Utrecht University

Applied Data Science Master's degree programme

Spatial Data Analysis and Simulation Modelling course

Instruction manual for Lab 2.2:
Geometric manipulations

Document version: 1

Document modified: 2020.11.12

Dr. Oliver Schmitz, Dr. Simon Scheider

Department of Human Geography and Spatial Planning

Faculty of Geosciences

s.scheider@uu.nl

CONTENTS:

CHAPTER

ONE

INTRODUCTION

Available GIS software packages provide a broad variety of operations to perform analysis of spatial datasets. In case an operation is not built-in one can implement the required functionality. The aim of these exercises is to learn how to manipulate geometric information of geospatial data. Basic geometric manipulations are fundamental for many analysis methods, and they are also needed when manipulating data from unknown or non-standardized formats.

You will use GDAL (https://gdal.org), a software library for creating and manipulating geospatial data, and Python. You will learn to access geometries of existing layers, to query and compute attributes of geometries and to create layers consisting of new geometries.

1.1 Study area

In this exercise you will use a dataset derived from the Dutch BAG (Basisregistratie Adressen en Gebouwen), containing cadastral information about building shapes and addresses in the Netherlands. The layer *Pand* contains information about the surface area of buildings and the layer *Verblijfsobject* coordinate information. The GeoPackage Amsterdam_BAG.gpkg also contains a layer *Wijken* with the districts of Amsterdam, derived from https://maps.amsterdam.nl/gebiedsindeling/?LANG=nl.

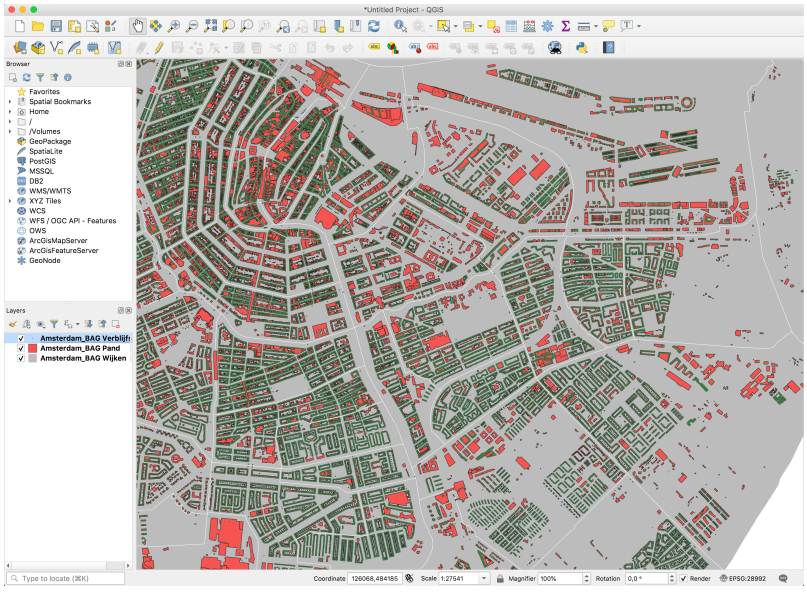

Fig. 1.1: Exemplary display of input data from the BAG dataset. The districts of Amsterdam in grey, ground view of houses in red and front door locations in green.

1.2 Course prerequisites

1.2.1 Software

For the exercises you need Python and some additional software packages. You can use conda (https://docs.conda.io/en/latest/) to create an environment, e.g.

```
conda create --name lab22 -c conda-forge python=3.7 gdal requests
```

in case you do not have QGIS installed you can also install it with conda:

```
conda create --name lab22 -c conda-forge python=3.7 gdal requests qgis
```

if you do not have an editor or IDE installed you can also install Spyder

```
conda create --name lab22 -c conda-forge python=3.7 gdal requests qgis spyder
```

1.2.2 Data

You will use a BAG dataset from the municipality of Amsterdam. The download location of the BAG dataset is provided in BlackBoard.

1.2.3 Recommended reading

Before you start the exercises it is recommended that you make yourself familiar with the vector data model used by GDAL/OGR (https://gdal.org/user/vector_data_model.html). It is not required to view the implementation of the individual classes.

A brief outline of the terminology used:

- Dataset: a file or database containing one or more layer objects.
- Layer: a layer in a dataset consists a set of features
- Feature Class Definition: the schema (set of field definitions) for a group of related features (normally a whole layer).
- Feature: the definition of a whole feature, that is a geometry and a set of attributes
- Spatial Reference: the definition of a projection and datum
- Geometry: the geometry, such as point or polygon, including a spatial reference system

The Python binding of GDAL provides three modules, *gdal* mainly for raster-based operations, *ogr* for vector-based operations and *osr* providing support for spatial reference systems. All modules are submodules of the *osgeo* module. You can find the GDAL Python API at https://gdal.org/python/index.html. You can briefly browse it to get an overview of all classes implemented by GDAL.

It is also recommended to briefly review the WKT/WKB (well-known text/well-known binary) representations of geometries, described e.g. by OGC 17-087r13 (https://www.ogc.org/standards/sfa).

CHAPTER

TWO

EXPLORING CADASTRAL DATASETS

2.1 Querying datasets

You will start exploring the dataset *Amsterdam_BAG.gpkg* by basic listing of the main contents of the dataset. Create a new Python script `explore_bag.py` and add:

```
from osgeo import gdal, ogr
```

You can open an existing GeoPackage with

```
data_source = ogr.GetDriverByName('GPKG').Open(filename, update=0)
```

You can get the number of layers with:

```
data_source.GetLayerCount()
```

Task: Print the number of layers included in the dataset.

To obtain more information about each layer you can retrieve the name and coordinate reference system (CRS) used:

```
layer = data_source.GetLayerByIndex(layer_index)
srs = layer.GetSpatialRef()
```

`GetName()`, `GetAuthorityName(None)` and `GetAuthorityCode(None)` are methods that provide the corresponding information.

Task: Print for each layer the layer name and CRS.

Next you will retrive more detailed information from a specific layer. To identify the number of features in the layer *Verblijfsobject* use `GetFeatureCount()`. You can accesss the layer with

```
buildings = data_source.GetLayerByName('Verblijfsobject')
```

Task: Print the number of features in the layer.

The *Verblijfsobject* layer contains several fields with additional information, the attribute table. To query these fields you first need to get the feature definition of the layer:

```
locations_def = buildings.GetLayerDefn()
```

The number of fields can be retrieved with

```
locations_def.GetFieldCount()
```

and information on each field can be accessed with `GetFieldDefn(index)`, e.g. for the name of a field:

```
locations_def.GetFieldDefn(index).GetName()
```

`GetTypeName()` will return the field type as string.

Task: Print the name and type of each field in the layer.

You can also obtain information of each feature, in particular field values or the geometry. Field values can be retrieved with `feature.GetField(fieldname)`.

Task: Add code to your script that iterates over all features in the layer, retrieves the value of the field *oppervlakte* (surface area) and adds up the area.

Question: What is the total surface area given in the location layer?

You can query individual features by providing a particular index. The feature provides subsequently access to its geometry with:

```
feature = buildings.GetFeature(index)
geometry = feature.GetGeometryRef()
```

As the building addresses are given as point geometries you can access the x coordinate and y coordinate with the `GetX()` and `GetY()` methods, respectively.

Question: What is the coordinate of the feature with the index 439774?

2.2 Obtaining building properties

The layer *Pand* contains the surface area of each building in the city. To calculate the building density in a district we consider a building as belonging to a district when the centroid of a building falls into a district. You will also need to calculate the surface area of each building.

Create a new Python script `building_surface_areas.py`. Open the GeoPackage *Amsterdam_BAG.gpkg* and then the layer *Pand*.

You will create a new GeoPackage to store the results of the following exercises. You also need to create a new layer to store the centroids. For this new layer you will need to assign a CRS, in our case *Amersfoort / RD New*.

```
from osgeo.osr import SpatialReference

rdNew = SpatialReference()
rdNew.ImportFromEPSG(28992)
```

Create the new dataset with the output layer *centroids* using a point geometry type as:

```
centroid_source = ogr.GetDriverByName('GPKG').CreateDataSource('centroids.gpkg')
centroid_layer = centroid_source.CreateLayer('centroids', srs=rdNew, geom_type=ogr.
↪wkbPoint)
```

Note that you can only add once a layer to an existing dataset. You can remove a layer from a dataset with:

```
if data_source.GetLayerByName(layername):
  data_source.DeleteLayer(layername)
```

To add a new field holding floating point values to a layer use:

```
field = ogr.FieldDefn(name, ogr.OFTReal)
layer.CreateField(field)
```

Task: Add a field *area* to your layer *centroids*.

To assign new features to a layer you first need to get its feature definition:

```
centroid_layer_def = centroid_layer.GetLayerDefn()
```

You can then iterate over each feature in a layer using a *for* loop. For each feature retrieve the geometry with

```
house_geometry = feature.GetGeometryRef()
```

A variety of methods can be applied to geometries, such as `Centroid()` or `GetArea()`. Add each centroid as a point geometry to the output layer:

```
point_feature = ogr.Feature(centroid_layer_def)
point = ogr.Geometry(ogr.wkbPoint)
point.AddPoint(centroid.GetX(), centroid.GetY())
point_feature.SetGeometry(point)
```

You can set the value of a field with `SetField` and then add the feature to the layer with `CreateFeature`:

```
point_feature.SetField('area', house_area)
centroid_layer.CreateFeature(point_feature)
```

Task: Calculate the area and the centroid location of each building.

Open both GeoPackages in QGIS and inspect your new layer.

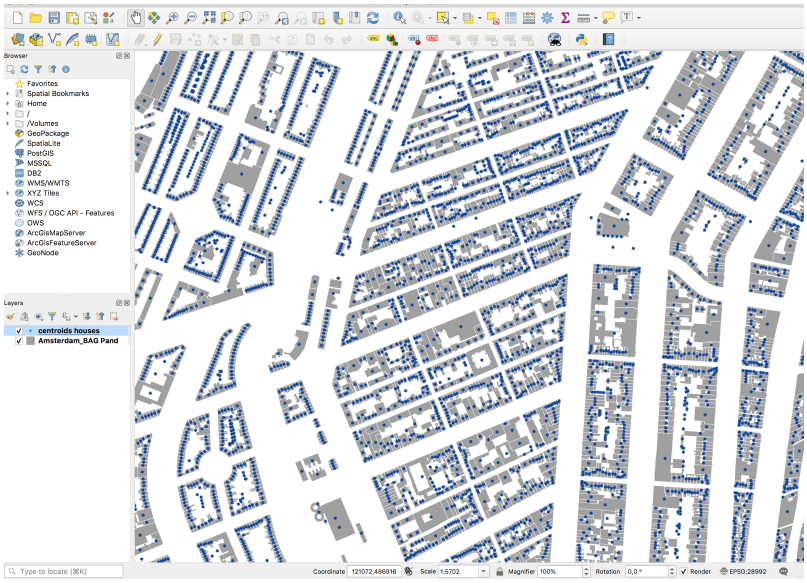

Fig. 2.1: Example of the created dataset. Displayed houses in Amsterdam and their centroids.

2.3 Computing building densities

As final step you will create a density map for houses in the city of Amsterdam using your centroid layer.

Create a new Python script `densities.py`. Open the layer *Wijken* from the Amsterdam dataset and the layer *centroids* from the centroids dataset. Then perform the following steps:

1. Create a new output layer *density* in the dataset *centroids.gpkg*
2. Add three fields to the new layer, *name* of type `ogr.OFTString` and *density* and *fraction* of type `ogr.OFTReal`
3. Iterate over the districts. For each district
   a. Get the name and the size of the area of the current district, and initialise variables to store the number and areas of houses
   b. Iterating over a layer works once. For a repeated iteration over a layer you need to use `ResetReading()` before you attempt to iterate another time:

```
centroid_layer.ResetReading()
```

   c. For each centroid test whether it is in the current district geometry. If so, accumulate the number and area. You can use `Within` to test the geometries:

```
if centroid_geometry.Within(district_geometry):
```

   d. Compute the density and fraction and assign these with the name of the current district to the output layer

---

Question: What is the density of the district with feature id 54 (*Museumkwartier*)?

CHAPTER

THREE

WORKING WITH AMSTERDAM SCHOOL DATA

In this exercise you will you will use available public data to calculate the area far from schools in Amsterdam. You will download school data by an API request, convert the JSON response to point geometries and buffer these to allow for area calculations.

3.1 Retrieving school data

The city of Amsterdam provides information about schools as open data and accessible by an API. You will use their Schoolwijzer OpenData API to retrieve the data, as described at https://schoolwijzer.amsterdam.nl/nl/api/documentatie. The API returns information about Amsterdam's schools in JSON.

Use the Requests Python module (https://requests.readthedocs.io/en/latest/user/quickstart/) to perform your request to the Schoolwijzer API. Note that the Schoolwijzer is provided by HTTPS and Requests verifies by default SSL certificates for HTTPS requests. For simplicity in this exercise you can disable this verification with:

```
requests.get(query, verify=False)
```

Use list as method, the primary schools as school type and no parameters in your request.

Task: Create a Python script `get_school_data.py` that executes an API request to the OpenData API, and writes the result to a file named `schools.json`. Use Python's `json` module to write the file to disk.

3.2 Constructing point geometries

The data contains informations about schools such as names, addresses and their locations given in WGS84. You will use the coordinates from the JSON file to create a new layer `locations` in a new GeoPackage named `schools.gpkg`. Save the following code to a Python script named `convert_json.py`:

```
import json
from osgeo.osr import SpatialReference, CoordinateTransformation
from osgeo import ogr, gdal
```

Assign the CRS *Amersfoort / RD New*.

```
rdNew = SpatialReference()
rdNew.ImportFromEPSG(28992)
```

Create the dataset and add a layer with:

```
ogr_ds = ogr.GetDriverByName('GPKG').CreateDataSource('schools.gpkg')
point_layer = ogr_ds.CreateLayer('locations', srs=rdNew, geom_type=ogr.wkbPoint)
```

With this script you will create a new dataset with an empty layer.

As next step add code to the script to open the `schools.json` file. Assign the contents of the file to the variable `school_data`. Check the data type of of the newly created variable.

You can obtain information of each school by iteration over `school_data` using the `results` key. Extract the latitude and longitude coordinates of each school from `school_data`. Browse the API documentation or the `schools.json` file to identify the necessary dictionary keys for the coordinates.

The school coordinates are given in WGS84, while the desired target CRS is Amersfoort / RD New. It is therefore required to transform each coordinate. You need to create once a transformation object specifying the target and destination reference systems:

```
wgs_to_rd = CoordinateTransformation(wgs84, rdNew)
```

with first argument the source reference system and second argument the target reference system. You can then transform each school coordinate by

```
point = wgs_to_rd.TransformPoint(latitude, longitude)
```

Note that information about schools may be incomplete in the JSON response from the Schoolwijzer API. Exclude schools that have 0, 0 coordinates and name the resulting point coordinates `rd_x` and `rd_y`. Afterwards you can add point geometries to the layer with:

```
feature = ogr.Feature(feature_def)
point = ogr.Geometry(ogr.wkbPoint)
point.AddPoint(rd_x, rd_y)
feature.SetGeometry(point)
point_layer.CreateFeature(feature)
```

Task: Complete your Python script to create the school layer and run it.

Open the `Amsterdam_BAG.gpkg` and `schools.gpkg` datasets in QGIS to visually inspect your new layer.

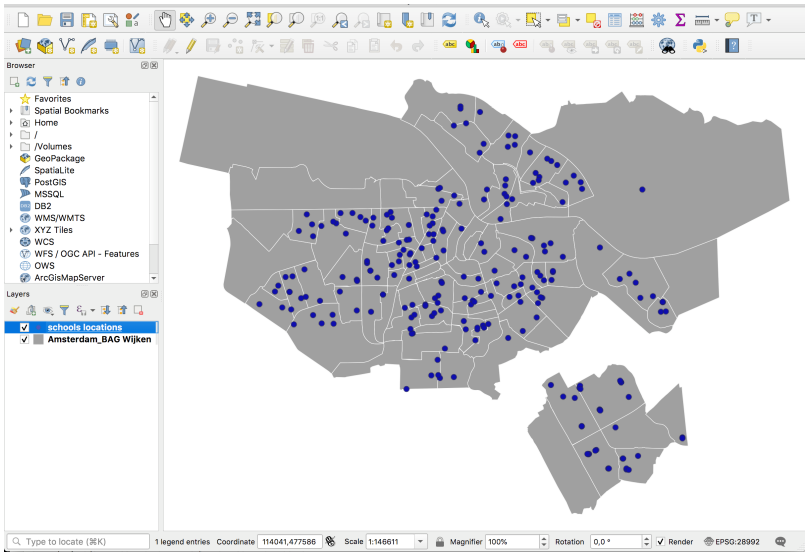

Fig. 3.1: School locations in Amsterdam. Coordinates were received using Schoolwijzer OpenData.

3.3 Adding buffer areas around schools

We consider areas farther away than 250 metres as far from schools. In a first step you will add a 250 meter buffer around each school to determine the areas that are considered as close to schools.

Create a new script named `create_buffer.py`. Open the dataset with

```
data_source = ogr.GetDriverByName('GPKG').Open('schools.gpkg', update=1)
```

Open the layer `locations` with

```
point_layer = data_source.GetLayerByName('locations')
```

and add a new layer `buffer`. The layer will be used to store the new features.

Question: What is the geometry type of the layer `buffer`?

Iterate over all features in the point layer and retrieve each point geometry. You can then use the `Buffer` method to add a buffer around the point:

```
point_geometry = point_feature.GetGeometryRef()
buffer_geometry = point_geometry.Buffer(buffer_distance)
```

Task: Create new features for the buffer geometries and add them to the buffer layer.

3.4 Merging geometries

To compute the area far from schools you will create two new layer, one with the merged school buffers and one with the merged districts. You will iterate over all features in an input layer, and merge individual geometries to one new geometry.

Create a new script named `merge_buffer.py`. Open the schools dataset and the input layer `buffer`. Add a new layer `merge` to the dataset. The layer will be used to store the new features.

Question: What is the geometry type of the layer `merge`?

Start with obtaining the first geometry from the input layer:

```
buffer_feature = buffer_layer.GetNextFeature()
```

Afterwards create a new feature `merge_feature` and add the geometry of the first buffer feature to the new feature. Also add a geometry `merge_geometry` and initialise it with the geometry of your first feature. You will use this geometry to construct the merged buffer area.

Next iterate over the features in your buffer layer performing the following steps:

1. Get the geometry of the current buffer feature
2. Merge the current buffer geometry with the previously merged area:

```
union = merge_geometry.Union(buffer_geometry)
```

3. Create a feature with the merged geometry
4. Update the `merge_geometry` with the new geometry.

With this you extend the buffered area by each iteration.

Task: Create a second script `merge_districts.py` to merge the districts from the *Wijken* input layer of the *Amsterdam_BAG.gpkg* dataset. Merge the district geometries to one new geometry and add the result to the `schools.gpkg` dataset as the new layer `districts`.

3.5 Compute area far from schools

In the last step you will calculate the area that is considered as far away from public schools. Create a new script named `schools_away.py`. Open the `schools.gpkg` dataset and the layer `merge` and `districts`.

Question: Which operation will you use to compute the area far away from schools?

a) Clip

b) Intersection

c) Erase

d) Difference

You can find GDAL's Python API documentation for a layer at https://gdal.org/python/osgeo.ogr.Layer-class.html, including the four operations stated above. These operations can also be used on other types, e.g. geometries. In our case we will apply an operation to the entire layer.

Basically, the suggested OGR operations need a result layer as argument. Add a new layer `away` to your dataset. Afterwards, use the appropriate OGR operation to fill the `away` layer, apply it as:

```
layer1.Operation(layer2, result_layer)
```

As final operation calculate the size of the resulting area. You can calculate the size of a geometry with `GetArea()`.

Question: What is the size of the area considered as far away from public schools?

a) 0.14 km2

b) 168.32 km2

c) 186.64 km2

d) 192.16 km2

CHAPTER

FOUR

ASSIGNMENT

Provide a written report document of your work and submit your document. Perform all the exercises, each of the given task needs to end up in the scripts you create. Insert your scripts on separate pages in the document. Additionally add the answers to all the questions.

Only accepted format of your report is PDF.