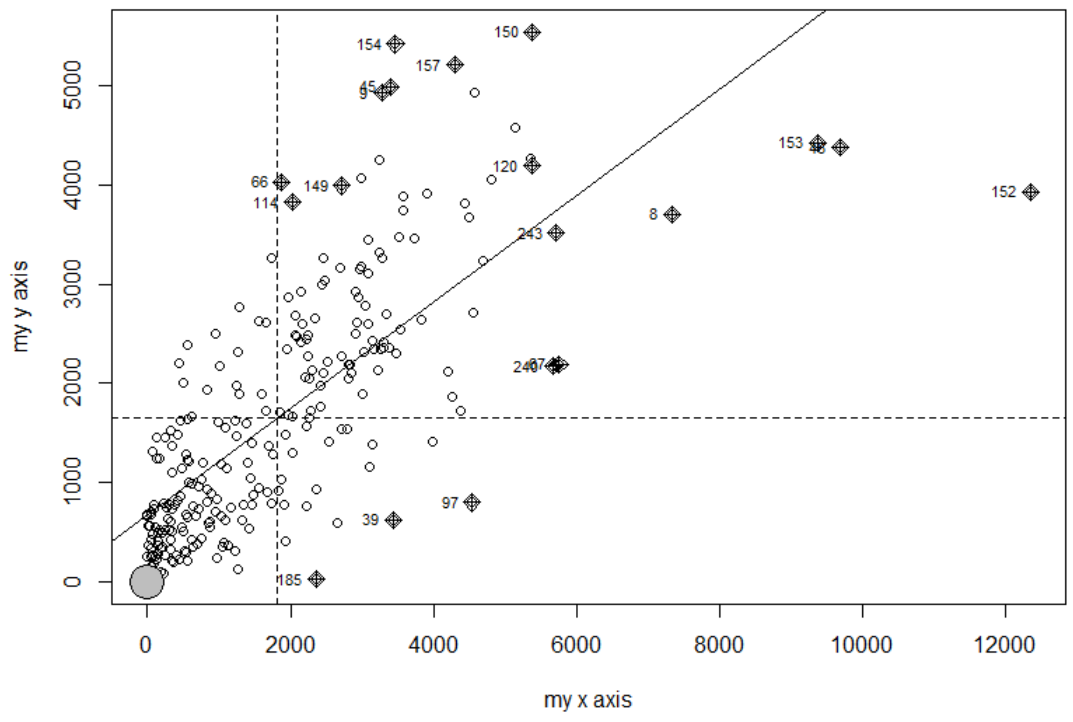
```
OK_dat <- bc_dat %>%
  dplyr::filter(CDNAME == 'Central Okanagan')
```

First, we test for global spatial autocorrelation and generate a standard statistical plot (Moran's I scatterplot). As an illustration, we run our analysis on population density (variable 'density'):

```
geo_calculate(data = OK_dat,
              var = density,
              statistic = 'global_ac',
              create_plot = TRUE)
```

Here, `geo_calculate()` ouputs the test statistic and p-value associated with the test for global spatial autocorrelation:

```
  statistic                               simulations test_statistic p_value
  <chr>                                         <int>          <dbl>   <dbl>
1 Monte-Carlo simulation of Moran I             10000          0.537  0.0001
```



Next, we calculate local spatial autocorrelation and generate an output dataset. Again, we use the population density variable for this analysis, as an illustrative example.

```
localg <- geo_calculate(data = OK_dat,
              var = density,
              statistic = 'local_ac',
              create_plot = FALSE)

# see output file
localg

Simple feature collection with 255 features and 9 fields
geometry type:  MULTIPOLYGON
dimension:      XY
bbox:           xmin: 4245014 ymin: 1925870 xmax: 4333571 ymax: 1993292
projected CRS:  PCS_Lambert_Conformal_Conic
First 10 features:
      DAUID                                 PRNAME          CDNAME CSDNAME
pop dwellings density
1  59350077 British Columbia / Colombie-Britannique Central Okanagan Kelowna
2693      1531   140.4
```

```
2  59350078 British Columbia / Colombie-Britannique Central Okanagan Kelowna
517       251   414.0
3  59350079 British Columbia / Colombie-Britannique Central Okanagan Kelowna
736       317  2251.5
4  59350080 British Columbia / Colombie-Britannique Central Okanagan Kelowna
971       380  3383.3
5  59350081 British Columbia / Colombie-Britannique Central Okanagan Kelowna
623       225  3014.0
6  59350082 British Columbia / Colombie-Britannique Central Okanagan Kelowna
479       180  2453.9
7  59350083 British Columbia / Colombie-Britannique Central Okanagan Kelowna
363       165  3579.9
8  59350086 British Columbia / Colombie-Britannique Central Okanagan Kelowna
662       398  7339.2
9  59350087 British Columbia / Colombie-Britannique Central Okanagan Kelowna
363       215  3276.2
10 59350088 British Columbia / Colombie-Britannique Central Okanagan Kelowna
866       315  4686.1
                          geometry     localg           cluster
1  MULTIPOLYGON (((4293883 195... -2.6903611 Cold spot: 99% CI
2  MULTIPOLYGON (((4290959 194... -0.7791236   Not significant
3  MULTIPOLYGON (((4290580 194...  0.5233985   Not significant
4  MULTIPOLYGON (((4290653 194...  1.1537163   Not significant
5  MULTIPOLYGON (((4290245 194...  0.3777174   Not significant
6  MULTIPOLYGON (((4289926 194...  2.0569677  Hot spot: 95% CI
7  MULTIPOLYGON (((4289680 194...  2.8679259 Hot spot:  99% CI
8  MULTIPOLYGON (((4289142 194...  3.5445971 Hot spot:  99% CI
9  MULTIPOLYGON (((4289005 194...  4.0291994 Hot spot:  99% CI
10 MULTIPOLYGON (((4288792 194...  2.4985473  Hot spot: 95% CI
```

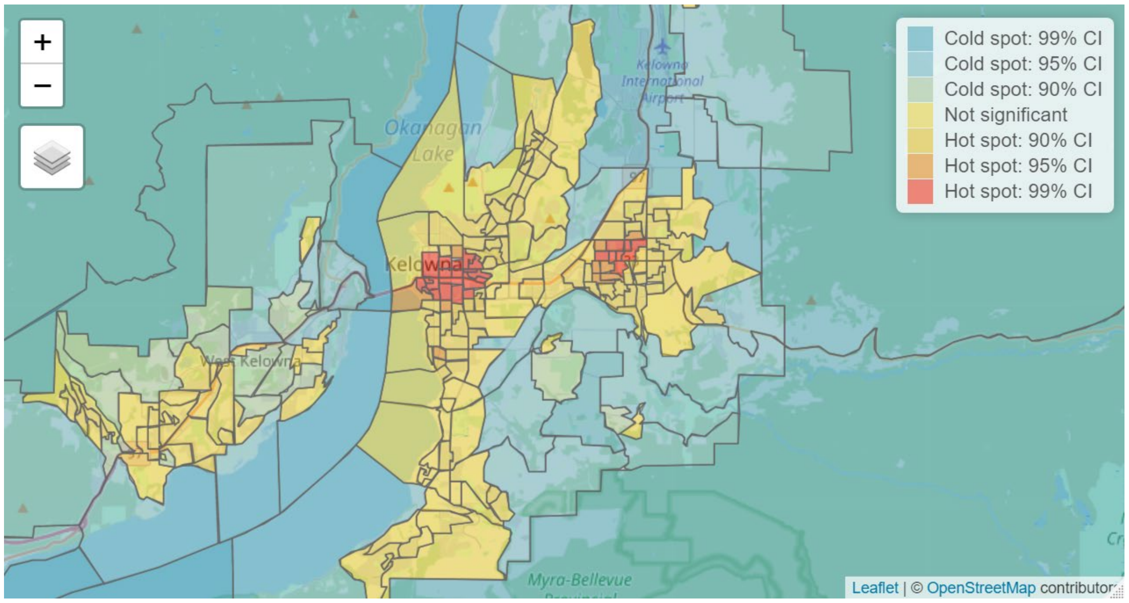
We see that the output includes each of the regions present in our analysis dataset, but now the test statistic ('localg') and significance category ('cluster') have been added.

We can export this output data for later use, using the geo_export() function:

```
# output path and filename
outfile <- "C:/Users/Michael/Documents/R/working dir/output/my_local_ac"

# export as shapefile (because output still contains geometry data)
geo_export(x = localg,
           path = outfile,
           filetype = "shp")
```

Finally, we calculate local spatial autocorrelation but now also generate an interactive map showing the significant hotspots and coldspots

```
geo_calculate(data = OK_dat,
              var = density,
              statistic = 'local_ac',
              create_plot = TRUE)
```



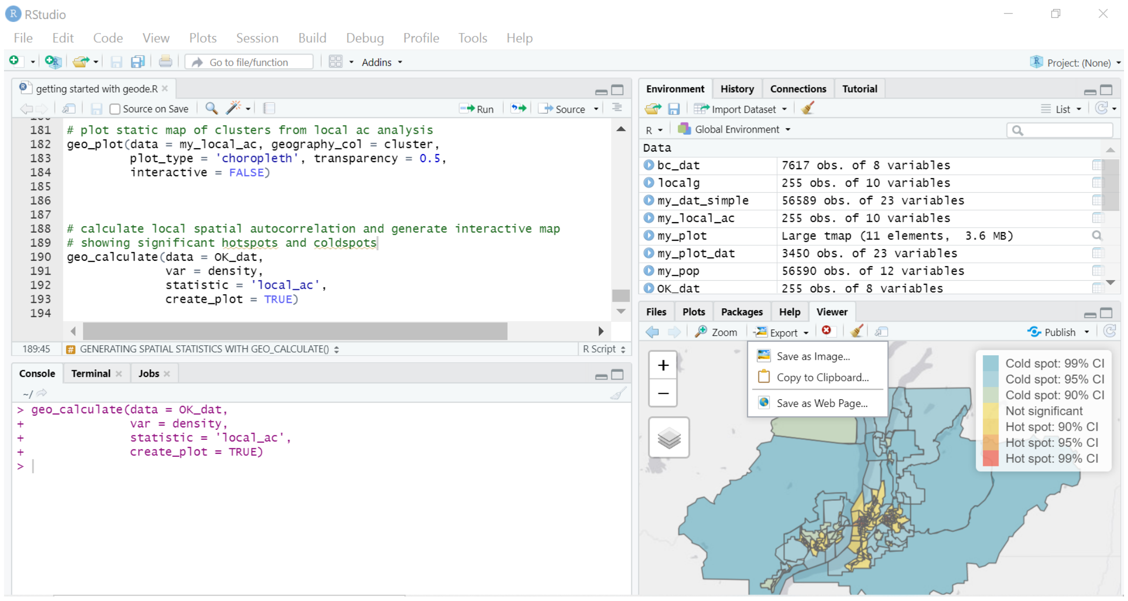
As with `geo_plot()` the transparency of the choropleth map can be adjusted to reveal more or less of the background reference layer (using the `transparency =` argument), and the information shown when hovering over a region can be modified (any column in the input dataset can be specified with the `hover_id =` argument).

2.2.5 Additional information on exporting output

The **geode** package contains a specific function, `geo_export()`, for saving spatial and attribute data to a file. More information on this function can be found in chapters 3 and 5 and examples can be found in chapter 4. Note that this function is only for exporting data, not plots.

Above, we provided an example of how plots and maps generated in **geode** can be exported as images using the `png()` function, or as pdfs using the `pdf()` function.

Additionally, it is worth noting that plots and maps can be exported manually using the RStudio interface. The lower right window of RStudio has a 'Plots' tab (for static maps) and 'Viewer' tab (for interactive maps), under which is an 'Export' option with drop-down menu:


RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
Project: (None)
getting started with geode.R
Source on Save
Run
Source
Environment History Connections Tutorial
Import Dataset
List
Global Environment
Files Plots Packages Help Viewer
Zoom
Export
Publish
Save as Image...
Copy to Clipboard...
Save as Web Page...
Console Terminal Jobs
189:45
GENERATING SPATIAL STATISTICS WITH GEO_CALCULATE()
R Script

```
# plot static map of clusters from local ac analysis
geo_plot(data = my_local_ac, geography_col = cluster,
         plot_type = 'choropleth', transparency = 0.5,
         interactive = FALSE)



# calculate local spatial autocorrelation and generate interactive map
# showing significant hotspots and coldspots
geo_calculate(data = OK_dat,
              var = density,
              statistic = 'local_ac',
              create_plot = TRUE)
```

```
~/
> geo_calculate(data = OK_dat,
+              var = density,
+              statistic = 'local_ac',
+              create_plot = TRUE)
>
```

```
Data
bc_dat          7617 obs. of 8 variables
localg          255 obs. of 10 variables
my_dat_simple   56589 obs. of 23 variables
my_local_ac     255 obs. of 10 variables
my_plot         Large tmap (11 elements,  3.6 MB)
my_plot_dat     3450 obs. of 23 variables
my_pop          56590 obs. of 12 variables
OK_dat          255 obs. of 8 variables
```

🟦	Cold spot: 99% CI
🟦	Cold spot: 95% CI
🟩	Cold spot: 90% CI
🟨	Not significant
🟨	Hot spot: 90% CI
🟧	Hot spot: 95% CI
🟥	Hot spot: 99% CI

Once a map has been generated in the 'Plots' window (or the Viewer window, if the map is interactive), users can choose from this menu to export the map as an image, copy it to the clipboard (and then paste into another program), or even save it as a webpage (if it is an interactive map). All of these options open a preview screen that allows you to manually adjust the image size, choose the output format and specify the output location.

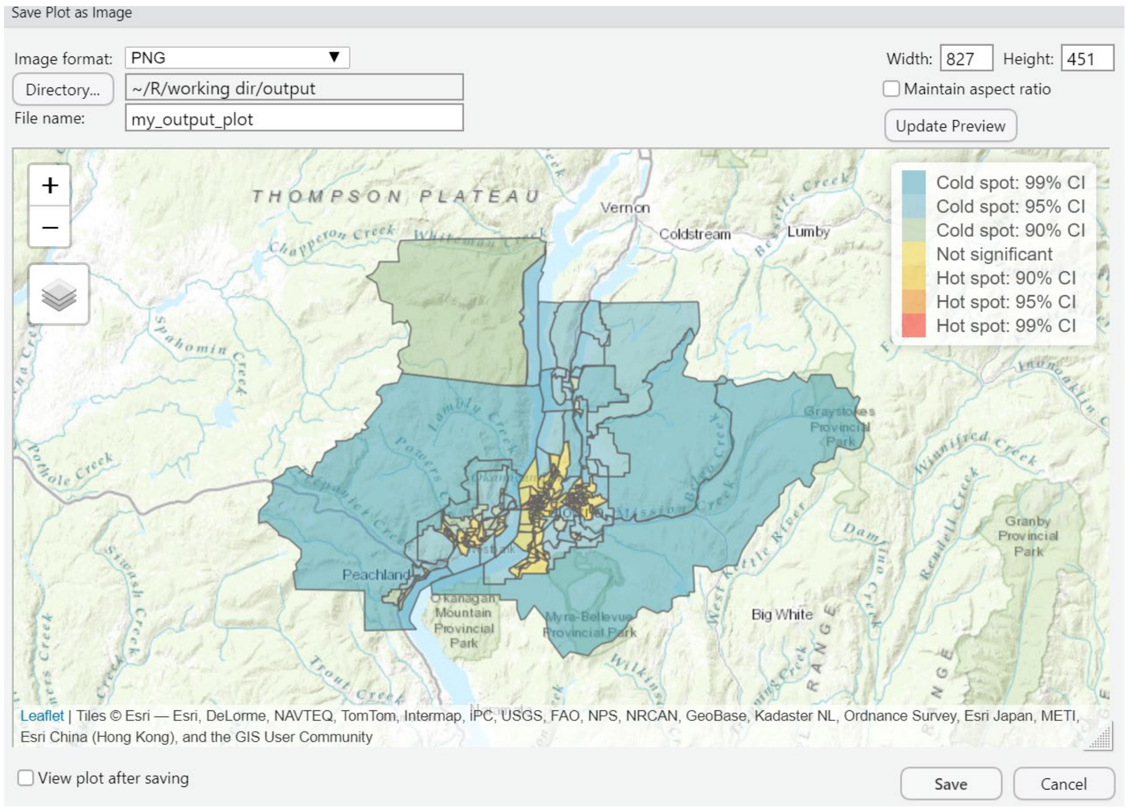


Image format: PNG

Directory... ~/R/working dir/output

File name: my_output_plot

Width: 827 Height: 451

☐ Maintain aspect ratio

🟦	Cold spot: 99% CI
🟦	Cold spot: 95% CI
🟩	Cold spot: 90% CI
🟨	Not significant
🟨	Hot spot: 90% CI
🟧	Hot spot: 95% CI
🟥	Hot spot: 99% CI

Leaflet | Tiles © Esri — Esri, DeLorme, NAVTEQ, TomTom, Intermap, iPC, USGS, FAO, NPS, NRCAN, GeoBase, Kadaster NL, Ordnance Survey, Esri Japan, METI, Esri China (Hong Kong), and the GIS User Community

☐ View plot after saving

Although this approach requires several manual steps and is therefore not easily reproducible, it can be useful for generating experimental figures quickly.

3 Technical details

Here, technical/statistical details and references are provided for the core functions in **geode**.

3.1 `geo_import()`

This function imports spatial or attribute data from an existing source file. Spatial files must be valid shapefiles (.shp), whereas attribute files can be any georeferenced data from a standard comma separate values (.csv) file.

Importation of shapefiles makes use of the `st_read()` function from the **sf** package. Simplification of shapefiles during import (simplify = TRUE) removes vertices, resulting is smaller but less detailed geographic boundaries. Boundaries are simplified to a level of 1000 m. This is useful for generating more manageable working files for experimental use. Although the process attempts to preserve topology, caution is needed as simplification can affect geographic calculations.

Checking the validity of each geometry (validity_check = TRUE) during import generates an output list indicating any invalid geometries and the reason(s) for invalidity. Additional information can be found in:

https://geocompr.robinlovelace.net/geometric-operations.html

In principle, invalid geometries can be corrected using the `st_make_valid` function from the sf package. For more information, see

https://r-spatial.github.io/sf/reference/valid.html

Importation of csv files makes use of the `read_csv()` function from the **readr** package, including standard default options.

3.2 `geo_plot()`

This function creates static or interactive maps from geospatial and attribute data.

Choropleth mapping makes use of the functionality from the **tmap** package. Point- and heat-maps make use of the mapping functionality from the **ggplot2** package.

Note that for point-maps and heat-maps, the coordinate reference system (CRS) for the attribute data is automatically transformed to match that of input geographic data. If CRS is missing for the attribute data, it is assumed to be WGS84.

For choropleth maps, user-defined output styles (e.g. colour palettes) can be created following the examples shown in `tmap_options()` (see ?tmap::tmap_options). For illustrative purpose, a selection of styles are included based on the colour palettes of the **wesanderson** R package (zissou, royal, darjeeling, and fantasticfox). If the style= option is omitted from `geo_plot()`, the output plot will use the default style for the particular plot_type.

3.3 `geo_distance()`

This function calculates and maps proximity metrics based on known point locations and geographic units.

Proximity may be measured in a variety of ways. The methodology used in the `geo_distance()` function examines proximity relative to user-specified point locations. Given input data on a set of geographic units (e.g., census regions, health regions, etc.) and the point locations of places or events of interest, the function calculates the centroid of each geographic unit and the pairwise distances to all point locations of interest. Proximity for each geographic unit is then calculated as the average distance to the n nearest locations.

The coordinate reference system (CRS) for the point location data is automatically transformed to match that of the input geographic data. If CRS is missing for the location data, it is assumed to be WGS84.

The plotting of proximity metrics makes use of the functionality in the **ggplot2** package.

3.4 `geo_detect()`

This function helps to identify regions that form an unusual spatial aggregation of events.

Public health data are often available by administrative regions, for example health regions, census geographies, postal code areas, etc. The `geo_detect()` function focuses on detection of clusters in data organized and aggregated into such administrative regions. The function makes use of event counts and corresponding population sizes provided by the user, for calculation of rates (events/population), and compares the observed value for a region against the expected value (as calculated, or as provided by the user) to determine significant clusters.

The Kulldorff cluster detection method is used, which creates groupings of regions by consecutively aggregating nearest-neighbour areas until a set proportion of the total population is reached. The number of cases or events in the aggregation is used to calculate the likelihood based on either a binomial or poisson model. The final output of this method is the group of neighbouring regions that is the *most likely cluster*, and any secondary clusters, with a significance measure based on Monte Carlo sampling.

The Kulldoff method can use either a binomial model (does not use expected counts) or a Poisson model (uses expected event counts provided by the user or calculated automatically). Both methods require i) geometry data (coordinates of each region's centroid), ii) the observed event counts by region, and iii) the population size to use for each region.

Note that population sizes that are very small (or zero), or very rare events, can result in a calculation error that halts the cluster detection algorithm. In these cases, users may wish to utilize the Poisson model, which is less sensitive to small counts. Alternatively, users can replace small population sizes with slightly larger values in order to allow the calculations to proceed. However, care should be taken that replacing small population sizes does not result in biased or erroneous results.

For both binomial and Poisson methods, the `geo_detect()` function specifies that a cluster can contain up to a maximum of 10% of the population size. Monte Carlo samples (N = 999) are generated for calculation of p-values and an alpha level of 0.05 is used for determining statistical significance. The output reports the single most likely cluster based on the calculated significance values, and any secondary clusters meeting the specified significance threshold. If a map output is requested, the primary and top two secondary clusters are plotted.

Further information on the Kulldorff cluster detection method used here can be found in the documentation for the **SpatialEpi** package at http://faculty.washington.edu/jonno/SISMIDmaterial/SpatialEpiVignette.pdf

It is important to recognize that statistical clusters of events may or may not be relevant to public health – the definition of a cluster must be determined on a case by case basis. Also, note that statistical considerations, including corrections for multiple comparisons, need to be examined carefully when performing cluster detection.

When interpreting the output of this function, it is important to consider the standard cautions and limitations of cluster detection, including ecologic fallacy, modifiable areal unit problem (MAUP), and the trade-off between resolution and sample size.

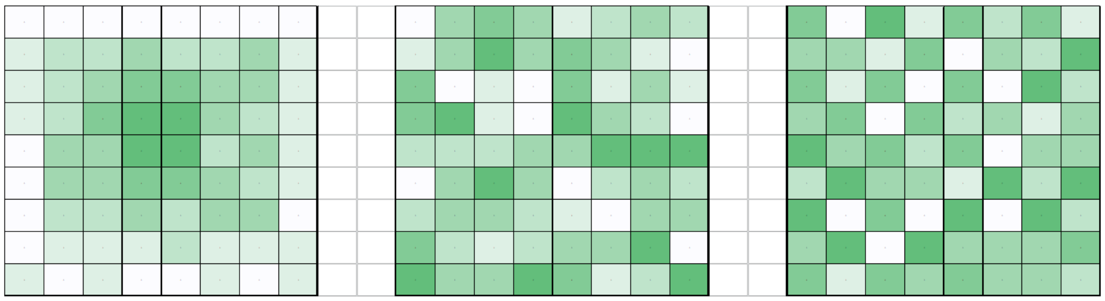
3.5 `geo_calculate()`

This function helps to measure if and how much events are aggregated in space across an area of interest. Two measures of spatial autocorrelation are used:

- global spatial autocorrelation - a measure of *clustering* used to describe spatial patterns of events across an entire study area
- local spatial autocorrelation – a measure for *detection of clusters* that indicates each region's similarity (in event frequency) to its neighbouring regions

3.5.1 global spatial autocorrelation

The calculation of global spatial autocorrelation is the cross-product of two matrices: i) values of similarity (in event counts) between each pair of regions; and ii) values indicating proximity (distance, neighbouring, etc.) between each pair of regions. High cross-product values (similar regions are close together) indicate *positive spatial autocorrelation* and small values (dissimilar regions are close together) indicate *negative spatial autocorrelation*.



Positive spatial autocorrelation **Spatial randomness** **Negative spatial autocorrelation**

The geo_calculate function uses a standard measure of global spatial autocorrelation, Moran's I, which is a spatially weighted version of the standard Pearson's correlation coefficient. See the moran.mc() function from the **spdep** package for further details.

$$I = \left(\frac{1}{s^2}\right) \frac{\sum_{i=1}^{N}\sum_{j=1}^{N} w_{ij}(Y_i - \overline{Y})(Y_j - \overline{Y})}{\sum_{i=1}^{N}\sum_{j=1}^{N} w_{ij}},$$

Global Moran's *I* statistic

Moran's I values may be negative or positive, indicating negative or positive spatial autocorrelation, respectively. This function defines spatial neighbours using the poly2nb() function from the **spdep** package and rook adjacency rules. Further, it is specified that neighbour weights must sum to 1.0 for each geographic region (style = "W") and that regions with no neighbours are ignored.

Visualization of the global autocorrelation statistic can be done in the geo_calculate() function by generating a Moran's I scatterplot (plot = TRUE).

3.5.2 local spatial autocorrelation

Note that global autocorrelation does not detect clusters *per se*, so geo_calculate() also utilizes a local measure of spatial autocorrelation to assist with finding aggregations of events and mapping of 'hotspots' and 'coldspots.'

As with global spatial autocorrelation, calculation of local autocorrelation uses similarity between a region and its neighbours, weighted by proximity of those neighbours. Generically, this may be expressed as:

$$\sum_{j=1}^{N} w_{ij}\ \mathrm{sim}_{ij}.$$

The `geo_calculate()` function uses a standard measure of location spatial autocorrelation, the local Getis-Ord *Gi* statistic.

$$G_i^* = \frac{\sum_{j=1}^{n} w_{i,j} x_j - \bar{X} \sum_{j=1}^{n} w_{i,j}}{S \sqrt{\frac{\left[n \sum_{j=1}^{n} w_{i,j}^2 - \left(\sum_{j=1}^{n} w_{i,j} \right)^2 \right]}{n-1}}}$$

In the calculation of this statistic, the function uses the same approach for determining neighbouring regions and the spatial weights matrix as describe above for global spatial autocorrelation.

Visualization of hot- and cold-spots based on the local Getis-Ord statistic can be done by generating a map in the `geo_calculate` function. The distribution of the calculated *Gi* values is used to define breaks points according to standard z-scores for levels of significance (99%, 95%, 90% CIs). These breakpoints are then mapped as hot- or cold-spots at the appropriate significance level.

It should be noted that that there are a number of challenges in determining statistical significance for local measures of spatial autocorrelation, including appropriate test distributions and problems associated with multiple (regional) testing. Further discussion of these issues can be found in Waller and Gotway, 2004 (Chapter 7).

3.6 `geo_export()`

This function exports spatial and attribute data to file. The output file type can be a shapefile (.shp), comma separated value file (.csv), Excel workbook (.xlsx), or a SAS data file (.sas7bdat).

Exporting to a shapefile makes use of the functionality in the **sf** package. Exporting to a csv file makes use of the functionality in the **readr** package. Exporting to a SAS file makes use of the functionality in the **haven** package. Exporting to EXCEL makes use of the functionality in the **xlsx** package. Further details can be found in the documentation for these packages (see https://cran.r-project.org/web/packages/available_packages_by_name.html).

Generic missing values (i.e., empty cells: "" and cells with a single empty space: " ") and properly coded missing values (*NA*) are by default converted to the character string ‘NA’ in the output dataset. Alternatively, a user-specified character string can be used instead of ‘NA’ in the output dataset (using the `na =` option).

4 Examples

Here, several worked examples of **geode** are provided to illustrate typical usage. Each example is structured as a common workflow, with setup, importation of data, plotting and analysis. The R code shown in the boxes can be copied directly to your RStudio session if you wish to run these examples yourself.

4.1 Example 1: Importing and wrangling data

4.1.1 Background

This example illustrates basic usage of the **geode** package, focusing on the importation and manipulation of data. We import geographic and attribute data using freely available Statistics Canada 2016 Census data. These data are then 'wrangled' as needed and joined together. Finally, basic plots are generated.

Note that each function in the **geode** package has an associated help file. A help file can be accessed simply by including a '?' before the name of the function of interest.

For example, we can access the help file for the `geo_import()` function using:

```
?geo_import
```

Each help file provides a simple introduction to the function and describes the necessary syntax for using that function.

The data files used in this example come from the Statistics Canada 2016 Census. These files are freely available online and can be directly downloaded (and unzipped) in R as shown below. Make sure to update the 'download_folder' path shown below to an appropriate location on your computer.

```
# specify location and filename for input shapefile data
#   Statistics Canada 2016 Census Dissemination Block boundaries
stcn_url <- "http://www12.statcan.gc.ca/census-recensement/2011/geo/bound-
limit/files-fichiers/2016"
infile_zip <- "ldb_000b16a_e.zip"
download_folder <- "C:/Users/Michael/Documents/R/data"

# specify location for input attribute data
#   Statistics Canada 2016 Census Geographic Attribute file
stcn_url_attr <- "http://www12.statcan.gc.ca/census-
recensement/2016/geo/ref/gaf/files-fichiers"
infile_attr_zip <- "2016_92-151_XBB_csv.zip"
```

These files only need to be downloaded and unzipped once.

```
# download and unzip the input data
download.file(url = paste(stcn_url, infile_zip, sep = "/"),
              destfile = paste(download_folder, infile_zip, sep = "/"))
```

```r
unzip(zipfile = paste(download_folder, infile_zip, sep = "/"),
      exdir = download_folder)

download.file(url = paste(stcn_url_attr, infile_attr_zip, sep = "/"),
              destfile = paste(download_folder, infile_attr_zip, sep = "/"))
unzip(zipfile = paste(download_folder, infile_attr_zip, sep = "/"),
      exdir = download_folder)
```

4.1.2 Setup

Load the **geode** package. This makes all **geode** functions available for use in your current R session.

```r
library(geode)
```

Before importing data, we must specify the names and locations of the shapefiles or attribute files we wish to import. It is convenient to define these as R objects at the beginning of your R script (e.g., storing the input directory path in the object 'indir') so that they can be easily modified later. As before, you must modify the input directory path ('indir') shown below to match the location on your computer where these two files have been saved.

```r
indir <- "C:/Users/Michael/Documents/R/data"
infile_shp <- "ldb_000b16a_e.shp"
infile_attr <- "2016_92-151_XBB.csv"
```

4.1.3 Data importation

We import a basic shape or attribute file using the `geo_import()` function and only two required arguments: `path` and `filetype`.

```r
# shapefile
my_shp <- geo_import(path = paste(indir, infile_shp, sep = "/"),
                     filetype = "spatial")

# attribute file
my_attr <- geo_import(path = paste(indir, infile_attr, sep = "/"),
                      filetype = "attribute")
```

From the `geo_import()` help file, we see that two optional arguments are available: `simplify` and `validity_check`. These have not been specified in the code above and are therefore set automatically at their default values (i.e., do not simplify or run a validity check during import).

We can now view our imported files using any of the standard approaches, for example:

```r
my_shp
tibble::glimpse(my_shp)
```

Shapefiles are imported as a 'simple features' data object. These data objects also contain key information about the geometry type, the bounding box dimensions and projected coordinate reference system (CRS). This information is shown when the shapefile is viewed in R.

```
Simple feature collection with 489676 features and 27 fields
geometry type:  MULTIPOLYGON
dimension:      XY
bbox:           xmin: 3689439 ymin: 659338.9 xmax: 9015737 ymax: 5242179
projected CRS:  PCS_Lambert_Conformal_Conic
First 10 features:
         DBUID DBRPLAMX DBRPLAMY PRUID
PRNAME CDUID
1  10010202002  8979444  2148775    10 Newfoundland and Labrador / Terre-
Neuve-et-Labrador  1001
2  10010203001  8979186  2149065    10 Newfoundland and Labrador / Terre-
Neuve-et-Labrador  1001
3  10010204001  8979382  2148616    10 Newfoundland and Labrador / Terre-
Neuve-et-Labrador  1001
4  10010204002  8979490  2148537    10 Newfoundland and Labrador / Terre-
Neuve-et-Labrador  1001
5  10010204004  8979255  2148525    10 Newfoundland and Labrador / Terre-
Neuve-et-Labrador  1001
6  10010204005  8979610  2148480    10 Newfoundland and Labrador / Terre-
Neuve-et-Labrador  1001
7  10010206001  8978939  2149068    10 Newfoundland and Labrador / Terre-
Neuve-et-Labrador  1001
8  10010206003  8978866  2148511    10 Newfoundland and Labrador / Terre-
Neuve-et-Labrador  1001
9  10010206004  8978899  2148301    10 Newfoundland and Labrador / Terre-
Neuve-et-Labrador  1001
10 10010206012  8978496  2148545    10 Newfoundland and Labrador / Terre-
Neuve-et-Labrador  1001
```

Note that imported shapefiles should always contain a 'geometry' column that specifies the vertices of the geographic boundaries. Attribute files can contain any information associated with the geographies in the shapefile (e.g., income, population size, etc.).

An overview of functions for manipulating simple features data in R can be found in the cheat sheet for the `sf` package:

https://github.com/rstudio/cheatsheets/raw/master/sf.pdf

We may wish to know the size (memory usage) of the imported file, particularly because shapefiles may be very large and difficult to manage. We can get file size using the R function `object.size()`.

```
object.size(my_shp)
object.size(my_attr)
```

4.1.3.1 Simplifying a shapefile during importation

We can reduce the size and memory usage of a shapefile by specifying the `simplify` option in `geo_import`. This may be helpful for generating a more manageable and efficient working file. However,it should be noted that simplification (which removes vertices from the geographic boundaries) may affect spatial calculations so should be used with care.

Also, note that importing and simplifying takes about 3x longer than importing alone.

```
my_shp_simple <- geo_import(path = paste(indir, infile_shp, sep = "/"),
                            filetype = "spatial",
                            simplify = TRUE)
```

Simplifying a shapefile usually reduces the object size by at least half, but often more.

```
# what is the size of the simplified shapefile?
object.size(my_shp_simple)
```

4.1.4 Data wrangling

Once a file has been imported, it can be cleaned, subsetted, transposed, etc. as needed and linked to other data. For most data wrangling, we use functions from the **tidyverse** package.

Further details on data wrangling functions can be found at the tidyverse website (https://www.tidyverse.org/) and in the cheat-sheets for data wrangling and data transformation:

https://rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf
https://github.com/rstudio/cheatsheets/raw/master/data-transformation.pdf

```
# if you have not already done so, first download and install the tidyverse
#  package
install.packages("tidyverse")

# now, load the package so that the functions are available for use in your R
#  session
library(tidyverse)
```

We might start, for example, by subsetting the shapefile to only one province (e.g., Ontario) and keeping only certain selected variables.

```
my_shp_on <- my_shp %>%
  filter(PRNAME == 'Ontario') %>%
  select(PRNAME, DBUID, ERNAME, FEDNAME, CMANAME, geometry) %>%
  mutate(DBUID = as.numeric(DBUID)) # reformat variable DBUID as numeric

glimpse(my_shp_on)
```

Note that the dissemination block ID variable ‘DBUID’ was originally formatted as a character variable, but we have reformatted it above to be numeric. Later, we will join the geographic and attribute data using this variable as a linkage key, so it is helpful if it has a consistent (numeric) format.

We then may wish to wrangle our attribute data. For example, we might subset the attribute file to the matching province used in our shapefile and then select certain key variables. In the 'select' function we also rename these variables to be more user-friendly and/or to match the variable names in our shapefile.

```
my_attr_on <- my_attr %>%
  filter(`PRname/PRnom` == 'Ontario') %>%
  select(DBUID = `DBuid/IDidu`,
         DBpop = `DBpop2016/IDpop2016`,
         DBtdwellings = `DBtdwell2016/IDtlog2016`,
         DBudwellings = `DBurdwell2016/IDrh2016`,
         DBarea = `DBarea2016/IDsup2016`)
```

Finally, we may wish to link our geographic (shapefile) data and attribute data by joining on a common variable. In this case, we join based on the Dissemination Block ID

```
my_data_on <- my_shp_on %>%
  left_join(my_attr_on, by = "DBUID")

glimpse(my_data_on)
```

Now that we have linked geographic boundary and attribute data, population size, dwelling counts and other metrics are available for each region. We can generate a quick map of population size, for example, using `geo_plot()`. We first subset the data to include only the Guelph region.

```
my_data_guelph <- my_data_on %>%
  filter(CMANAME == 'Guelph')

geo_plot(data = my_data_guelph,
         geography_col = DBpop,
         plot_type = 'choropleth',
         legend_title = "Population size")
```

Note that data wrangling functions can be nested within **geode** functions, if desired. For example, we could have created the above plot simply by subsetting the Ontario data *within* the `geo_plot()` function: rather than create a separate data object for Guelph, we filter the Ontario data as we plot to include only those rows for the Guelph census metropolitan area (CMA).

```
geo_plot(data = filter(my_data_on, CMANAME == 'Guelph'),
         geography_col = DBpop,
         plot_type = 'choropleth',
         legend_title = "Population size")
```

A wide variety of functions are available for wrangling data in R – an overview can be found in the **tidyverse** cheat sheet:

https://rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf

4.1.5 Data export

Finally, we can export our new linked geodata to a file for future use. We make use of the `geo_export()` function, which allows us to output data as a shapefile, an excel workbook, or a SAS data file. Note that because our geodata contains a geometry column (which specifies the the vertices of our geographic boundaries), we must export as a shapefile.

```
# specify path and name of output file
outfile <- "C:/Users/Michael/Documents/R/working dir/data/guelph_db"

# export as a shapefile
geo_export(my_data_guelph,
           path = outfile,
           filetype = "shp")
```

4.2 Example 2: mapping and proximity analysis

4.2.1 Background

This example illustrates basic usage of the **geode** package, focusing on creation of plots and running a proximity analysis. Here, we make use of the geographic boundary data for BC's Community Health Service Areas (CHSA) and the point locations of BC acute care hospitals. These data files are freely available through DataBC:

https://catalogue.data.gov.bc.ca/dataset/community-health-service-areas-chsa
https://catalogue.data.gov.bc.ca/dataset/hospitals-in-bc

Help files for **geode** functions can be accessed simply by including a '?' before the name of the function of interest. In this example, we focus on the functions `geo_plot()` and `geo_distance()`.

```
?geode::geo_plot
?geode::geo_distance
```

Note that in R, if a package has been installed but not loaded, we must specify the package name and the function name (`package_name::function_name`) in order to use that function or access its help file. Once a package has been loaded, we only need to specify the function name.

4.2.2 Setup

As always, we begin by loading the **geode** package.

```
library(geode)
```

As in Example 1, we also make use of the **tidyverse** package for data manipulation ('wrangling'). The **geode** package has been designed so that it is completely compatible with all **tidyverse** functions.

```
library(tidyverse)
```

We specify the locations and names of data files to be imported. Recall that you must modify the input directory path ('indir') shown below to match the location on your computer where the input files have been saved.

```
indir <- "C:/Users/Michael/Documents/R/working dir/data"
infile_boundaries <- paste(indir, "CHSA_2018.shp", sep = "/")
infile_hosp <- paste(indir, "hlbc_hospitals.csv", sep = "/")
```

4.2.3 Importing data

We begin by importing the geographic boundary data (BC community health service areas) using the `geo_import()` function. We may simplify the shapefile (i.e., reduce the file size by removing vertices) for the purposes of creating a smaller, more efficient, working file. However, for generating final calculations and plots it is advisable to use the unsimplified (`simplify = FALSE`) file.

```
bc_geo <- geo_import(path = infile_boundaries,
                     filetype = 'spatial',
                     simplify = TRUE)
```

Next, we import the hospital location data. This originates from a generic csv file, so we specify that this is 'attribute' data, not 'spatial' data.

```
bc_hosp <- geo_import(path = infile_hosp,
                      filetype = 'attribute')
```

We may wish to do some data wrangling in order to make these input files cleaner and easier to work with. As in Example 1, we use the **tidyverse** function `select()` to keep only those variables needed for analysis and to rename them as appropriate for our proximity analysis. Recall, the `geo_distance()` function requires the attribute data to have an 'id' column, as well as point locations identified by column names 'x' and 'y.'

```
bc_geo <- select(bc_geo, CHSA_Name, LHA_Name, HSDA_Name, HA_Name, geometry)
bc_hosp <- select(bc_hosp, HA_Name = RG_NAME, id = SV_NAME, x = LONGITUDE, y
= LATITUDE)
```

Finally, we will use the **tidyverse** `filter()` function to create new subsets of both datasets that include only those rows corresponding to the Fraser Health region of BC. We will use these subsets later in our analysis.

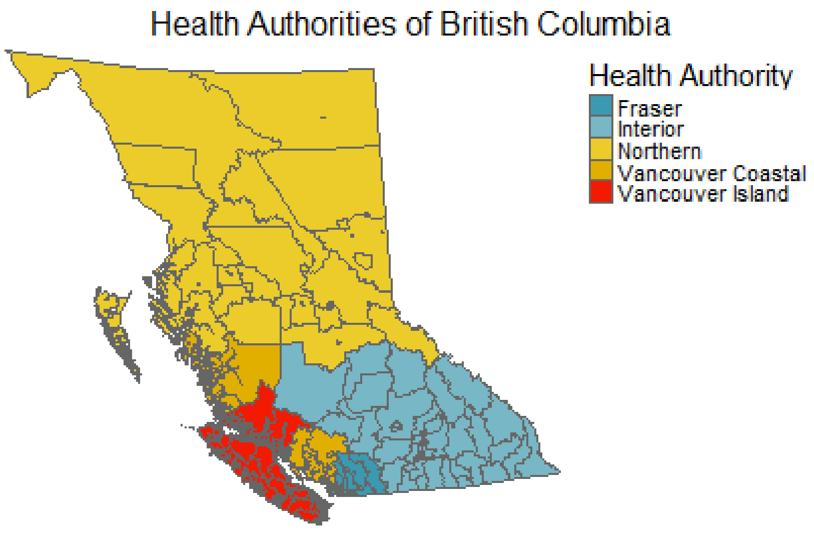
```
fraser_geo <- bc_geo %>% filter(HA_Name == 'Fraser')
fraser_hosp <- bc_hosp %>% filter(HA_Name == 'Fraser Health Authority')
```

4.2.4 Creating maps

4.2.4.1 Choropleth maps

Often we wish to create quick and simple maps from our data, the most common being a static choropleth map. In **geode**, this requires only minimal coding using the `geo_plot()` function. We only need to specify the name of the imported data (`data =`) being mapped, the name of the data variable (`geography_col =`) identifying the geographic regions we wish to map, and the type of plot we wish to generate (in this case a `plot_type = 'choropleth'` map). The remaining arguments shown here (`plot_title =` and `legend_title =`) are optional.

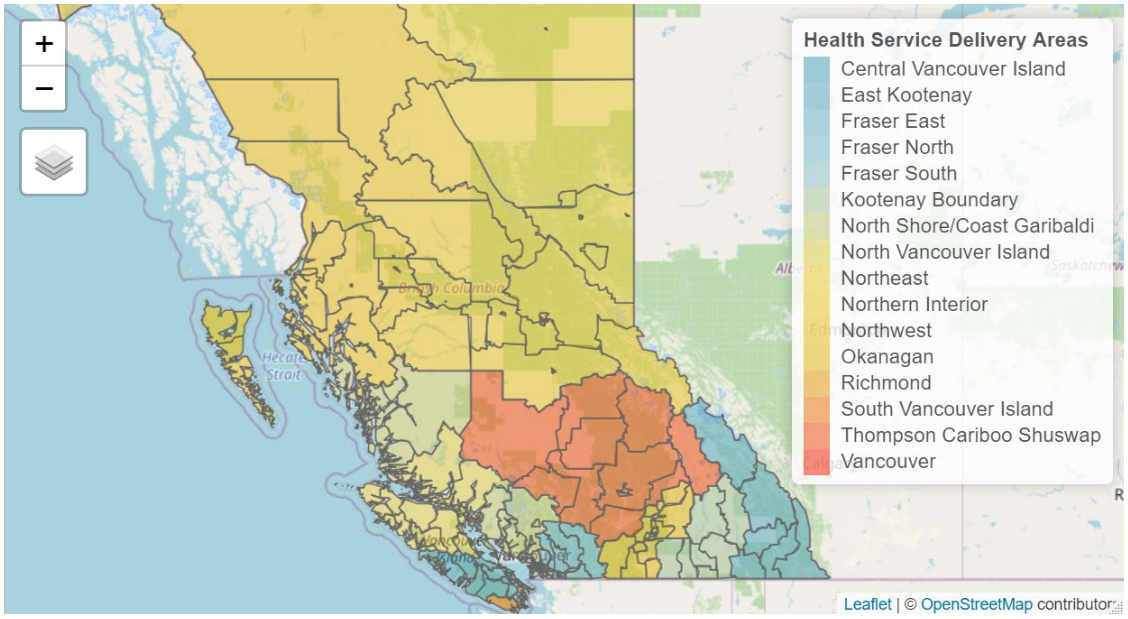
```
geo_plot(data = bc_geo,
         geography_col = HA_Name,
         plot_type = 'choropleth',
         plot_title = 'Health Authorities of British Columbia',
         legend_title = 'Health Authority')
```



Although static images are useful for publications, interactive maps can be a powerful way to explore and communicate the data. Here, we need only specify the option `interactive = TRUE`, which places our data against interactive background layers (similar to those used in Google Maps) that allow us to zoom in and out and obtain information by hovering over an area. Here we use the `hover_id` option to specify that the values shown when hovering over a region are those found in the data column 'CHSA_Name' (if we had excluded this option, the hover-over values would be those found in the `geography_col`, i.e., the

HSDA_Name values). Note that we have also set the transparency of our data layers to be 50% (`transparency = 0.5`) so that we can see some of the background detail through our shading.

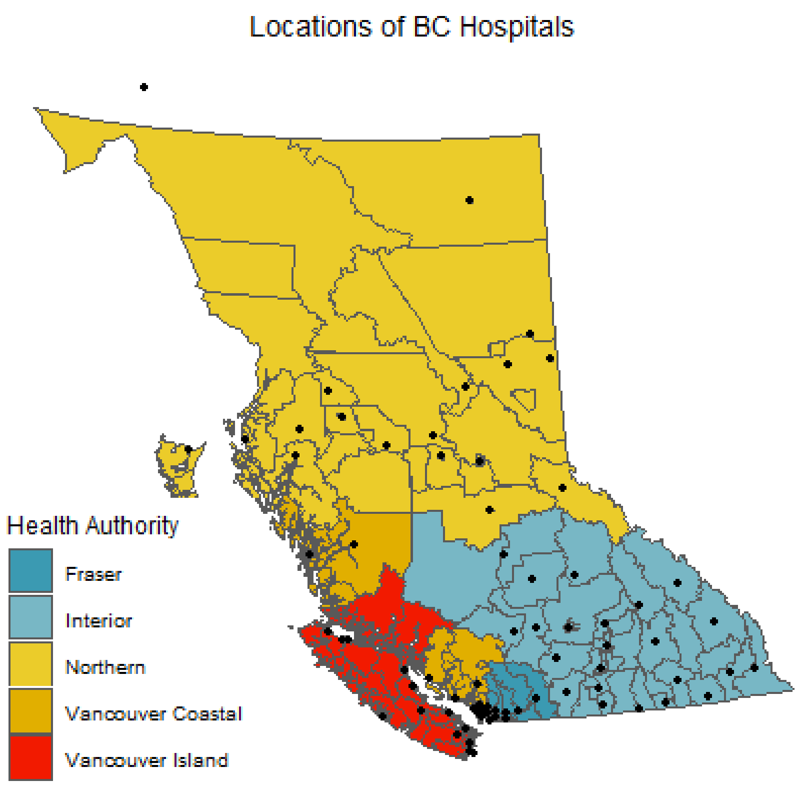
```
geo_plot(data = bc_geo,
         geography_col = HSDA_Name,
         plot_type = "choropleth",
         legend_title = 'Health Service Delivery Areas',
         transparency = 0.5,
         hover_id = CHSA_Name,
         interactive = TRUE)
```



4.2.4.2 Pointmaps

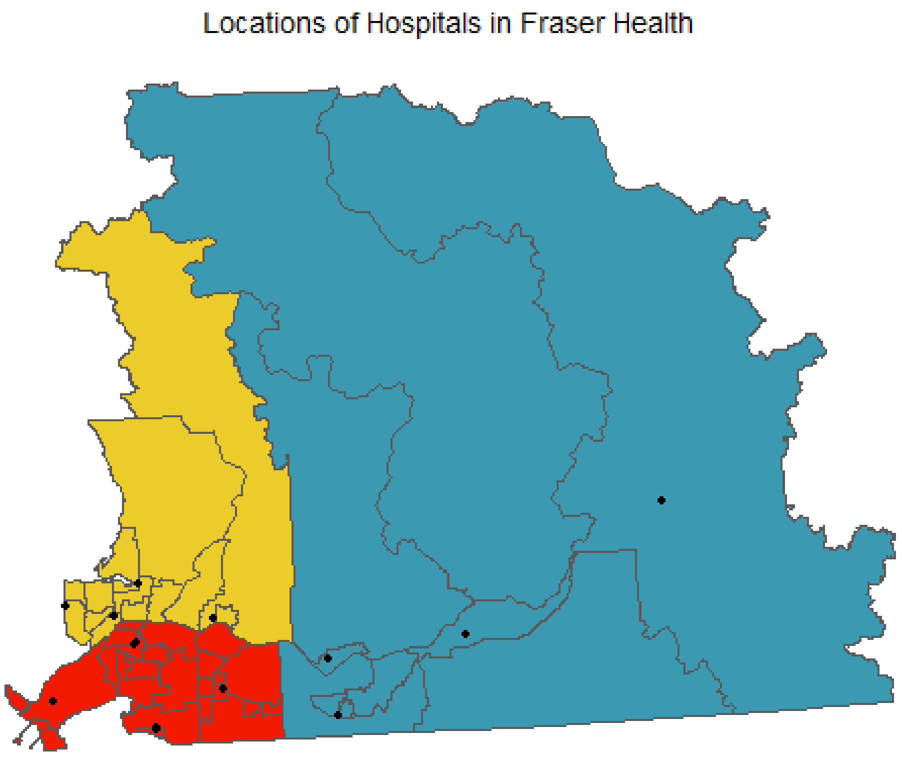
In oder to map point locations of interest (e.g., hospitals), we change the map type to be `plot_type = 'pointmap'` and specify the input data object containing our x-y locations using `attribute_data`.

```
geo_plot(data = bc_geo,
         geography_col = HA_Name,
         attribute_data = bc_hosp,
         plot_type = 'pointmap',
         plot_title = 'Locations of BC Hospitals',
         legend_title = 'Health Authority')
```



We can easily look at particular areas of interest using the subsetted data we created earlier. In this example, we look only at the Health Service Delivery Areas (HSDA) of the Fraser Health Authority. Optionally, we have specified `legend_title = 'none`, which removes the legend entirely.

```
geo_plot(data = fraser_geo,
         geography_col = HSDA_Name,
         attribute_data = fraser_hosp,
         plot_type = 'pointmap',
         plot_title = 'Locations of Hospitals in Fraser Health',
         legend_title = 'none')
```



Note that for illustration purposes we created a separate data object containing only the data for Fraser Health. We did this as a separate step, but could have generated the same plot simply by applying the `filter` function to the geographic and attribute data *within* the `geo_plot()` function itself.

```
geo_plot(data = filter(bc_geo, HA_Name == 'Fraser'),
         geography_col = HSDA_Name,
         plot_type = 'pointmap',
         attribute_data = filter(bc_hosp, HA_Name == 'Fraser Health 
Authority'),
         legend_title = "HSDA",
         plot_title = "Fraser Health Regions")
```

All maps generated using **geode** can be exported as an image file or pdf for later use. As described in Chapter 2, the `png()`, `bmp()`, `jpeg()` or `tiff()` functions can be used to save a map image to your computer. The `pdf()` function similarly allows you to export a map as a pdf.

Using the above map of Fraser Health as an example, we export as both a png image and a pdf:

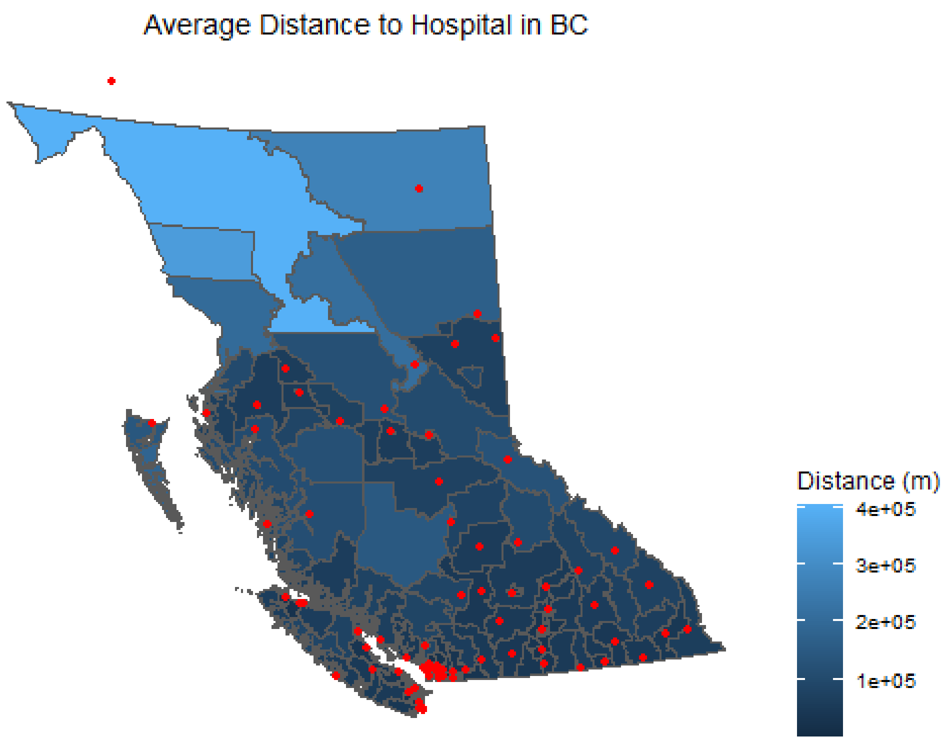
```
my_plot <- geo_plot(data = filter(bc_geo, HA_Name == 'Fraser'),
                    geography_col = HSDA_Name,
                    plot_type = 'pointmap',
                    attribute_data = filter(bc_hosp, HA_Name == 'Fraser 
Health Authority'),
                    legend_title = "HSDA",
                    plot_title = "Fraser Health Regions")

# save map as png image
png("C:/Users/Michael/Documents/R/working dir/output/output_plot.png")
print(my_plot)
dev.off()

# save map as pdf
pdf("C:/Users/Michael/Documents/R/working dir/output/output_plot.pdf")
print(my_plot)
dev.off()
```

This example uses only the default options for the `png()` and `pdf()` functions. Users may specify a variety of arguments in these functions, including image size and resolution. The appropriate syntax for these options can be found by calling the help files for these functions: `?png` or `?pdf`.

4.2.5 Running a proximity analysis

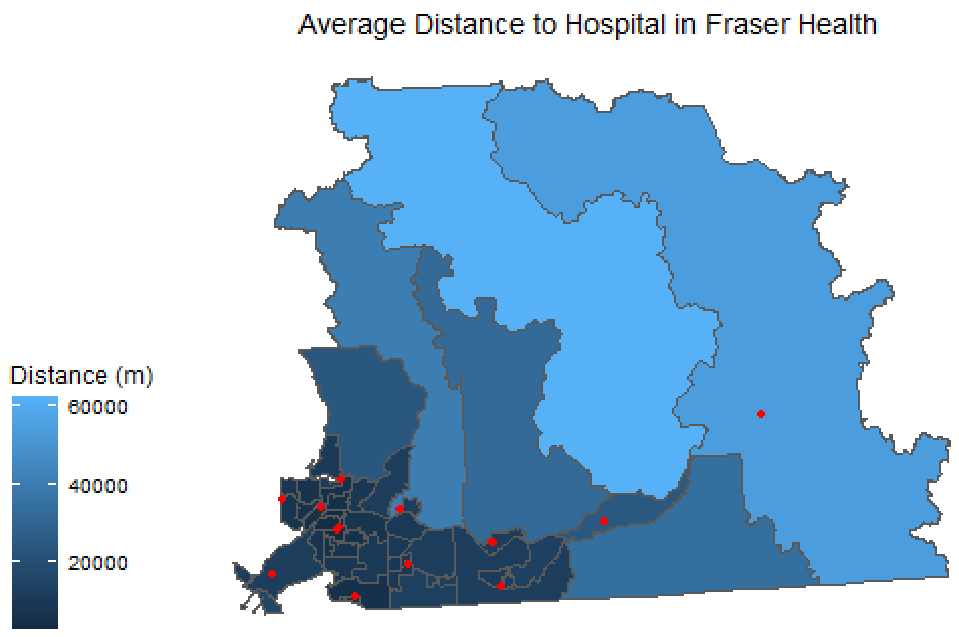
One way in which we might analyze these data is to run a proximity analysis. In **geode**, a proximity analysis is implemented using the `geo_distance()` function. This function calculates the average distance, by region, to point locations of interest. Here we use it to determine average distance to hospital for each community health region in BC. The arguments to the `geo_distance()` function are the same as those encountered in `geo_plot()`, except we now specify the data object containing our point locations using `location_data`. Optionally, we put the legend on the right side of the plot (instead of the default left) using the `legend_position` argument.

```
geo_distance(data = bc_geo,
             geography_col = CHSA_Name,
             location_data = bc_hosp,
             legend_title = "Distance (m)",
             legend_position = "right",
             plot_title = "Average Distance to Hospital in BC")
```



As before, we can look at any particular regions of interest, simply by using the subsetted data created earlier (or by applying the `filter` function directly within the `geo_distance` function). Here, we use the `n_nearest` argument to specify that the average distance calculation should be based on the nearest 2 hospitals rather than the default (n = 3).

```
geo_distance(data = fraser_geo,
             geography_col = CHSA_Name,
             location_data = fraser_hosp,
             n_nearest = 2,
             legend_title = "Distance (m)",
             plot_title = "Average Distance to Hospital in Fraser Health")
```



4.3 Example 3: Cluster detection

4.3.1 Background

This example illustrates the cluster detection functionality of the **geode** package. Here, we make use of the 2016 geographic boundary data for Dissemination Blocks (DB), freely available through Statistics Canada:

https://www12.statcan.gc.ca/census-recensement/2011/geo/bound-limit/bound-limit-2016-eng.cfm

In particular, we will use the Dissemination Block boundaries and population size data that we linked and exported in Example 1. Proceed first through Example 1 in order to create this data file.

Typically, health data are not publicly available at small geographies like Dissemination Blocks. For this example, we simulate mock health data for demonstrating cluster detection.

Recall that help files for **geode** functions can be accessed simply by including a '?' before the name of the function of interest. In this example, we focus on the `geo_detect()` function.

```
?geode::geo_detect
```

4.3.2 Setup

Begin by loading the **geode** and **tidyverse** packages.

```
library(geode)
library(tidyverse)
```

We also specify the location and name of data file to be imported.

Recall that you must modify the input directory path ('indir') shown below to match the location on your computer where the file from Example 1 (guelph_db.shp) was saved.

```
indir <- "C:/Users/Michael/Documents/R/working dir/data"
infile <- "guelph_db.shp"
```

4.3.3 Importing data

We import the example geodata using the geo_import() function. Note how we use the `paste` function here to efficiently combine the input directory path and input filename *within* the importation function.

```
guelph_geo <- geo_import(path = paste(indir, infile, sep = "/"),
                  filetype = 'spatial')
```

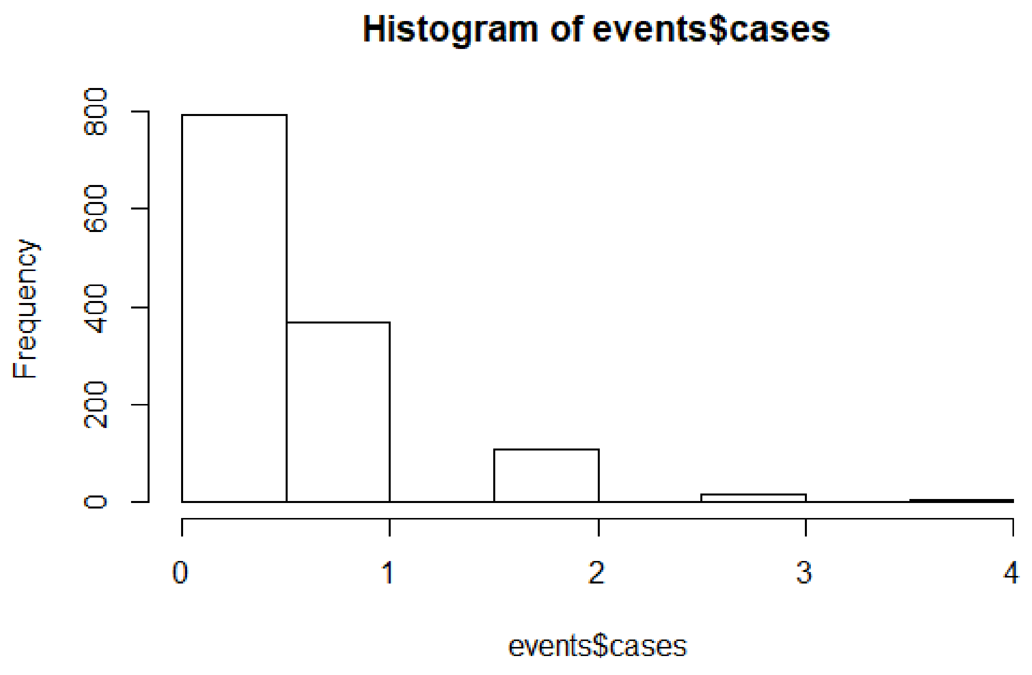
Examine the structure of the geodata. Note that each Dissemination Block (data row) includes information on population size, which we will use in our cluster detection methods.

```
glimpse(guelph_geo)
```

4.3.4 Simulating health event data

For illustrative purposes, we generate a simple distribution of health events (e.g., numbers of cases) using the negative binomial function `rnbinom()`. The size of the distribution is the number of DBs (data rows) in the guelph_geo dataset. We can look at the distribution by generating a simple histogram using the function `hist()`.

```
events <- tibble(cases = rnbinom(n = nrow(guelph_geo), size = 20, mu = 0.5))

hist(events$cases)
```



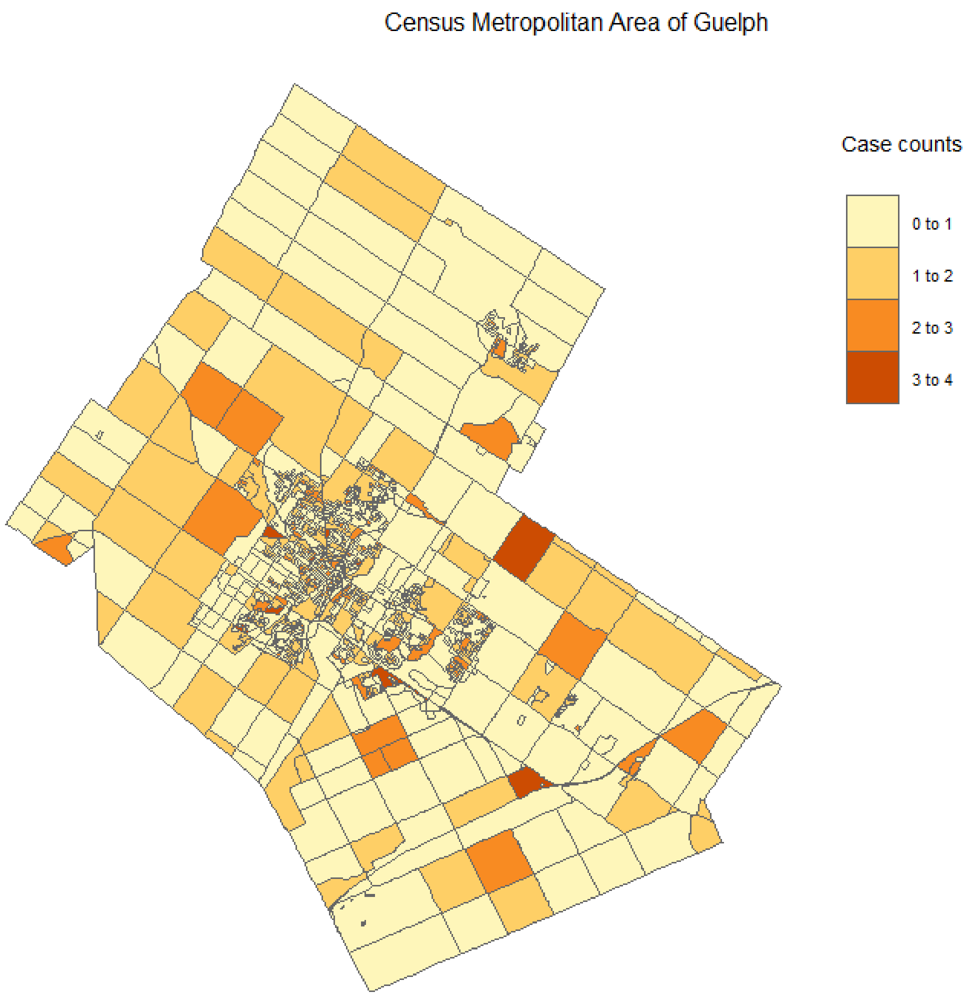
We see that many regions have zero events, but a few have event counts up to four. This distribution is meant to illustrate the occurrence of some rare event.

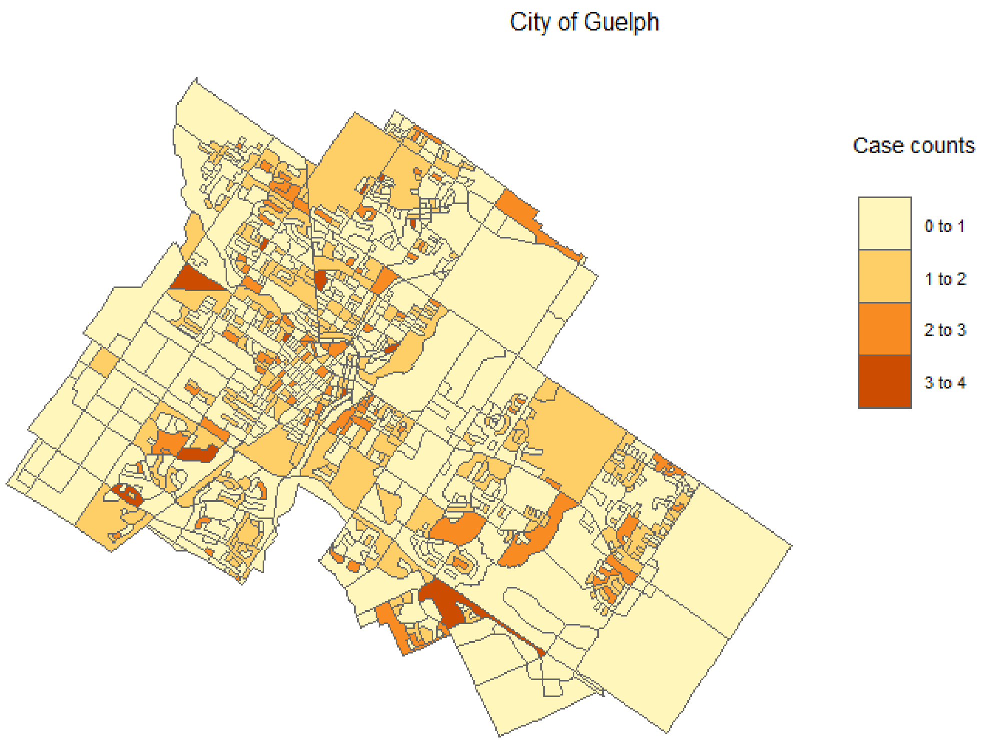
Next, we bind the simulated health event counts ('cases') to our geodata using the function `bind_cols()`. Note that because these are simulated data, there are certain artifacts that we need to correct (likely these would not be found in real data): i) DBs with zero population size must have zero cases, ii) the minimum population size must be greater than zero for all DBs (see Chapter 3 for more information on cluster detection and small population sizes), and iii) case counts cannot be greater than population size in any DB.

```
guelph_data <- guelph_geo %>%
  bind_cols(events) %>%

# regions with zero population must have zero cases
  mutate(cases = ifelse(DBpop == 0, 0, cases)) %>%

# reset min population size to be 5 per region
  mutate(DBpop = ifelse(DBpop == 0, 5, DBpop)) %>%

# ensure cases are not > than population size
  mutate(cases = ifelse(cases>DBpop, cases/2, cases))
```

Finally, let's generate some quick maps of our simulated case counts. We can plot the entire Guelph region, but also generate a 'zoomed-in' map for just the city of Guelph. Note that in the second plot, we simply embed the `filter()` function *within* the plot function to subset the data to only those rows for the city of Guelph.

```r
# map of entire study region
geo_plot(data = guelph_data,
         geography_col = cases,
         plot_type = 'choropleth',
         legend_title = "Case counts",
         plot_title = "Census Metropolitan Area of Guelph")


# map of only the city of Guelph region
geo_plot(data = filter(guelph_data, FEDNAME == 'Guelph'),
         geography_col = cases,
         plot_type = 'choropleth',
         legend_title = "Case counts",
         plot_title = "City of Guelph")
```



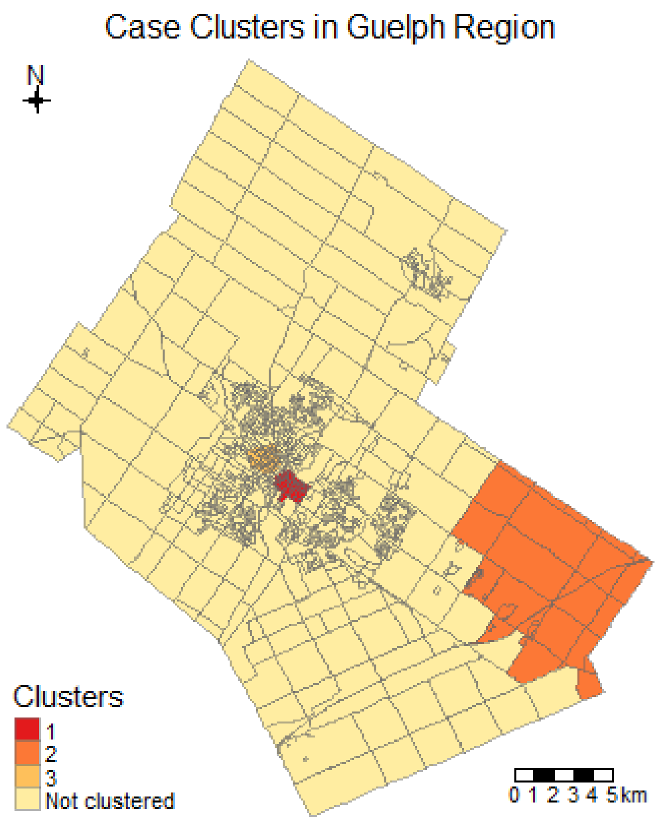


4.3.5 Detecting clusters

In order to run our cluster detection algorithm using the `geo_detect()` function, we must at least specify the geodata and the columns corresponding to the event (case) counts and the population sizes for each region. The default settings are to run the Kulldorff binomial model and generate an interactive plot as the output; hence, these do not need to be specified, but have been included here for clarity.

Note that statistical significance of the clusters is calculated using simulation methods, which may require several minutes of computation time depending on the size of the dataset.

```
geo_detect(dat = guelph_data,
           counts = cases,
           pop = DBpop,
           method = 'kulldorff_binomial',
           plot = TRUE)
```



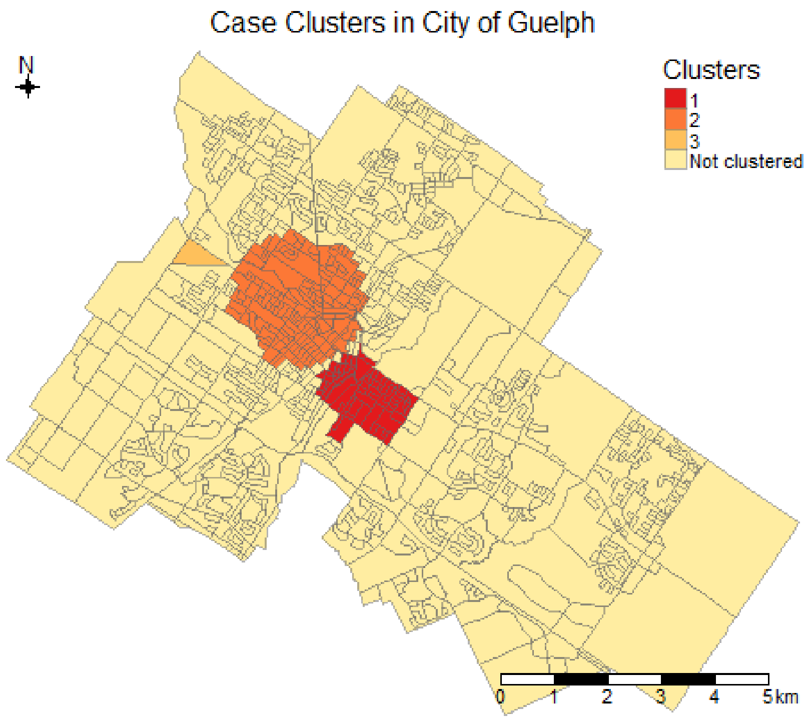
As described above, these are purely artificial data and do not represent any real health events for this region.

The alternative method is the Kulldorff Poisson model, which can be run following the same syntax as above. This model allows us to specify the expected event counts for each region, or to have them calculated automatically (which we do here, by specifying `expected_counts = NULL`). For this example, we do not request a plot output (`plot = FALSE`), so the function instead provides the full listing of the Dissemination Blocks included in the top 3 clusters.

```
geo_detect(dat = guelph_data,
           counts = cases,
           pop = DBpop,
           expected_counts = NULL,
           method = 'kulldorff_poisson',
           plot = FALSE)
```

Notice that the algorithm detects significant clusters in the city of Guelph region, but that these can be hard to visualize in a map of the entire region because the urban Dissemination Blocks are so small. It may therefore be useful to run a separate cluster detection analysis on just the city of Guelph region. As before, we can do this simply by embedding the `filter()` function *within* the detection function to subset to only those rows for the city of Guelph.

```r
geo_detect(dat = filter(guelph_data, FEDNAME == 'Guelph'),
           counts = cases,
           pop = DBpop,
           method = 'kulldorff_binomial',
           plot = TRUE)
```



As with geo_plot() the transparency of the choropleth map can be adjusted to reveal more or less of the background reference layer (using the transparency = argument), and the information shown when hovering over a region can be modified (any column in the input dataset can be specified with the hover_id = argument).

As before, all of these maps can be exported for further use. Using the above cluster map for Guelph as an example, we could export it as either an image or pdf:

```r
my_clusters <- geo_detect(dat = filter(guelph_data, FEDNAME == 'Guelph'),
                          counts = cases,
                          pop = DBpop,
                          method = 'kulldorff_binomial',
                          plot = TRUE)

# save map as png image
png("C:/Users/Michael/Documents/R/working
dir/output/cluster_output_plot.png", height = 4, width = 6, res = 300)
print(my_clusters)
dev.off()

# save map as pdf
```

```
pdf("C:/Users/Michael/Documents/R/working
dir/output/cluster_output_plot.pdf")
print(my_clusters)
dev.off()
```

5 Help files

This chapter provides detailed information on the use of each function in **geode**. Simple examples are also provided to illustrate the basic syntax.

5.1 `geo_import()`

Import spatial or attribute data from an existing file

Description

This function imports spatial or attribute data from an existing source file. Spatial files must be valid shapefiles (.shp), whereas attribute files can be any georeferenced data from a standard csv file. Spatial and attribute data can be imported separately and then linked based on matching geographic units. The source files must be local (e.g., on your computer) or reside at an accessible network location.

Usage

```
geo_import(
  path,
  filetype,
  simplify = FALSE,
  validity_check = FALSE
  )
```

Arguments

path: Filepath specifying location of source file. REQUIRED.

filetype: Name of file type: must be either 'spatial' (.shp file) or 'attribute' (.csv file). REQUIRED.

simplify: TRUE or FALSE (default). If importing a shapefile, optionally create a simplified dataset by removing vertices; useful for creating smaller, more manageable working versions of large shapefiles. OPTIONAL.

validity_check: TRUE or FALSE (default). If importing a shapefile, optionally runs a validity check on each of the geometries and prints the results. Geometries may be invalid due to, for example, slivers or self-intersections; such issues may be present in the imported source file, or arise during simplification. OPTIONAL.

Value

Returns a simple features (if source is a shapefile) or tibble (if source is a csv file) data object

Examples

```
# Import spatial data (shapefile) ------------------------------------------
geo_import(path = "C:/Users/Michael/Documents/my_shapefile.shp",
           filetype = 'spatial')

# Import attribute data (csv file) -----------------------------------------
geo_import(path = "C:/Users/Michael/Documents/my_attribute_data.csv",
           filetype = 'attribute')

# Optionally simplify imported spatial data and run validity check ---------
geo_import(path = "C:/Users/Michael/Documents/my_shapefile.shp",
           filetype = 'spatial',
           simplify = TRUE,
           validity_check = TRUE)
```

5.2 `geo_plot()`

Create maps from geospatial and attribute data

Description

This function creates static or interactive maps (choropleth, pointmap or heatmap) from geospatial and attribute data.

Usage

```
geo_plot(
  data,
  geography_col,
  attribute_data = NA,
  plot_type,
  style = NA,
  transparency = NA,
  levels = 5,
  hover_id,
  plot_title = NA,
  legend_title = NA,
  scale_bar = FALSE,
  compass = FALSE,
  interactive = FALSE
  )
```

Arguments

data: Input data for plotting; must be a simple features (sf) shape object, e.g., imported by geo_import(). REQUIRED.

geography_col: Name of the data column containing the features, e.g., geographic units, to be plotted. REQUIRED.

attribute_data: Name of the dataset containing attributes, e.g., points, to be plotted. Dataset must have a pair of columns for geographic coordinates of the attributes (e.g., lat and lon) titled ‘x’ and ‘y.’ Coordinate reference system (CRS) for attribute data is automatically transformed to match that of input geographic data. If CRS is missing for attribute data, it is assumed to be WGS84. REQUIRED (for pointmaps and heatmaps only).

plot_type: Name of one of the predefined plot types: current options are ‘choropleth,’ ‘pointmap’ and ‘heatmap.’ REQUIRED.

style: Name of a defined output style. User defined output styles can be created following examples shown in the help files for tmap (see ?tmap::tmap_options). For illustrative purposes, the current selection of styles are “zissou,” “royal,” “darjeeling,” “bottlerocket,” “moonrise” and “isleofdogs” (from the wesanderson R pacakge); ’viridis" and “magma” (from the viridis R package); and “terrain” (from base R). If omitted, defaults to tmap “white” style. OPTIONAL

transparency: value between 0 and 1 (default) that defines transparency of map colours, from transparent (0) to opaque (1). Intermediate values allow more or less visibility of underlying (reference) layers in interactive maps. Applies only to choropleth maps. OPTIONAL.

levels: Preferred number of shading categories for choropleth maps when geography_col is a numeric variable. Default is 5. Generates equally sized categories of round numbers, resulting in a number of levels close to, but not necessarily exactly, the specified number of levels. OPTIONAL.

hover_id: Name of the data column containing the label or value to be shown when hovering over a particular geographic unit. For interactive choropleth maps only. Default is the name or value given in specified geography_col. OPTIONAL.

plot_title: Main title of plot. If omitted, no title is shown. OPTIONAL.

legend_title: Title of plot legend. If ‘none,’ legend is removed. If no value is given, feature_col name is used. OPTIONAL.

scale_bar: TRUE or FALSE (default). Indicating whether reference scale bar should be shown (bottom left of plot). OPTIONAL.

compass: TRUE or FALSE (default). Indicating whether reference compass should be shown (top right of plot). OPTIONAL.

interactive: TRUE or FALSE (default). Indicating whether map should be generated as an interactive view or as a static image. OPTIONAL.

Examples

```
# simple static choropleth map with plot title and legend title -------------
geo_plot(data = my_geo_data,
 geography_col = region_data_column,
 plot_type = 'choropleth',
 plot_title = 'My Geographic Regions',
 legend_title = 'Region Name')

# interactive choropleth map with partial transparency of shaded regions ----
geo_plot(data = my_geo_data,
 geography_col = region_data_column,
 plot_type = 'choropleth',
 transparency = 0.5,
 hover_id = region_popsize_column,
 interactive = TRUE)

# static pointmap with plot title but no legend ------------------------------
geo_plot(data = my_geo_data,
 geography_col = region_data_column,
 attribute_data = my_point_location_data,
 plot_type = 'pointmap',
 plot_title = 'My Point Locations',
 legend_title = 'none')
```

5.3 `geo_distance()`

Calculate and maps proximity metrics based on known point locations and geographic units.

Description

The function examines proximity relative to user-specified point locations. The input data are i) a set of geographic units (e.g., census regions, health regions, etc.), and ii) the locations of places or events of interest. The function calculates the centroid of each geographic unit and the pairwise distances to all locations of interest. Proximity for each geographic unit is calculated as the average distance to the n nearest locations.

Usage

```
geo_distance(
  data,
  geography_col,
  location_data,
  n_nearest = 3,
  plot_title = NA,
  legend_title = NA,
  legend_position = "left",
  plot = TRUE
  )
```

Arguments

data: Geographic input data; must be a simple features (sf) shape object, e.g., imported by geo_import(). REQUIRED.

geography_col: Name of the data column containing the features, e.g., geographic units, to be plotted. REQUIRED.

location_data: Name of the dataset containing point locations to be used in distance calculations. Dataset must contain a column for location name or ID, titled 'id,' and a pair of columns for the geographic coordinates of the points (e.g., longitude and latitude), titled 'x' and 'y.' Coordinate reference system (CRS) for these data is automatically transformed to match that of the input geographic data. If CRS is missing for location data, it is assumed to be WGS84. REQUIRED.

n_nearest: number of point locations to use in calculation of proximity (average distance from centroid to n nearest locations of interest). Default is 3. If value is set at 1, proximity for each geographic unit will simply be the distance from the centroid to the nearest location. Must be specified as a positive integer value. OPTIONAL.

plot_title: Main title of plot. If omitted, no title is shown. OPTIONAL.

legend_title: Title of plot legend. If 'none,' legend is removed. If no value is given, geography_col name is used. OPTIONAL.

legend_position: location of legend relative to plot. Can be set as "left" (default) "right," "top," or "bottom." OPTIONAL.

plot: TRUE (default) or FALSE, indicating whether or not a plot should be generated. If FALSE, a dataset is returned containing the calculated proximity values for each geographic unit (with new column 'mean_dist'). OPTIONAL.

Examples

```
# Basic example of proximity analysis using default options and outputting a
#  map ---------
geo_distance(data = my_geo_data,
  geography_col = region_data_column,
  location_data = my_point_location_data,
  legend_title = "Distance (m)",
  legend_position = "right",
  plot_title = "Average Distance to Point Locations")

# Proximity analysis using nearest 5 point locations and generating data
#  output ----------
geo_distance(data = my_geo_data,
  geography_col = region_data_column,
  location_data = my_point_location_data,
  n_nearest = 5,
  plot = FALSE)
```

5.4 `geo_detect()`

Detect regions that form an unusual spatial aggregation of events.

Description

This function identifies clusters in data organized and aggregated into regions. The Kulldorff cluster detection method is used, which creates groupings of regions by consecutively aggregating nearest-neighbour areas until a set proportion of the total population is reached. The number of cases or events in the aggregation, compared to the expected count (as calculated or user-specified) is used to calculate the likelihood based on either a binomial or Poisson model. The output is the group of neighbouring regions that form the *most likely cluster* and, any secondary clusters. Optionally a map of the clusters can be generated.

Usage

```
geo_detect(
  data,
  counts,
  pop,
  expected_counts = NULL,
  method = "kulldorff_binomial",
  plot = TRUE,
  legend_title = "Clusters",
  plot_title = NA,
  transparency = NA,
  hover_id,
  interactive = TRUE
  )
```

Arguments

data: Name of input data for analysis. REQUIRED.

counts: Name of the data column that contains the observed counts that will be tested for clustering. REQUIRED.

pop: Name of data column that contains population sizes to be used for each region. For a cluster analysis, all regions must have a population size > 0. REQUIRED

expected_counts: Name of the data column that contains the expected counts for each region. Required only when method = kulldorff_poisson is used. OPTIONAL.

method: Statistical method to be used for cluster detection. Available methods are kulldorff_binomial (default) and kulldorff_poisson. OPTIONAL.

plot: Should an output plot be created showing clustered regions? Default is TRUE. OPTIONAL.

legend_title: Title of plot legend. If no value is given, 'Cluster' is used as the default title. OPTIONAL.

plot_title: Main title of plot. If omitted, no title is shown. OPTIONAL.

transparency: value between 0 and 1 (default) that defines transparency of map colours, from transparent (0) to opaque (1). Intermediate values allow more or less visibility of underlying (reference) layers in interactive maps. OPTIONAL.

hover_id: Name of the data column containing the label or value to be shown when hovering over a particular geographic unit. For interactive maps only. Default is the value given in specified 'counts' column. OPTIONAL.

interactive: TRUE (default) or FALSE. Indicating whether map should be generated as an interactive view or as a static image. OPTIONAL.

Examples

```
# Cluster analysis and map of event rate based on Kulldorff binomial model --
geo_detect(data = my_geo_data,
  counts = events_data_column,
  pop = pop_data_column,
  method = "kulldorff_binomial",
  plot = TRUE)

# Cluster analysis and map of event rate based on Kulldorff Poisson model ---
geo_detect(data = my_geo_data,
  counts = events_data_column,
  pop = pop_data_column,
  expected_counts = expected_data_column,
  method = "kulldorff_poisson",
  plot = TRUE)
```

5.5 `geo_calculate()`

Calculate and plot spatial statistics from geospatial and attribute data

Description

Given spatial input data, this function calculates and outputs spatial statistics and, optionally, a plot or map. Currently, the function will calculate the global (Moran's I) or local (Getis-Ord Gi*) spatial autocorrelation of a single spatial variable or rate. Rates are calculated automatically if a denominator variable is specified. Optional plots are Moran's I scatterplot for global spatial autocorrelation and a hotspot/coldspot map showing locations of significant local spatial autocorrelation.

Usage

```
geo_calculate(
  data,
  var,
  statistic,
  denom = NA,
  create_plot = FALSE,
  plot_title = NA,
  transparency = NA,
  hover_id,
  interactive = TRUE
  )
```

Arguments

data: Input data for plotting; must be a simple features (sf) shape object, e.g., imported by geo_import(). REQUIRED.

var: Column name of (numeric) variable to be analyzed. For calculation of rates, specify numerator with var and denominator with denom. REQUIRED.

statistic: Name of analysis to run on variable specified by 'var' (or rate specified by var/denom). Current options include 'global_ac' (global autocorrelation using Moran's I) or 'local_ac' (local autocorrelation using Getis-Ord Gi*). REQUIRED.

denom: Column name of (numeric) variable to be used as denominator in the calculation of rates. If included, numerator is assumed to be variable specified by var argument and rates are calculated and analyzed automatically. OPTIONAL.

create_plot: TRUE or FALSE (default). Indicating whether or not plot/map should be generated with statistical output. OPTIONAL.

plot_title: Main title of plot. If omitted, no title is shown. Applies only to static image maps. OPTIONAL.

transparency: value between 0 and 1 (default = 0.6) that defines transparency of map colours, from transparent (0) to opaque (1). Intermediate values allow more or less visibility of underlying (reference) layers in interactive maps. OPTIONAL.

hover_id: Name of the data column containing the label or value to be shown when hovering over a particular geographic unit. For interactive maps only. Default is the name or value given in specified var column (or the rate, if denom is also specified). OPTIONAL.

interactive: TRUE (default) or FALSE. Indicating whether map should be generated as an interactive view or as a static image. OPTIONAL.

Examples

```
# Test for global spatial autocorrelation and generation of Moran's I
#  scatterplot -----
geo_calculate(data = my_geo_data,
   var = rate_column_name,
   statistic = 'global_ac',
   create_plot = TRUE)

# Calculate local spatial autocorrelation and generate hotspot-coldspot map
# (Event rates are calculated automatically as numerator (var) divided by
#   denominator (denom))
geo_calculate(data = my_geo_data,
   var = event_count_column,
   denom = pop_size_column,
   statistic = 'local_ac',
   create_plot = TRUE)
```

5.6 geo_export()

Export geospatial and/or attribute data to a file

Description

This function exports spatial and attribute data to file. The output file type can be a shapefile (.shp), comma separated value file (.csv), Excel workbook (.xlsx), or a SAS data file (.sas7bdat).

Usage

```
geo_export(
  x,
  path,
  filetype,
  na = NA,
  overwrite = FALSE
  )
```

Arguments

x: Name of R data object to output. REQUIRED.

path: Filepath specifying location and name of output file to create. REQUIRED.

filetype: Type of output file to create. Must be one of the following: “shp” (default), “csv” (.csv comma separated value file), “sas” (.sas7bdat data file), “excel” (.xlsx workbook file). REQUIRED.

na: Character string to be used for missing values in the output file. Replaces all empty values ("" and " "), and properly coded missing values (NA), with the specified string. If no string is provided, default is NA. OPTIONAL.

overwrite: If output dataset already exists at the path location, should R overwrite (replace existing file) or not? Default is to not overwrite an existing file. OPTIONAL.

Examples

```
# simple export shp file ----------------------------------------------------
geo_export(x = my_dat,
 path = my_outdir,
 filetype = "shp",
 overwrite = TRUE)

# exporting when file already exists --------------------------------------
geo_export(x = my_dat,
 path = my_outdir,
 filetype = "shp",
 overwrite = FALSE) # will not replace existing file

geo_export(x = my_dat,
 path = my_outdir,
 filetype = "shp",
 overwrite = TRUE) # replaces existing file

# specify new value ('unknown') to replace empty and missing values in output
#  file ---
geo_export(x = my_dat,
 path = my_outdir,
 filetype = "shp",
 na = "unknown",
 overwrite = TRUE)
```

6 References

Here we provide a selected set of references and resources to support geospatial analysis and working with data in R. This is by no means an exhaustive list - users are encouraged to always search for new and useful references.

6.1 Key references

- Bivand, R.S., Pebesma, E.J., Gómez-Rubio, V. and Pebesma, E.J. 2013. Applied spatial data analysis with R, 2nd edition. New York: Springer. Preview available at: https://books.google.ca/books?hl=en&lr=&id=v0eIU9ObJXgC

This text book is an excellent reference for key concepts in geospatial data, analysis and visualization. Note that the examples are not specifically oriented to public health (see instead the text book by Waller & Gotway).

- Lovelace, R., Nowosad, J., Muenchow J. 2019. Geocomputation with R. Updated version of full book available at: https://geocompr.robinlovelace.net/

This is currently the single best reference for working with and mapping spatial data using modern R tools, although the examples are not from public health.

- Waller, L.A. and Gotway, C.A. 2004. Applied spatial statistics for public health data. John Wiley & Sons. Preview available at: https://books.google.ca/books?id=OuQwgShUdGAC

This is a very clear and useful reference on spatial analysis from a public health standpoint. The authors focus on concepts and examples relevant to public health and provide descriptions that are ideal for those coming from a background in health and epidemiology. Note that this is not an R text book and no R examples are provided (see instead the book by Lovelace et al.)

- Grolemund, G. and Wickham, H. R for Data Science. Full book available at: https://r4ds.had.co.nz/

This is an essential book for new users of R. The authors provide clear descriptions and examples of how to work efficiently with data in R using the tidyverse tools. Note that there is no specific content related to geospatial analysis.

6.2 Additional resources

6.2.1 Books

- Albert, D.P., Gesler, W.M. and Levergood, B. eds. 2003. Spatial analysis, GIS and remote sensing: applications in the health sciences. CRC Press. Preview at: https://books.google.ca/books?id=Abj0LW4BlU0C

- Brunsdon, C., and Comber, L. 2018. An Introduction to R for spatial analysis and mapping. SAGE Publications Limited. Preview at: https://books.google.ca/books?id=zbRkDwAAQBAJ
- Cromley, E.K. and McLafferty, S.L. 2011. GIS and public health. Guilford Press.
- Fotheringham, A.S. and Rogerson, P.A., eds. 2008. The SAGE handbook of spatial analysis. Sage. Preview at: https://books.google.ca/books?id=phEgXfbCU_YC
- Krygier, J. and Wood, D. 2016. Making maps: a visual guide to map design for GIS. Guilford Publications. Preview at: https://books.google.ca/books?id=yw3U89CUWGwC
- Lamigueiro, Ó. P. 2014. Displaying time series, spatial, and space-time data with R. Chapman and Hall/CRC. Preview at: https://books.google.ca/books?id=Q5A-AwAAQBAJ
- Tufte, E.R. 2001. The visual display of quantitative information (Vol. 2). Cheshire, CT: Graphics press. Preview at: https://books.google.ca/books?id=GTd5oQEACAAJ
- Wikle, C.K., Zammit-Mangion, A., and Cressie, N. (2019). Spatio-Temporal Statistics with R. Chapman & Hall/CRC, Boca Raton, FL. Preview at: https://books.google.ca/books?id=FD-IDwAAQBAJ
- Wilkinson, L. 2012. The grammar of graphics. In Handbook of Computational Statistics (pp. 375-414). Springer, Berlin, Heidelberg. Preview at: https://books.google.ca/books?id=aSv09LwmuRYC

6.2.2 Papers

- Feng, X., Tan, X., Alenzi, E.O., Rai, P. and Chang, J., 2016. Spatial and temporal variations of screening for breast and colorectal cancer in the United States, 2008 to 2012. Medicine, 95(51).
- Kulldorff, M. and Nagarwalla, N. 1995. Spatial disease clusters: Detection and Inference. Statistics in Medicine, 14: 799–810.
- Lofters, A.K., Gozdyra, P. and Lobb, R., 2013. Using geographic methods to inform cancer screening interventions for South Asians in Ontario, Canada. BMC Public Health, 13(1), p.395.
- Lovelace, R., Cheshire, J., Oldroyd, R. et al. 2017. Introduction to visualising spatial data in R.
- Mobley, L.R., Kuo, T.M., Urato, M., Subramanian, S., Watson, L. and Anselin, L., 2012. Spatial heterogeneity in cancer control planning and cancer screening behavior. Annals of the Association of American Geographers, 102(5): 1113-1124.
- Tennekes, M. tmap: Thematic Maps in R. Journal of Statistical Software, 84(6): 1-39.

- Tobler W. 1970. A computer movie simulating urban growth in the Detroit region. Economic Geography, 46(Supplement): 234–240.

- Zandbergen, P. A. 2014. Ensuring confidentiality of geocoded health data: assessing geographic masking strategies for individual-level data. Advances in Medicine 2014: 567049.

6.2.3 Websites and online courses

- Bell, N. Geographical Information Systems (GIS) and Epidemiology. PopDataBC. https://www.popdata.bc.ca/etu/online_courses/HGEO101

- Supporting material for the book An Introduction to R for Spatial Analysis and Mapping by Brundson and Comber. https://bookdown.org/lexcomber/brunsdoncomber2e/

- Spatial Data Science with R — R Spatial. Robert J. Hijmans. https://www.rspatial.org/

- R packages and resources for spatial analysis. https://cran.r-project.org/web/views/Spatial.html

- DataCamp online courses for learning R. https://www.datacamp.com/search?q=&facets%5Btechnology%5D%5B%5D=R

- DataCamp online courses for working with geospatial data in R. https://www.datacamp.com/tracks/spatial-data-with-r

- References and resources for the essential data science tools in R. https://www.tidyverse.org/

- UBC Stat 545: Data wrangling, exploration, and analysis with R. http://stat545.com/index.html

6.2.4 Tutorials and examples

- Leroux, D. Cloutier, L. 2014. GIS and Cervical Cancer Screening: The Contribution of Spatial Analysis. 2014 Esri User Conference Paper Sessions. http://proceedings.esri.com/library/userconf/proc14/papers/364_412.pdf

- Moraga, P. 2018. Disease risk modelling and visualization using R. https://www.youtube.com/watch?v=yKpRmxM7kdA&t=14s and https://www.youtube.com/watch?v=wmb9sheiF3s

- ZevRoss Spatial Analysis. 2018. Creating beautiful demographic maps in R with the tidycensus and tmap packages. http://zevross.com/blog/2018/10/02/creating-beautiful-demographic-maps-in-r-with-the-tidycensus-and-tmap-packages/

- Moreno, M. and Basille, M. 2018. Drawing beautiful maps programmatically with R, sf and ggplot2. R-spatial.org. https://www.r-spatial.org/r/2018/10/25/ggplot2-sf.html, https://www.r-spatial.org/r/2018/10/25/ggplot2-sf-2.html and https://www.r-spatial.org/r/2018/10/25/ggplot2-sf-3.html