R als GIS — Eine kleine Einführung

MLI

2019-04-04

R-Grundlagen:

Skripte:

Skripte sind Programme die in *Skriptsprachen* geschrieben sind. Es sind Textdateien die mit einem normalen Texteditor bearbeitet werden können. Diese Skripte werden vom *Interpreter* der jeweiligen Sprache ausgeführt. Jede Programmiersprache ermöglicht das einfügen von Kommentaren, die vom Interpreter ignoriert werden. In **R** werden Kommentare von einem #-Zeichen eingeleitet.

Ein sehr einfaches Beispiel in **R**:

```
1 + 1 # Addieren zweier Zahlen und Ausgabe des Ergebnisses
```

```
## [1] 2
```

```
print("Hello world!") # Ausgabe einer Zeichenkette
```

```
## [1] "Hello world!"
```

Das obige Skript besteht nur aus zwei Zeilen, die

- eins und eins zusammenzählen und
- den Text "Hello world!" ausgeben.

Kommentare sind wichtig, um den Code zu erklären. Andere Menschen oder das zukünftige Selbst haben es sonst schwer, das Programm zu verstehen.

Variablen

Variablen sind Speicherorte für Daten. Sie könne beliebige Arten und Mengen von Daten enthalten und sollten daher immer sinnvoll benannt sein. In **R** werden Daten mit dem Zuweisungsoperator " <- " in Variablen geschrieben.

```
summand <- 41
summe <- summand + 1
print(summe)
```

```
## [1] 42
```

Im obigen Skript wird

1) die Zahl *41* in die Variable *summand* geschrieben,
2) die Summe aus *summand* und 1 in die Variable *summe* geschrieben und
3) das Ergebnis der Berechnung ausgegeben.

Beachte: Die Ergebnisse der ersten beiden Zeilen werden nicht an den Benutzer zurück gemeldet sondern in die Variablen geschrieben. Die letzte Zeile hingegen schreibt nicht in eine Variable sondern gibt ihr Ergebnis auf dem Bildschirm aus.

Funktionen

In **R** wird fast alles mit *Funktionen* erledigt. Funktionen tun immer etwas, geben meistens etwas zurück und nehmen oft wenigstens eine Eingabe entgegen. Die Funktion **round()** zum Beispiel rundet einen numerischen Wert:

```
round(x = 1.234, digits = 1)
```

```
## [1] 1.2
```

Die Teile innerhalb der Klammer sind die *Argumente* der Funktion. Im obigen Beispiel werden der Funktion **round()** zwei Argument mitgegeben, die zu rundende Zahl (*x*) und die Anzahl Dezimalstellen (*digits*) auf die gerundet werden soll.

Datentypen:

Es ist wichtig, darauf zu achten, was für eine Art Daten man verarbeitet. Es gibt neben numerischen Werten (*numeric*) und Zeichenketten (*character*) noch viele, viele weitere.

```
# Zeichenketten werden von Anführungszeichen gerahmt
text <- "zweiundvierzig"
noch.ein.text <- "42"

zahl <- 42 # das ist eine Zahl, man kann damit rechnen
zahl + 0
```

```
## [1] 42
```

Daten können in ganz verschiedenen (und beinahe beliebig komplexen) Kombinationen und Strukturen angeordnet sein. Eine sehr nützliche und weithin bekannte Struktur ist die Tabelle (*data.frame*):

```
data.frame.example <- data.frame(a = text, b = noch.ein.text, c = zahl)

data.frame.example
```

```
##               a  b  c
## 1 zweiundvierzig 42 42
```

Geodaten in R

Geodaten werden standardmäßig nicht von R unterstützt, doch Dank einer großen, freundlichen, kompetenten Community gibt es eine riesige Menge an Erweiterungen (sogenannte *Packages*), mit denen sich der Funktionsumfang den eigenen Bedürfnissen anpassen lässt.

Eine dieser Erweiterungen ist das **simple features** Package (*sf*). Es erlaubt das komfortable Laden, Editieren und Speichern von Geodaten (Points, Lines, Polygons, ...) mit Hilfe von bekannten Tools aus dem GIS-Werkzeugkasten (Puffer, Union, Projezieren, ...). All diese Werkzeuge kommen natürlich als R-Funktionen daher und beginnen mit "*st_*".

Ein weiteres Package **mapview** ermöglicht die dynamische Visualisierug von Geodaten.

Ein einfaches Beispiel:

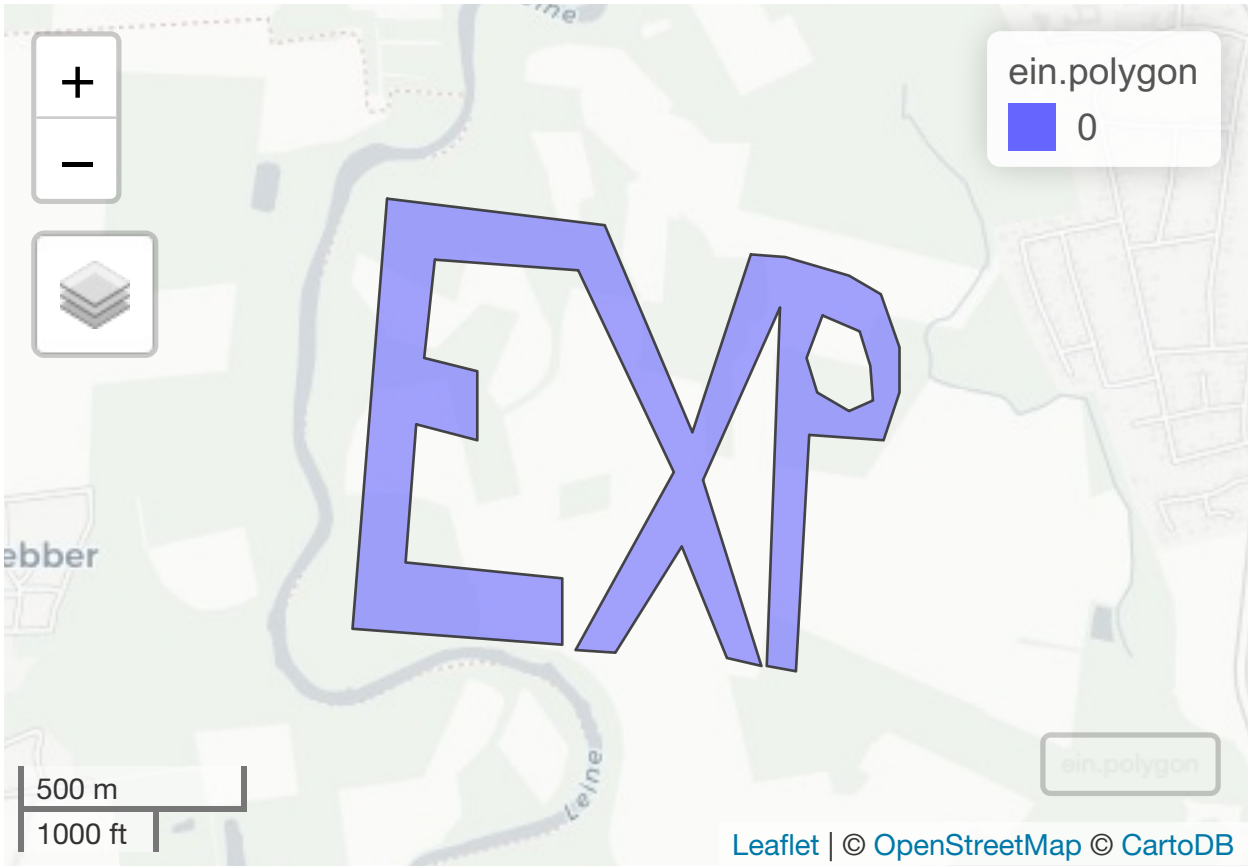
Laden und visualisieren eines Polygons aus einem shp-File.

```r
library(sf) # lade das sf Package
library(mapview) # lade das mapview Package
library(tidyverse) # eine Sammlung großartiger Packete zur Handhabe 'sauberer Daten' (optional aber seh

# Laden eines shapefiles mit der st_read() Funktion und Ablegen in einer Variable:
ein.polygon <- st_read(dsn = "exp.shp") # "dsn" steht für data source name
```

```
## Reading layer `exp' from data source `/Users/markolipka/Nextcloud/UKA/R als GIS Tutorial/RalsGIS/exp
## Simple feature collection with 1 feature and 1 field
## geometry type:  POLYGON
## dimension:      XY
## bbox:           xmin: 9.587599 ymin: 52.66418 xmax: 9.605324 ymax: 52.67349
## epsg (SRID):    4326
## proj4string:    +proj=longlat +datum=WGS84 +no_defs
```

```r
# Visualisieren mit mapview():
mapview(ein.polygon)
```

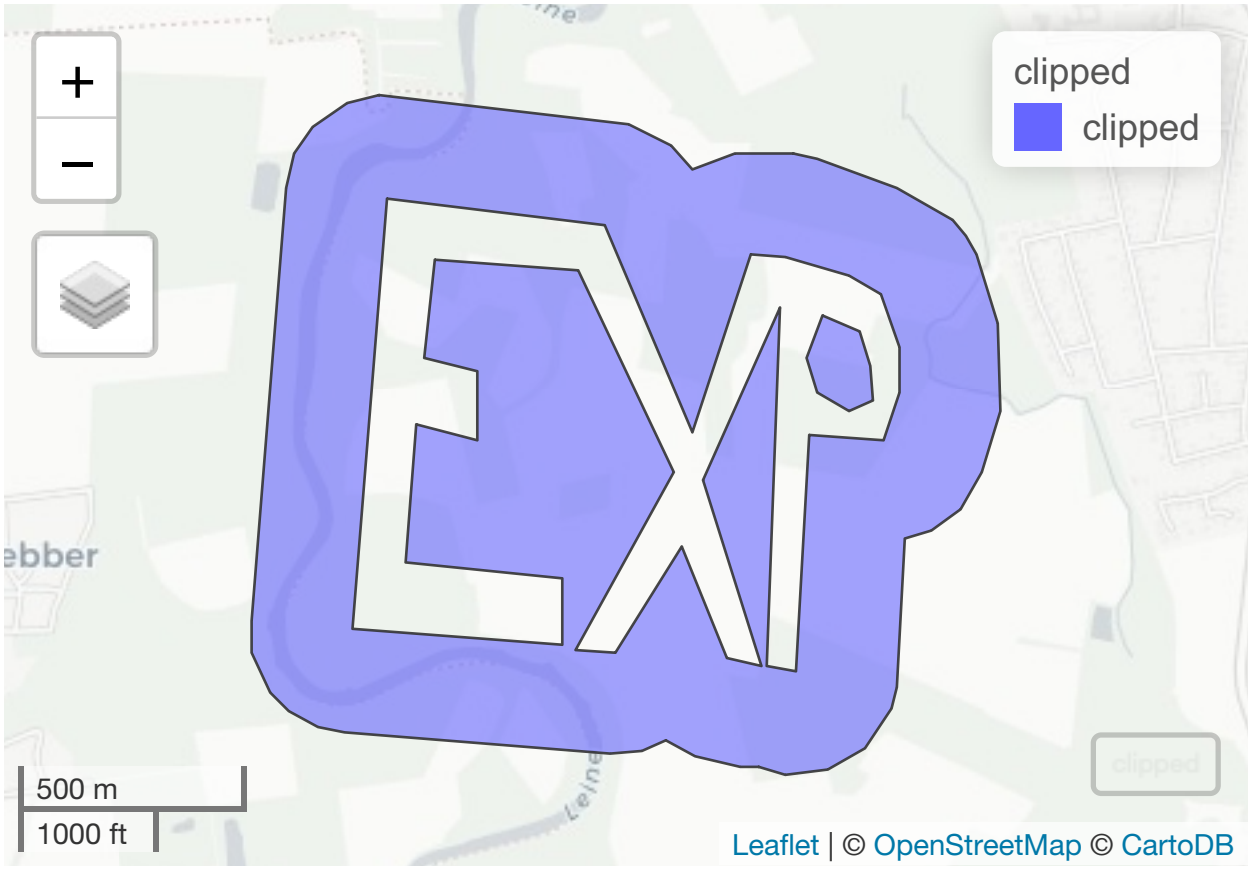


Beispiel 2:

Wir wollen aber Geodaten verarbeiten, nicht nur angucken. z.B. Transformieren in UTM32, Puffer, Clip, Visualisieren

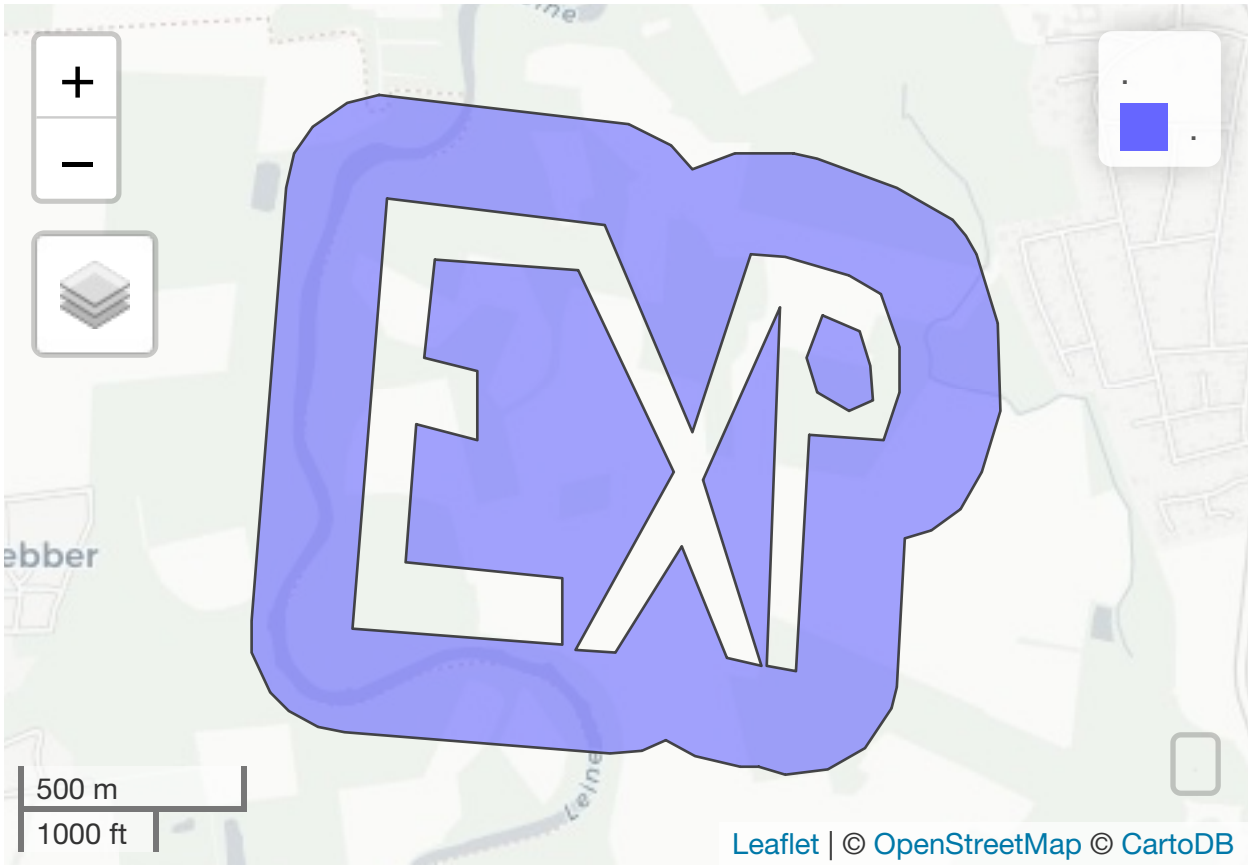
Im Folgenden werden nacheinander verschiedene Geo-Verarbeitungsschritte vorgenomen, was entweder erfordert, dass man das Ergebnis jedes Zwischenschritts in einer Variable speichert und damit im nächsten Schritt weiter macht ...

```r
# Transformation:
ein.polygon.utm32 <- st_transform(x = ein.polygon, crs = 32632)
# Puffer
ein.puffer        <- st_buffer(x = ein.polygon.utm32, dist = 222)
# Clip:
clipped           <- st_difference(x = ein.puffer, y = ein.polygon.utm32)
# Visualisierung
mapview(clipped)
```



... oder aber, man nutzt eine wunderbare Abkürzung, den sogenannten Pipe-Operator %>%. Damit kann man eine Abfolge von Operationen bauen, bei der immer das Ergebnis aus dem Vorhergehenden Schritt die Eingabe für den nächsten Schritt ist:

```r
ein.polygon %>%
  st_transform(crs = 32632) %>% # Transformation
  st_buffer(dist = 222) %>% # Puffer
  st_difference(st_transform(ein.polygon, crs = 32632)) %>% # Clip
  mapview() # Visualisierung
```



noch ein Beispiel, schon recht fortgeschritten:

Wollen wir einen Multi-Buffer durchführen, können wir uns eine eigene Funktion schreiben, die genau das tut, was wir wollen.

```
st_multibuffer <- function(x, list.of.distances) { # Name und Argumente unserer Funktion

  # Hier, im 'body' der Funktion, wird in Form eines Mini-Skripts zusammengefasst, was die Funktion mit

  # Iterieren der Puffer-Funktion über die Werte-Liste mittels der Schleifen-Funktion lapply():
list.of.buffer.rings <- lapply(list.of.distances,
                      function(distance) st_buffer(x, dist = distance))

# Umbauen der Puffer-Ring-Liste zu einem data.frame ...
bound <- do.call("rbind", list.of.buffer.rings) %>%
  # ... und Schreiben der Pufferdistanzen in die Attributtabelle:
  mutate(dist = factor(rownames(.)))

# Was die Funktion schließlich ausgeben soll:
return(bound)
}

# laden eines herzförmigen Polygons aus einer .shp-Datei:
heart.utm32 <- st_read("NI_heart.shp") %>%
  st_transform(32632)
```

```
## Reading layer `NI_heart' from data source `/Users/markolipka/Nextcloud/UKA/R als GIS Tutorial/RalsGIS
## Simple feature collection with 1 feature and 1 field
## geometry type:  POLYGON
## dimension:      XY
## bbox:           xmin: 10.24379 ymin: 53.32745 xmax: 10.25385 ymax: 53.33583
## epsg (SRID):    4326
## proj4string:    +proj=longlat +datum=WGS84 +no_defs
```

```r
# Definieren einer Liste von Werten, mit denen gepuffert werden soll:
pufferdistanzen.um.herz <- list(riesig = 5000,
                groß   = 1000,
                klein  = 100,
                winzig = 10)

# Ausführen der Funktion, plotten der Karte:
st_multibuffer(heart.utm32, pufferdistanzen.um.herz) %>%
  mapview(zcol = "dist")
```