Craig Vis – 5873320

INFOMSDASM – Spatial Data Analysis and Simulation Modelling

Course Lecture Summary

Module 1.

Lecture 1.a: Reference Systems

- **Reference system**: A system that provides rules to interpret individual observations with respect to others, and to document the rules so that results can be repeated and compared.
- Geographic information consists of 3 components: **space**, **time**, and **attributes**. Attributes here, are or can range from observable physical properties (e.g., trees, hills, roads) to aesthetic judgements (e.g., color, size):
    - **Temporal reference systems:** A relatively basic system, it has an origin and a unit of measurement such as a second, minute, hour, or duration over a longer span. It could also be periodical, like sequences of events, or cyclical, such as years, calendars.

    - **Spatial reference systems:** A reference system using analytical geometry to formalize relationships between places on a geometric body. It establishes an origin, reference axes, units of measure, and geometric meaning of measurements.
        - **Geodesy**: Science of measuring the shape of the earth and establishing positions on it.
        - The earth is not modelled as a lumpy geoid, but as a **reference ellipsoid**, of which many exist to fit a particular shape of the earth.
        - A **geodetic datum** is a geodetic reference system that is either **horizontal** (I.e., latitude-longitude coordinates on an ellipsoid), or **vertical** (I.e., height).
    - Geometric transformations can take place through projections (I.e., converting latitude-longitude coordinates into planar coordinates) so that 3-dimensional objects can be changed to 2-dimensional representations. When doing this, there is always some l**oss of data**. Loss of data is dependent on:
        - The developable surface/ class (I.e., is the ellipsoid projected as plane, cylinder, or cone?
        - Point of secancy. (I.e., tangent or secant)
        - Aspect (I.e., How is the world viewed: normal, transverse, or oblique)
        - Distortion property (I.e., equal area, equidistant, conformal, or transverse)
            - Conformal: Preserves shape of geographic features (I.e., angles), sacrificing linear and areal scales.
            - Transverse: A projection oriented at right angles to the equator, loses the shape of the geographic features (I.e., angles).

    - **Attribute reference systems:** Some attributes have guidelines or rules for attribute measurement. There are several levels of measurement scales that allow information to be translatable to other scales without loss of information. These are **nominal, ordinal, intervals,** and **ratio** (reference to lecture 2.a for a more elaborate explanation on these terms). In addition to these levels of measurement, there also are **absolute scales** (I.e., a predetermined scale lying between zero to one), **cyclical measures** (I.e., A measure that returns to its origin, such as angles in a 360 cycles), **counts** (I.e., Counts of values that

can only be discrete), and **graded membership** (I.e., It was assumed that members of a group belong exclusively to one scale, while this is not necessarily the case).

Lecture 1.b: Geodata Retrieval

I have not summarized all the sorts of data types because I think it was very arduous and not particularly interesting for the material that we are discussing. Please not, therefore, that this summary might be considered slightly incomplete from what was discussed in relation to the lecture.

Most geodata either consist of vectors, which are data encoded as geometric points, lines and areas, or raster, which are encoded values to cells.

Vector file formats include SHP, WKT, GML & GeoJSON

Raster file formats include: GEoTIFF, netCDF, IMG, ESRI.

Lecture 1.c: Geodata Quality

To evaluate the quality of geodata, we must look at four quality components of the data, these are accuracy, resolution, completeness, consistency and precision.

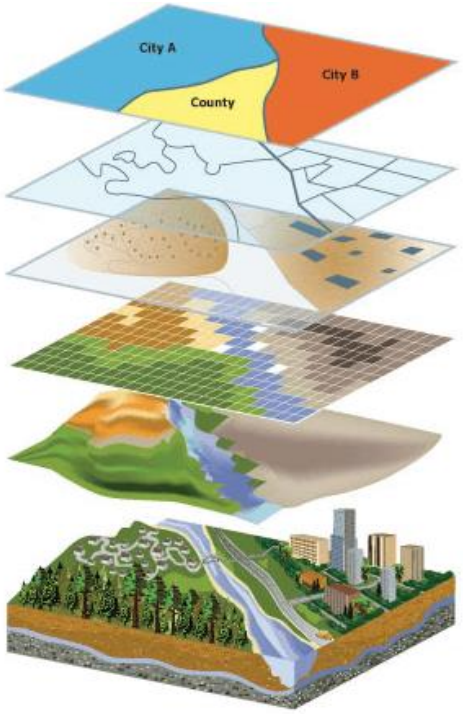
- **Accuracy** regards the discrepancies between the database and a variety of reference sources. Does the database depict the source document from which it was derived? For instance, does a point of the data refer to what it was intended to measure?
- **Precision** regards to the standard error of measurement of the data or the resolution of digital number representations. The more precise the data, the more representative.
- **Resolution** refers to the level of detail within the data. The higher the resolution, the higher the file's size is. The higher the resolution, the smaller the cell size, of for instance rasters. One needs to consider the right resolution for the required analysis
- **Completeness** regards whether the data can be considered complete enough so that it can realistically represent the sample of the data that is measured and/ or presented.
- **Consistency** regards whether the data repeatedly agrees with its values and therefore relates to the integrity of the database.

Module 2.

Lecture 2.a: Geodata Models and Core Concepts

<u>Principles of spatial data transformation:</u>

- Layer principle:
    - Geographical information systems make use of layers of data to explore and study geographical and spatial phenomena.
    - Layers can be **'overlaid'** to achieve several goals:
        - To spatially analyze landscapes
        - To derive new layers from already existing layers
        - To aggregate and summarize data of layers into new layers (Lecture 3.a)
    - Layers can either be **vector overlays** or **raster overlays**.
        - Vectors are values that are either points (individual values), lines (multiple values), and polygons (multidimensional values) that reflect real word features.
            - For example, you can place a height-layer on top of a soil-layer and on top of a vegetation-layer.
        - Raster function as individual cells equally spread out across a matrix, where each cell contains a value representing information (both quantitative (e.g., temperature as qualitative (e.g., land-use)). These cells are more than pixels, as they have cell coordinates and values. In addition, they can relate to both qualitative as quantitative characteristics. In Raster GIS, a **zone** refers to the set of cells sharing a certain value. A **region** is a zone with connected cells. Sometimes, cells might contain missing values. The cell size of a raster influences the **resolution,** which should be decided based on the needs of the analysis. A higher resolution requires more computation power.
            - For example, you can measure the suitability of a land-cell for development, by summing the cells for specific values appointed by a height-layer, soil-layer, and vegetation-layer.



^ An example of how layers of various geodata types are plotted to analyse an environment.

The question is: which GIS methods do you use to prepare and analyze a particular data source? Example questions are:

- When studying the amount of elderly in Amsterdam, do you make use of a **vector overlay method**, or **areal interpolation**?
- When studying CO2 pollution in Germany, do you make use of **aggregation** (I.e., summation) or **interpolation**?

Each method is used dependent on the data type that you are working with.

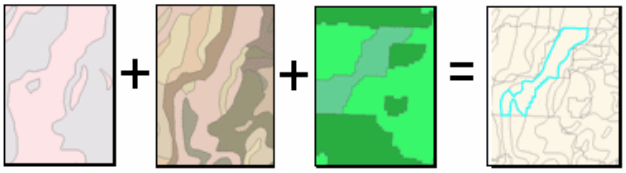
Core concept datatypes (CCD) ontology:

There are different models that work with these geometries, such as the *generalized geometric layer model*, the *vector geometry model*, and the **vector data model.** The latter is the model that we use the most, considering it is the most advanced and extensive. There exists an ontology that combines three dimensions to represent and study spatial data. These three dimensions are **layer types**, **core concepts,** and **levels of measurement**. The following section will study these three dimensions more extensively.

1. Layer types:
    - **Region:** A dataset where the geometric primitives are regions in the forms of polygons or cells (e.g., parks in Amsterdam, or provinces in the Netherlands).
        - **Tessellation**: A subtype of region. A tessellation is a collection of plane figures that fills the plane with no overlaps and no gaps (e.g., Provinces in the Netherlands)
            - **Raster** (regular tessellation): A subtype of tessellation where the regions are cells in a matrix (e.g., height maps).

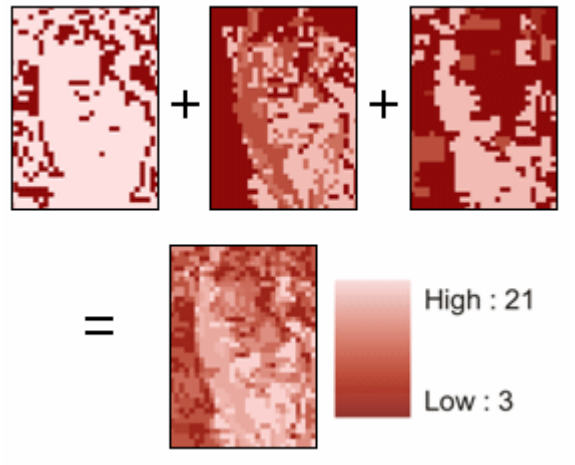
    - **Vector** (irregular tessellation): A subtype of tessellation that is not a raster (so the example of a map of Dutch provinces).
- **Lines**: Lines refer to coordinates that are extend on a map, that relate to networks (e.g., roads)
- **Point**: Points refer to individual coordinates on a map, that relate to objects that are placed on a map (e.g., trees)

In this *vector data model,* a data model consists of the before-mentioned **layers.** Each layer consists of particular **features,** which are **things** you can see, such as trees, houses, roads, and rivers. Each of these features contain their **geometry** and **attributes,** that contain the qualitative or quantitative information to describe the features.



FID	Shape*	FID_soils	CODE	CLASS	FID_sl	SLOPE	FID_veg	DET_TYPE
3039	Polygon	508	38F	6	0	60	117	A
3040	Polygon	508	38F	6	0	60	119	SS
3041	Polygon	508	38F	6	0	60	157	U
3042	Polygon	508	38F	6	0	60	158	A
3043	Polygon	508	38F	6	0	60	160	FC

^ An example of a vector overlay



^ An example of a raster overlay

2. Core concepts:

Core concepts of spatial information relate to the way in which we can represent specific types of data. They are described as 'cognitive lenses through which the environment can be studied'. These types of data can in some instances be **manipulated** to create other types of geodata. With regard to regions, *rasters* and *vectors* can often interchangeably represent the geodata as described below.

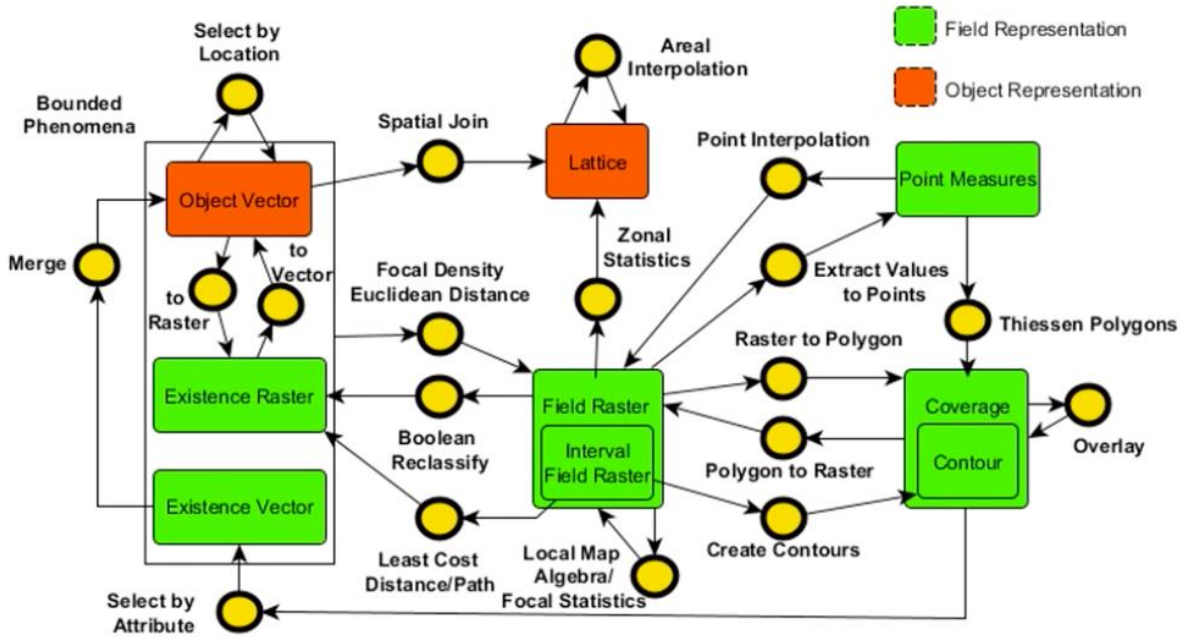
- **Field**: A field is a *continuous surface* of values in space.

    - **Contour**: A contour is a subtype of Field. Each contour encloses field values inside a given interval (e.g., a terrain contour map)
    - **Patch**: A patch is a subtype of Field. Each patch contains homogenous field values (so outside of a given interval) (e.g., dog-restricted areas in a park)
    - **Coverage:** Coverage is a subtype of Field. A coverage is similar to a patch, except that it is a tessellation, indicating that there are no overlapping nor gaps within the data. It is self-similar (I.e., values are consistent across other areas) (e.g., land use types)
    - **Field raster:** Field raster is a subtype of Field. Each cell in the raster represents the intensity of a value (e.g., Elevation map).
    - **Point measure:** Point measure is a subtype of Field. Point-like measurements are distributed across an area (e.g., temperature sensor measurements in a city).
    - **Line measure:** Line measure is a subtype of Field. Line measures represent a field in terms of lines of homogenous values (e.g., a coastline).

- **Object**: Objects are instances of *discrete values* that are named and are spatially bounded. Compared to Field, Object is a whole and indivisible entity of its own.
    - **Lattice**: A Lattice is a subtype of Object that represents not one point in space, but rather a tessellation. It is not self-similar (I.e., polygons) (e.g., values)
    - **Amount:** *NOT* an object on itself, and therefore not a subtype of Object. Instead, it rather represents several objects or matter.

- **Network**: A Network is a quantified relation between pairs of spatial objects. It relates to how object pairs measure against each other (e.g., traffic flow between two road intersections during rush hour).

- **Event:** An Event is a thing which happens in time and space. Events therefore always have a quantitative quality that is denoted as start and end time, as well as duration. The Event must be represented as a point in space, indicating where the event occurred. (e.g., earthquakes)
    - **Track:** A Track is a subtype of Event that are a collection of events demonstrating a trajectory of a moving object. Lines can be used to connect the tracking points. (e.g., a bird's trajectory).

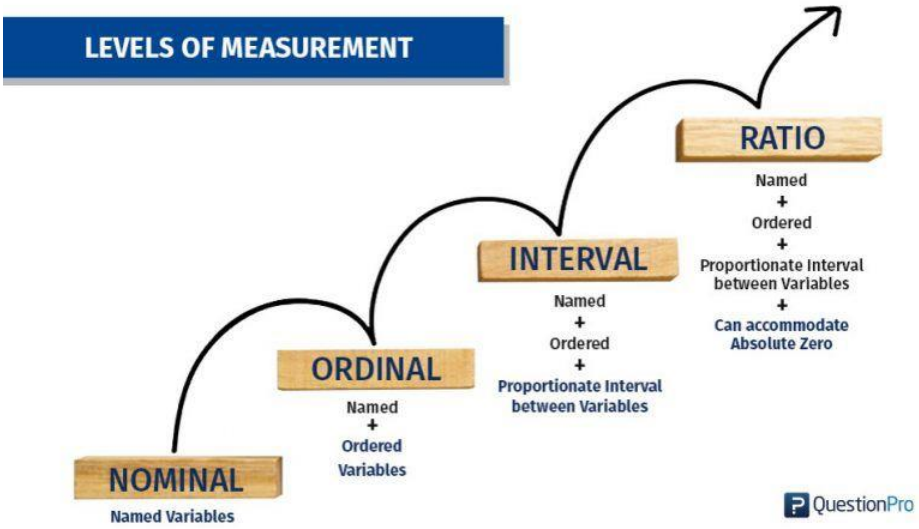


^ An overview showing the various manipulations that can be performed on geodata, demonstrating what input layer is needed, what manipulation is needed and what the consequent resulting output layer will become.

3. <u>Levels of measurement:</u>

Data can be of different kinds. The kind of data play a substantial role in the way in which they can or cannot be manipulated. Data generally is understood as:

- **Nominal**: Nominal data is arbitrary such as a name, or phone number, upon which no mathematical computations can be performed. They are unique identifier for values and are often used as primary keys.
- **Ordinal**: Ordinal data not only shows the identifier of a piece of data, but also shows how that data is ordered in relation to surrounding data.
- **Interval**: Interval data shows both the identifier and order between data, but also shows what the intervals, or gap are between these ordered data. Different interval levels can be interchangeable used while the data is preserved (e.g., temperature in K or C)
- **Ratio**: Whereas interval data shows the intervals between data points, it does not have a reference that indicates what the interval means. Ratio data have an absolute zero point, where the difference between numbers become significant.
- **Count**: Count data is a type of ratio data, where values are integer numbers representing a count of something that is whole.



^ A schematic overview showing the level of measurements. Each level is an advacement to the previous one.

Lecture 2.b: Quality of Maps

The quality of maps is dependent on multiple factors.

When visualizing data, not every **visual variable** (I.e., size, shape, color hue, color value) can be used on every data type. The level of measurement of the data influences how the data can be visualized. For instance, nominal values (I.e., **Discrete)** should not be visualized by means of **continuous** visual variables such as size or color value (that are intended for interval or ratio levels of data. Instead, for nominal values, use shape or arbitrary colors. Using continuous visual variables would suggest that there are either ordinal, interval, or rational differences between such nominal values, while this is not the case. On the other hand, continuous values should sensibly be visualized by attempting to project the continuity of the variable through the visuals. For instance, when using color on continuous values, you cannot simply use arbitrary colors to refer to a particular value, but instead opt for a spectrum of colors that allows for the transition to become apparent.

[The part within the presentation on 'measurement scales' has been added to the summary of the previous lecture, so that the ontology could be completed.]

^ A diagram showing which visual variables can be used on which data type.

Examples of maps are:

- **Choropleth map**. A thematic map where geographic areas are shaded or patterned in relation to a particular attribute value. These maps are **tessellated**. These values can be rational, from 0 to +1, but can also take on two colours, where the values range between –1 and +1. Do not use non-normalized
- **Proportional symbol map.** A map that uses a symbol at a specific or aggregate point, where the value is scaled by the size of a shape. Larger values are therefore larger symbols.
- **Dot density map.** A map that uses dots to indicate and represent data. The higher the density of the dots, the higher a value.
- **Contour map.** A map that uses differences in color hue (I.e., continuous scale) to demonstrate differences in value.

In his book on Thematic Cartography and Geographic Visualization, Slocum made a list of steps to go through when working on maps:

1. Is the map *general*, or *thematic*?
2. What is the spatial *dimensionality* of the map? (I.e., point, line, polygon, volume)
3. What is the *level of measurement*? (I.e., nominal, ordinal, interval, ratio)
4. Do the data require *normalization*? (I.e., is data absolute, or relative)
5. What is the *number of attributes* that need to be mapped?
6. What is the role of *time*?

Lecture 2.c: Introduction to OpenStreetMap

OpenStreetMap (OSM) is a collaborative project to create a free editable map of the world. The geodata underlying the map is considered the primary output of the project. People can attribute to the project voluntarily. It was partially started by dissatisfied cartography companies who had to pay large amounts of money for their products.

OSM data contains four elements:

- **Nodes**: Corresponds to a specific geographical point in space, containing the longitude and the latitude coordinates.
- **Ways**: An ordered list of between 2 to 2,000 nodes that define a polyline. These are used to represent linear features such as rivers and roads, but also buildings or forests.
- **Relations**: A multi-purpose data structure that documents a relationship between two or more data elements. They are used to explain how other elements work together.
- **Tags**: Descriptions about the meaning of the above-named elements.

OSM can easily be used in QGIS, by installing the plug-in and simply searching for nodes and tags.

Lecture 2.d: Spatial Databases

A **spatial database** is a database that is optimized for storing and querying data that represents objects defined in a geometric space. It is similar to a relational database but has some extensions that are used to store and manage spatial data like vector- and raster data.

Spatial data can also be stored in a **shapefile**, but because of the small maximum data storage of 2GB, spatial databases are usually preferred.

As mentioned above, multiple types of spatial data can be stored, like **vector data** (including points, lines, and polygons), and **raster data** (with differing resolutions based on the required analysis and outcomes). In addition, spatial databases can be used to perform various **computations**, like spatial join, spatial aggregation, and spatial computation. In addition, you can perform **spatial indexing** to facilitate the efficiency of queries.

One example of a spatial database, also the most popular one, is **PostGIS**. PostGIS is an extension of **PostgreSQL**, which is a free and open-source relational database management system. PostGIS adds spatial capabilities to the PostgreSQL relational database, which allows it to store, query and manipulate spatial data.

PostGIS defines various spatial datatypes:

- **Geometry**: Represents a feature in planar (Euclidean, I.e., x-y) coordinate systems. The geometric types that are recognized by PostGIS are points, multipoints, lines, polygons, and more.
- **Geography**: Represent a feature in geodetic coordinate systems. Geodetic coordinates are spherical coordinates expressed in angular units (degrees).
- **Raster**: Represents raster data (imported from TIFFs, PNGs).

Like QGIS, a **Spatial Reference Identifier (SRID)** is needed to associate the coordinates with a specific coordinate system. In the Netherlands, we use SRID:28992 --- Amersfoort / RD New. The World Geodetic System is SRID:4326 --- WGS 84.

PostGIS recognizes several spatial functions, that can manipulate geographical data:

- Creation of points, lines and polygons: ST_MakePoint, ST_MakeLine, ST_MakePolygon
- Access to geometries: ST_StartPoint, ST_EndPoint, ST_X, ST_Y
- Access to properties: ST_IsValid, ST_IsClosed, ST_Npoints, ST_IsSImple
- Edit geometries: ST_AddPoint, ST_Multi, ST_Translate
- Output geometries: ST_AsText, ST_AsKML, ST_AsGML

The lecture elaborates on the various functions, and how they can be used to create, or manipulate the data.

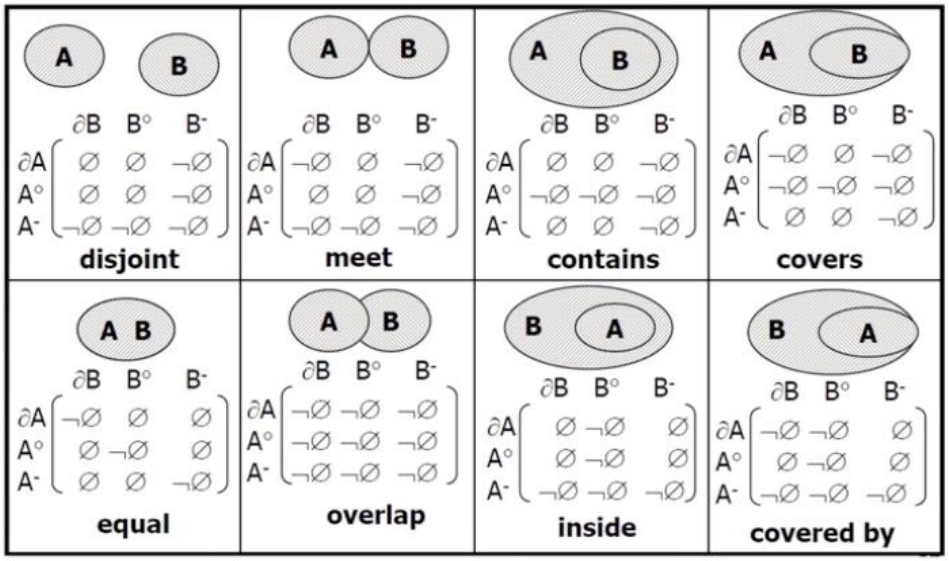
Module 3.

Lecture 3.a: Overlay Analysis

In Lecture 2.1, we discussed how layers can be visualized on top of each other. By doing so, you can perform various overlay analyses on both vector overlays and raster overlays. For each specific overlay, you perform different computations.

- Vector overlay:
    - **Attribute Relations:** Like relational databases, spatial data is stored in **rows** (I.e., entities) and **columns** (i.e., attributes). In QGIS, you can use the **Field Calculator** to add, edit or remove fields. To study uniquely identifiable rows, one of the attributes must be a **primary key**. We call this an **attribute relation**. You can join relations by merging the data of both relations on the primary key.
        - In QGIS, an Attribute Join can be done by using the **Join Attributes** function.

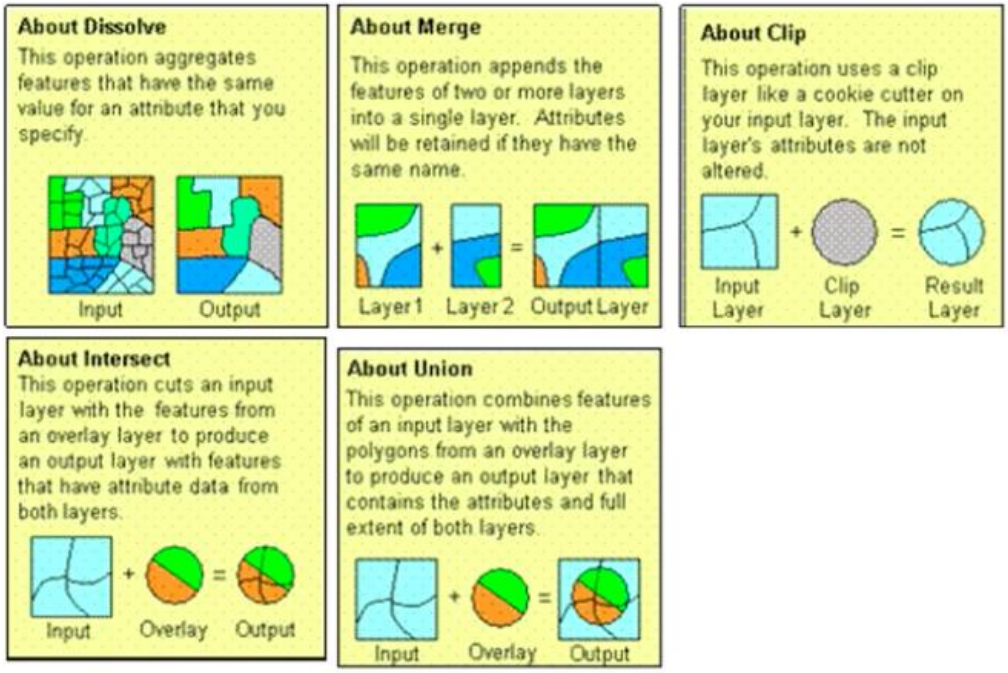
    - **Spatial Relations:** In addition to these attribute relations, spatial databases can make use of **spatial relations** to join relations, or in other words, construct new polygons**.** Such manipulations therefore use the values (I.e., boundaries and inner coordinates) of the two layers of polygons to compute a new entity. **Spatial Join** joins attributes from one feature to another based on a spatial relationship. The target features and the joined attributes from the join features are written to the output feature class.
        - In QGIS, a Spatial Join can be done by using the **Join Attributes by Location.**



^ An example of the various computations that can be used to establish new relations that create new layers.

^ An example of **spatial join**, where two overlapping layers, one being a point layer, and the other being a field layer, are written to a new feature where the two layers are joint.

- **Other overlay operations:** Additional operations can be applied to layers that either dissolve (or aggregate), merge, clip, intersect or union the layers. Here, the inputs cannot be points, but must be **lines** or **polygons**. From this, new lines or polygons are outputted.
  - In QGIS, these operations can be used by selecting the appropriately named functions (I.e., **Dissolve, Merge, Clip, Intersect and Union)**



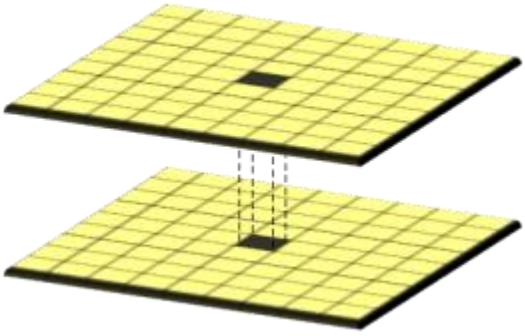
^ Examples of the other **overlay** operations dissolve, merge, clip, intersect and union to define new layers.

- **Areal Interpolation:** While this course uses a generally simplified approach of interpolation, it can be understood as an area weighted average. There are more sophisticated techniques to do so which will be discussed in the lecture on Distance-Based Analysis.

- Raster overlay: (https://gisgeography.com/map-algebra-global-zonal-focal-local/)
    - **Map algebra**: A **cell-by-cell** combination of raster data layers using algebraic operations. These are simple operations that are performed on the numbers that are stored as values within the raster cell locations. The basic elements of map algebra are:
        - **Objects** (I.e., datasets, layers, and values (as input or storage)
        - **Operators** (I.e., addition, subtraction, multiplication, etc.)
        - **Functions** (I.e., loc, foc, zon, glob)
        - **Actions** (I.e., the results of applying such functions)
            - **Qualifiers** on the actions decide the functionality of the action.

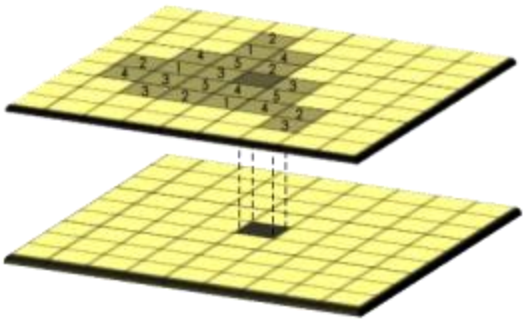
      Such functions are said to be **higher-order functions** as they take an algebraic function and apply it to a set of cells. This set of cells can be **local** (I.e., cell-by-cell), **neighborhood/focal** (I.e., moving neighborhood), **zonal** (I.e., within a homogenous zone), **global** (I.e., the full dataset).

        - In QGIS, use the **Rasterize (vector to raster)** **(I.e. Polygon to Raster)** function to perform convert vectors to rasters. You select the attribute in the input vector that should be written into the cell values of the raster.
        - In QGIS, use the **Reclassify by table** **(I.e. Boolean reclassify)** function to turn a raster in a new raster where each cell value is changed to a new value based on values from another table.
        - In QGIS, use the **Raster Calculator** to write down map algebra, where you can perform the above-named operations.
        - In QGIS, use the **Zonal Statistics** to take the vector polygon layers that defines the zone, and the raster layer, to compute a new raster layer that uses the zone of the vector polygon layer to calculate new values for the cells that lie in those zones. In QGIS, use the **Processing Modeler** to save/ rerun processing workflows for raster and export them into Python.

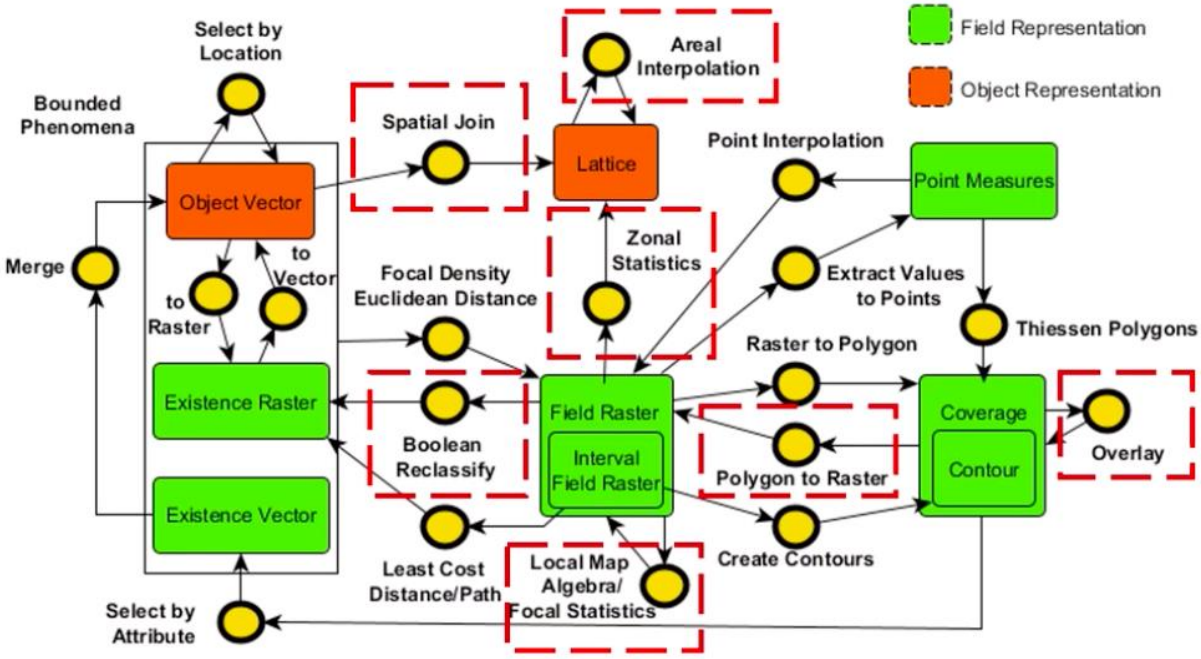


1	4	5		5	1	3		6	5	8
5	3	2	+	1	2	1	=	6	5	3
2	5	2		1	4	2		3	9	4

^ An example showing local map algebra.



^ An example showing zonal map algebra. (Neighborhood & global next lecture)

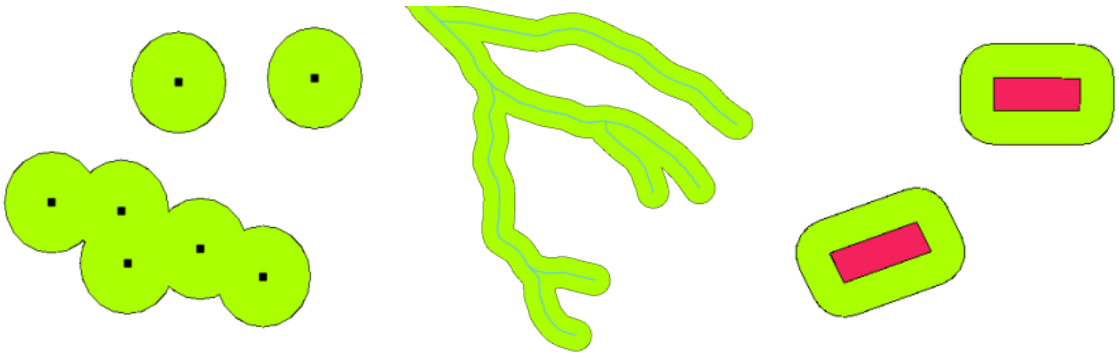


^ These are the manipulations that we have performed in this lecture. The manipulations are highlighted as red in the summary above.

Lecture 3.b: Distance-Based Analysis

- Vector Distance-Based Analysis
    - **Planar** distances are measured in a **projected coordinate system (CRS)**. These distances, however, are **distorted** on the spherical earth surface, or in other words, the **geodesic** distance. Therefore, one needs to account for varying distance measurements over the different planar and spheroid models. As QGIS does not support measurement on spheroid models of the Earth, but only Cartesian models instead, you need to take this into account when calculating these distances.

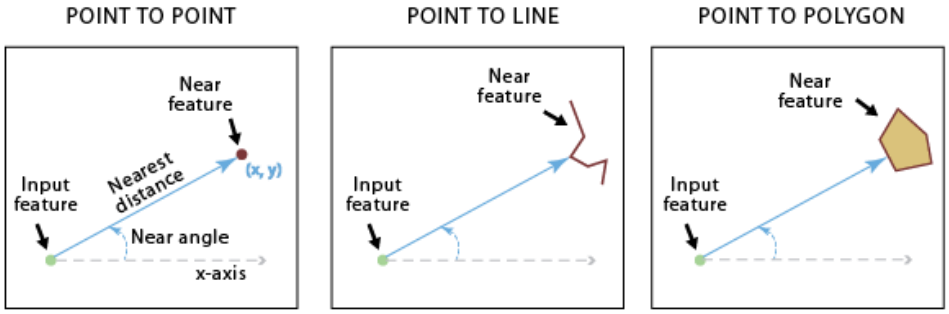
    - Three distance-based methods/ proximity analyses include:
        1. **Buffers**: A buffer creates two areas: one area that is within a specified distance to the selected real-world features, and one area that is beyond the selected real-world features. The area within the specified distance is called the **buffer zone**. A *buffer zone* is any area that serves the purpose of keeping real world features distant from each other (e.g., greenbelts between residential and commercial areas, border zones between countries). These *buffer zones* are vector polygons that enclose the *buffer points, lines, or polygons*. The **buffer distance** or buffer size **can vary** according to numerical values provided in the vector layer attribute table for each feature. The numerical values must be defined in map units according to the Coordinate Reference System (CRS) used with the data. For example, the width of a buffer zone along the banks of a river can vary depending on the intensity of the adjacent land use. For intensive cultivation the buffer distance may be bigger than for organic farming.Buffers can be **merged/ dissolved** into a single geometric object to avoid overlapping areas, which is recommended to minimize the number of geometric objects.
            - In QGIS, use the **Buffer** function to perform Buffer operations onto the data.



        2. **Nearest objects:** The 'distance to nearest', 'nearest object', or 'least cost/ distance path' calculates the distance and additional proximity between the

source/ input feature and the closest feature in **another layer**. This can be done for points, lines, and polygons.

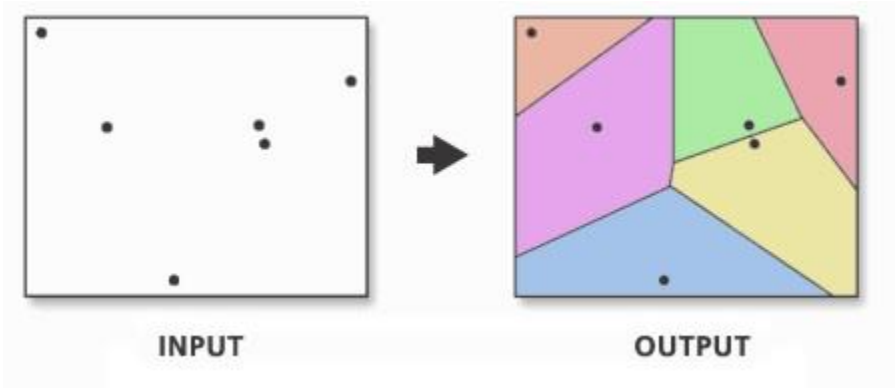
- In QGIS, use the **Distance to Nearest Hub** function to calculate these distances.



^ Three proximity calculations from point to point, point to line, and point to polygon.

3. **Thiessen Polygons:** Sometimes also known as Voronoi Diagrims/ Dirichlet tessellations, these polygons exactly bisect distances between **points** using a Delaunay Triangulation. A polygon is constructed around the point, splitting the area up into **polygons**, rather than **points**. So, the **points** within a polygon are those points that lie the closest to its particular center point, rather than to points outside of this polygon. This can be used to construct catchment areas or to interpolate measures.

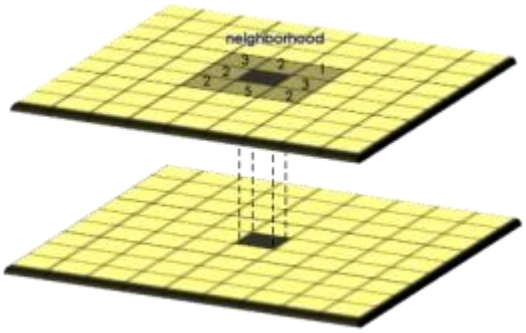
- In QGIS, **Thiessen polygons** can be made using the **Voronoi Polygons** function.



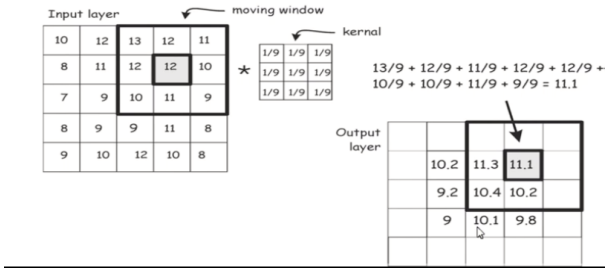
^ An example of how Thiessen polygons are constructed around points, where the values within a polygon lie the closest to the center point within that polygon.

- Raster Distance-Based Analysis: Whereas local and zonal maps were discussed in the previous lecture, the distance now increase. As a result, different computations are required for these map algebras.

- **Focal/neighborhood map algebra:** The f**ocal/neighboorhood function** (foc) is performed on cells within a **moving cell neighborhood**. A moving window is a rectangular arrangement of cells that **shifts in position**. By applying an operation to **each cell** from a moving window, it commonly smooths values in a raster. Focal functions can be of different shapes. A **kernel** is a set of constants for with the values within a window are multiplied. Kernels are used to find concentration or elevation changes.

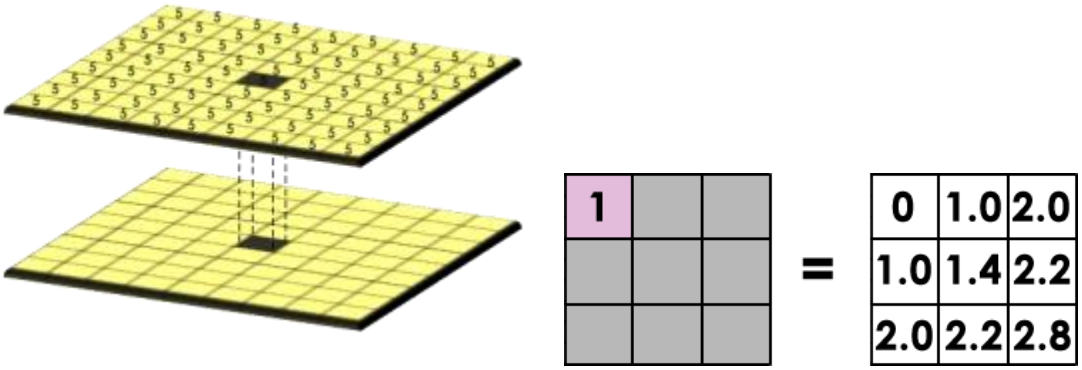


^ An example showing focal map algebra.



^ An example of a moving window and a kernel

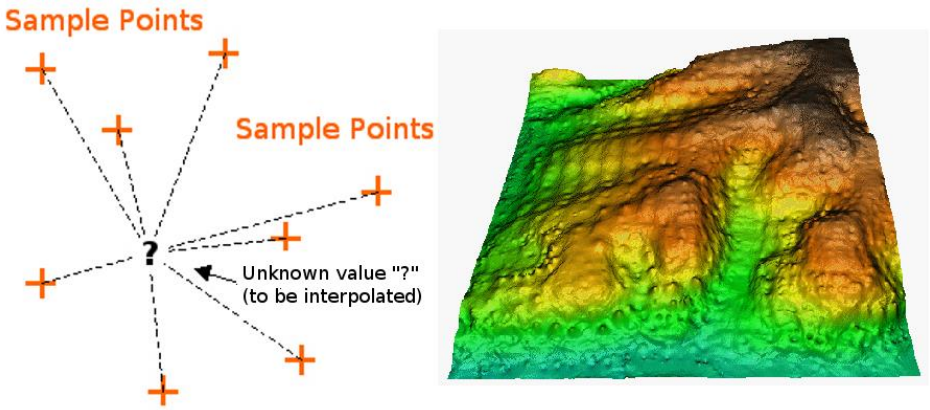
- **Global map algebra:** Global operations (glob) apply a bulk change to **all cells** in a raster. If you want to add a value of 1 to all grid cells, this is a global operation. For example, Euclidean distance is an example of a global operation. By calculating the closest distance away from a source, it applies the function globally in a raster.



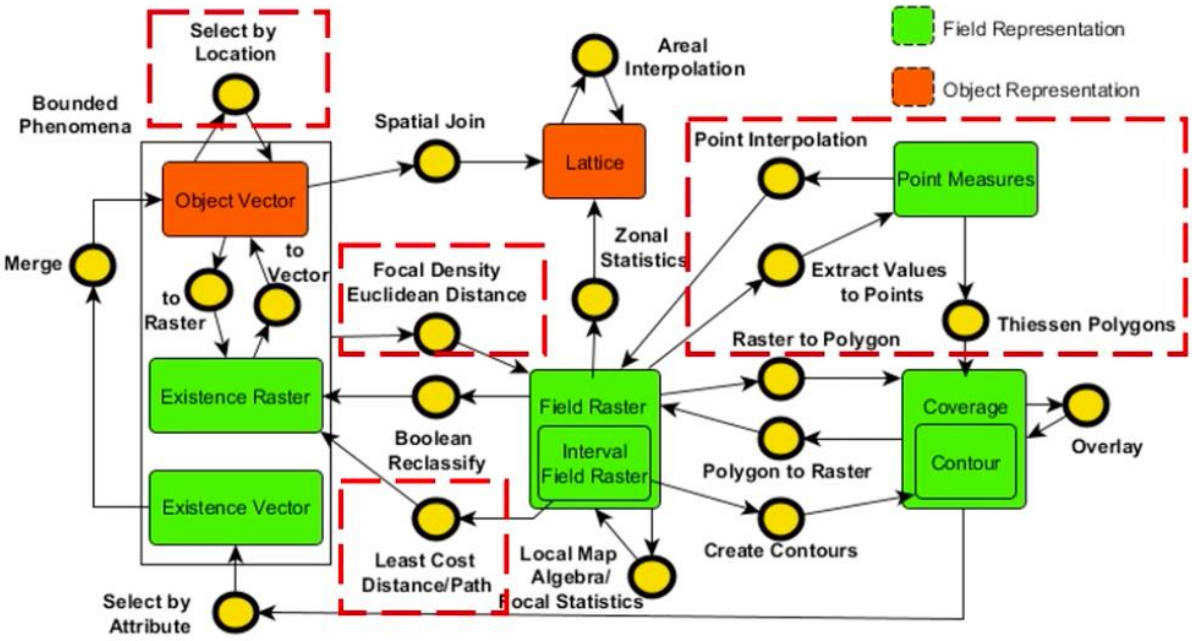
^ An example showing global map algebra. Values increase with distance of the source raster.

- In QGIS, you can calculate the proximity by using the **Proximity (Raster Distance)** function.
- In QGIS, you can calculate the focal statistics using the **r.neighbors** function.
- In QGIS, you can calculate the kernel density by using the **Heatmap (Kernel Density Estimation).** On the map, it is called **Focal Density**.

- **Point interpolation:** Interpolation **predicts** values for cells in a **raster** from a limited number of **sample data** points. It can be used to predict unknown values for any geographic point data, such as **elevation**, rainfall, chemical concentrations, and noise levels. **Inverse Distance Weighting (IDW)** uses the distance, where an increasing distance causes the factor to diminish. Closer points weigh heavier than more distant points. A **Kriging** is more sophisticated than IDW, giving a smoother picture, and adds a standard error. In IDW, you can change the parameters.
  - The lower the exponent n, the more distant points affect the interpolation. The greater exponent n, the more local points affect the interpolation.
  - The search neighborhood can be constrained to for instance 5nn, 12nn, and 25nn. The lower the nn, the less sharp the image as the averaging factor will be less.
  - In QGIS, you can apply **IDW** by using the **Grid (IDW With Nearest Neighbor Searching Function)**.



^ An example of point interpolation.



^ These are the manipulations that we have performed in this lecture. The manipulations are highlighted as red in the summary above

Lecture 3.c: Spatial Network Analysis

Spatial Network Analysis generally play a big role in spatial analysis. This lecture contains the basic concepts of spatial networks, accessibility analysis, and flow analysis.

Basic concepts for spatial networks

As discussed in lecture 2.a, we saw that **networks** are understood to be a particular **core concept** within spatial data. Networks are understood as **quantified relations between objects** and are therefore more than just embedded graphs. These quantifications can be **extensive** (e.g., flow) or **intensive** (e.g., distance) **between the networks** but can also be extensive (e.g., amounts) or intensive (e.g., distance to nearest node) on the level of the **individual objects**. Other measurement levels might also be applicable (e.g., boolean).

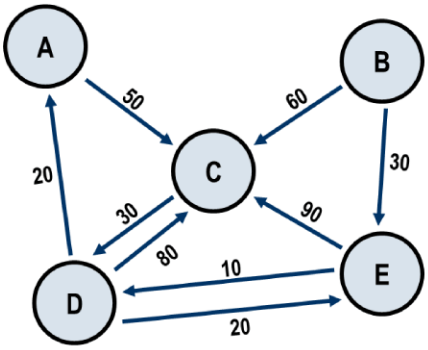
Network datasets generally consist of a geometric network, which contains the geometric points and lines of a network. These lines are each connected to end points to create a network. While the network can be visualized, this datafile however, does not store which points and lines are connected to each other.

A **logical network**, in turn, contains the neighborhood information (I.e., junctions) between nodes and edges. The nodes and edges are projected in a connectivity table. Question for lecture: **How are these two tables joint? On which attribute?**

These datasets can be used to calculate (shortest or quickest) paths in, for example, street networks in the study of transport network analysis. To calculate either the shortest or quickest (I.e., distance and speed) paths, you can make use of **Dijkstra's algorithm**. The results between shortest or quickest paths can vary significantly, when for example a highway is used to speed up the drive. By using this algorithm, you can compute zones or **catchment areas** (to allocate services (I.e., medical services) to a network)

An **O/D (= origin-destination) matrix**, can be used to visualize **spatial interactions**, showing what the distances (but also amount of flow) are across edges to nodes. These are essential in spatial network analysis to show costs within networks.

Spatial Interactions

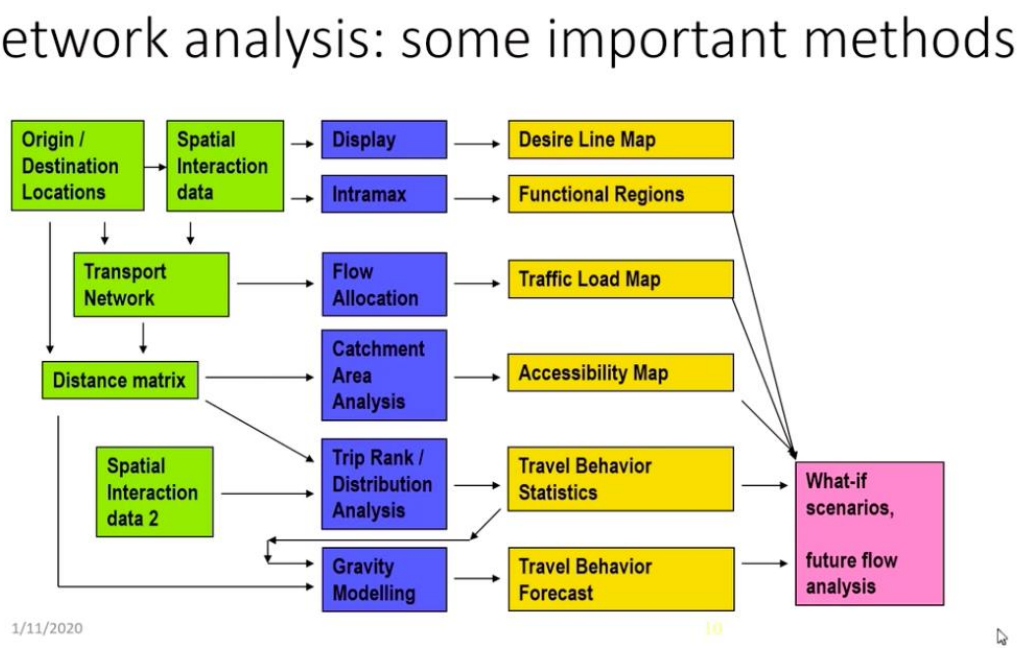


O/D Matrix

	A	B	C	D	E	Ti
A	0	0	50	0	0	50
B	0	0	60	0	30	90
C	0	0	0	30	0	30
D	20	0	80	0	20	120
E	0	0	90	10	0	100
Tj	20	0	280	40	50	**390**

^ An example of spatial interactions and their suitable O/D Matrix.

Methods in network analysis include:



^ An overview of spatial network methods.

Methods transform between intensive/ extensive object and network quality. For example, **catchment area methods** transform intensive network qualities (e.g., distance) with extensive objects (e.g., service potential, origins) into intensive object qualities (distance to the closest service. Also, **gravity models** transform intensive network qualities (e.g., distances) between extensive object qualities (e.g., number of residents) into extensive network qualities (e.g., flows).

Accessibility analysis

Accessibility analysis focusses on **distances** in networks. **Catchment areas** are areas from which a city, service, or institution attracts a population that uses its services like for example emergency centers such as fire departments, police departments, ambulance bases and hospitals. By first plotting the nearest neighbors and calculate the catchment areas around those regions, you can then plot the distances for areas towards such services. Specific colors would then correspond with distances, showing which areas have less access to services in comparison to others.

Thiessen polygons work with the Euclidian distances, instead of working with network distances. As a result, inaccessible areas such as rivers, canals, or roadless areas are included. Catchment areas instead, look at the network constructed through the streets and junctions. Question: **What exactly are the thresholds?**

Flow analysis

Flow analyses show the **flow** of values across geographical areas. For instance, **desire line maps** are maps where the lines represent movement of people or goods between regions. This could for instance be used to study refugee flow towards certain countries across different years. **Gravity models** estimate flows from object amounts and distance networks. It therefore does not take flow as input layer. In

addition, **trade area analysis** in regions calculates what percentage of the overall flow goes towards a service center.

Module 4.

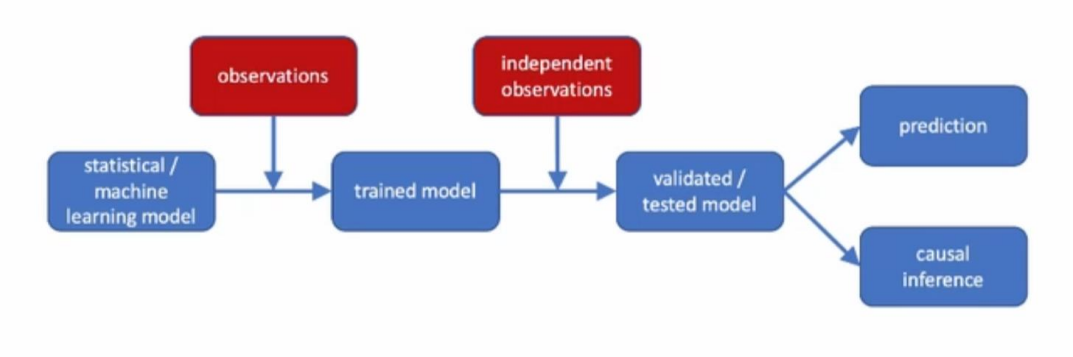
Lecture 1.a: Introduction to Simulation Modelling Modules

During the first part of the course, we largely analyzed spatial data. In the second part of the course, we will build upon this knowledge and focus on **simulation modelling**. This lecture serves as an introduction to the final modules of this course.

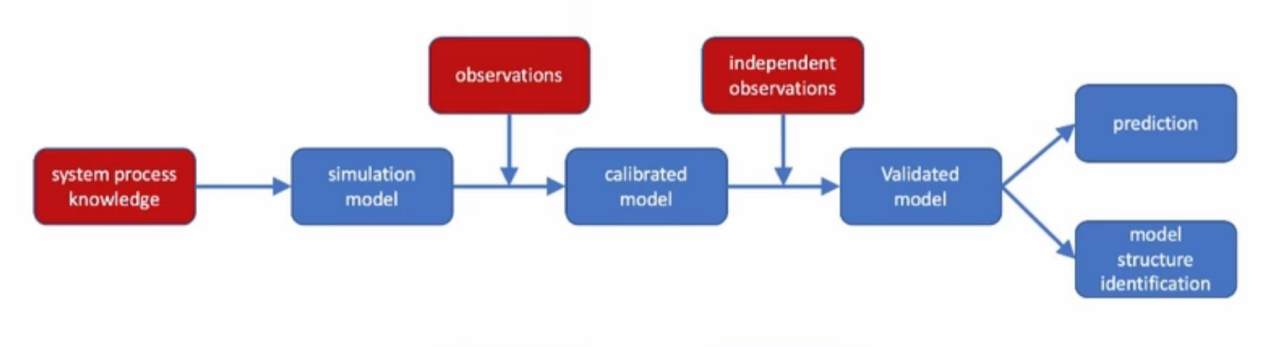
Throughout these upcoming three modules we will focus on **field-based modelling** (I.e., phenomena that are continuous fields such as vegetation biomass, temperature), **spatial agent-based modelling** (I.e., phenomena are discrete objects such as cars, trees, and people), and **model calibration** (I.e, 'fitting' or 'tuning' a model to observations).

The labs are slightly different from previous labs. The labs are found underneath '**communities'** on Blackboard. For the first module, you must answer questions in Blackboard, and do not have to hand in anything else. After answering the multiple-choice questions, you can immediately see the **feedback** of answers in the form of scripts. When you get stuck, just make these exercises so you can receive the correct script. For the other two modules, you must enter answers in a **text document** and upload them to Blackboard after finishing the labs.

Statistical learning and simulation modelling consist of multiple components. **Statistical learning** (or statistics or machine learning) is used to make **predictions** as to make **estimations** of a certain variable at unvisited locations or moments in time, based on observed previously observed data. **Simulation modelling** (or forward modelling, numerical modelling) aims to achieve similar ends in the sense that you aim to predict results, but here you not only rely on both observations, but **also** the **knowledge of system processes** (I.e., mechanisms that mimic processes in the real world). Here, data work as input to simulation models, but also as a means to **tune parameters** of such models.



^ A schema showing the steps in a classic machine learning process



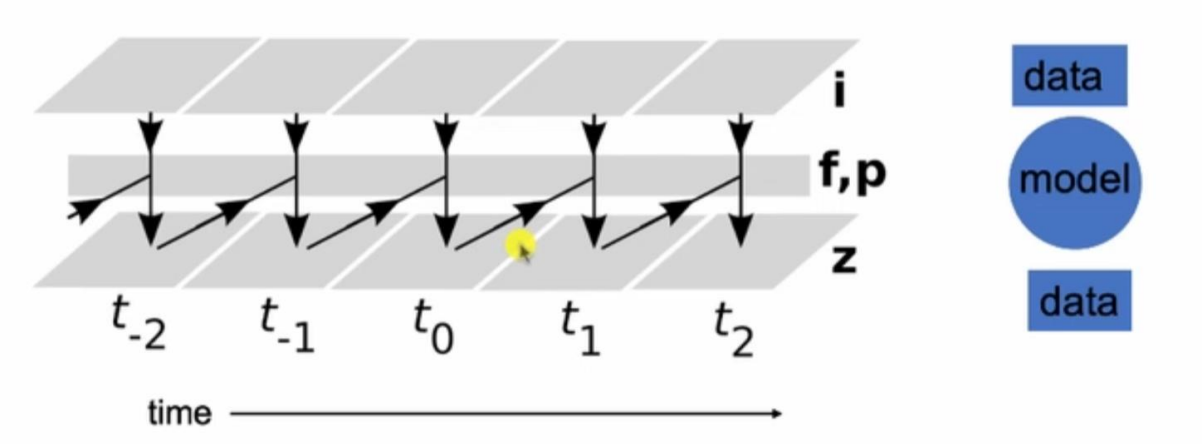
^ A schema showing the steps in classic simulation modelling. Notice the slight changes in terminology used to describe similar steps (I.e., training a model vs. calibrating a model).

Lecture 1.b: Introduction to Simulation Modelling

Simulation models:

A **simulation model** is also known as a numerical, forward, dynamic, or dynamical model. These terms have a slightly different meaning, but all refer to models that **mimic processes** that take place in a **spatial-temporal system**. In a model, a **state** refers to all the processes that take place within that system, such as for instance rainfall, runoff, infiltration within a hydrological model. This state is continuously updated over time by a **timestamp** which is defined by a function. Simulation models are used in almost every discipline, such as ecology, land use science, climate science, epidemiology, and more. For instance, land use predictions are important for management and policy planning of areas as ecosystem surfaces and water resources are influenced. How long does it take for water to reach a potential living area when a dyke is breached?

A simulation model can be understood in the form of a rough equation: $\mathbf{Z_{t+1} = f(Z_t, p, I)}$, where Z are the state variables, p are the parameters, I are the inputs, and f is the transition function that functions as a timestamp. This equation is exemplified in the following image. **Timesteps** are selected on the model builder based on the process that one is trying to simulate. **Inputs (I)** refer to drivers or boundary conditions of a model, and are given data that are being fed to the simulation model (e.g., rainfall). Such inputs generally vary over time. **State variables (Z)** are calculated by the simulation model, and are products of the simulation (e.g., river discharge). These calculations are performed by the **transition function (f)**, which contain the code of the simulation (e.g., infiltration equation in a hydrological model) and contain the rules that steer the changes over time. These rules need to be parameterized (I.e., what the data can or cannot be), and are contained within the (usually fixed) **parameters (p)** (e.g., Infiltration capacity of the soil).



^An overview of how simulation models can be understood. It shows that simulations can be made in the past, the present, and the future.

Relevance of simulation models:

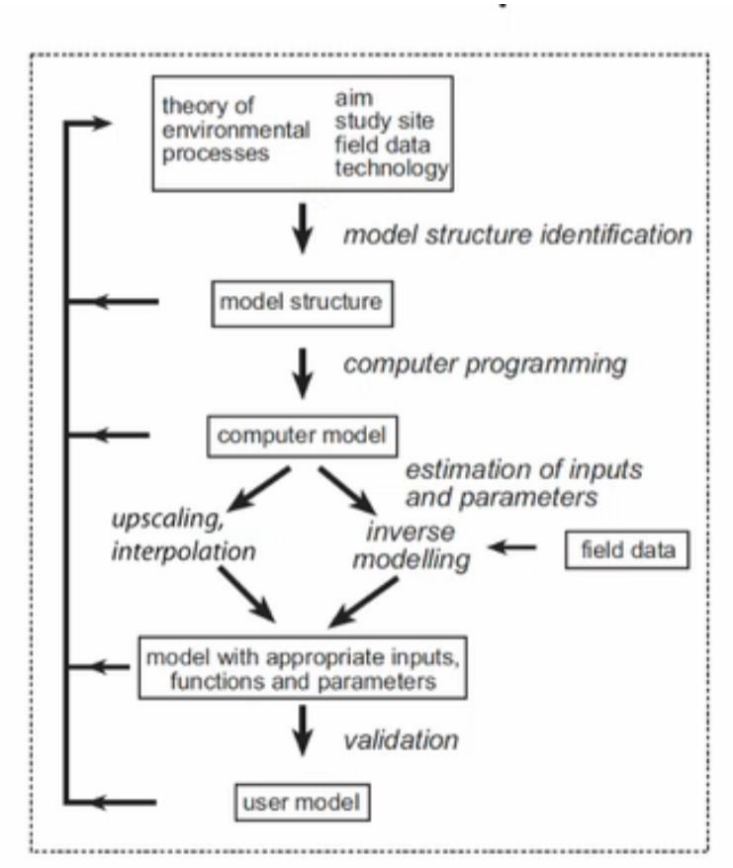
Simulation models are relevant for **three reasons**:

1. They **formalize process-knowledge** of particular aspects in the world. It combines and often converts the theory (I.e., the qualitative knowledge about processes) towards a quantitative mathematical equation that puts the theory into practice.
2. Simulation models allow us to **couple multiple processes together** and understand the system as a whole (e.g., in land use: population growth, suitability of land uses, spatial interactions between components, for instance the construction of infrastructure).
3. Simulation models can both **be run** with data, but can also **be trained** with data. Data can work as input for data and they can **calibrate** the model to fine-tune it, but they can also assimilate new data or at least, predict data.

A model is often regarded as a **mediator** between theory and data. As theory is often an abstract collection of information, data encompass the raw observations of processes within the world. You can test the theory by feeding both of these components into a model.

<u>Model development cycle:</u>

When building a model, there are multiple steps. Each model is dependent on the research question, the environment, and the processes that take place in a particular environment. Therefore, models need to be carefully designed so these fit the requirements of the model.



^ An overview demonstrating the model development cycle.

The steps of modelling include:

Step 1: Model structure identification.

Here, you convert the theory of environmental processes, the aim of the model, the study site, the field data, and the available technology in a model structure. This conceptual model can consist of graphical description, a description in words, but preferably also tables or equations (since these can be converted to code). Furthermore, you need to select the type of modelling: differential equations, cellular automata, probabilistic modelling, rule-based modelling, or agent-based modelling.

Step 2: Computer programming.

A model needs to be coded with a proper tool. These could be done with generic programming languages, but fortunately simple tools such as spreadsheets could be used. There are also modelling toolboxes specifically used for constructing models.

Step 3: Estimation of inputs and parameters.

Some inputs can immediately be directly estimated in the field. You can also use calibration (I.e., inverse modelling), where you measure model output and compare that with the simulated output and tune the model until it matches the output that is being generated.

Step 4: Validation/ evaluation

Here, you test the model against an independent data set to ensure that your model simulates behaves appropriately and generates the correct outputs.

Lecture 2.a: Introduction to Map Algebra

PCRaster is a package of software for constructing **space-time dynamic environmental** models. For this application, the spatial domain is visualized through 2-dimensional maps, which are **raster** based. A wide range of models have been built with the PCRaster software, such as:

- Rhineflow (I.e., modelling the discharge of the Rhine river)
- Lisflood (I.e., run-off model of large catchments in Europe)
- Ecological models (I.e., plant dispersal)
- Sedimentological models (I.e., river flood plan evolution).

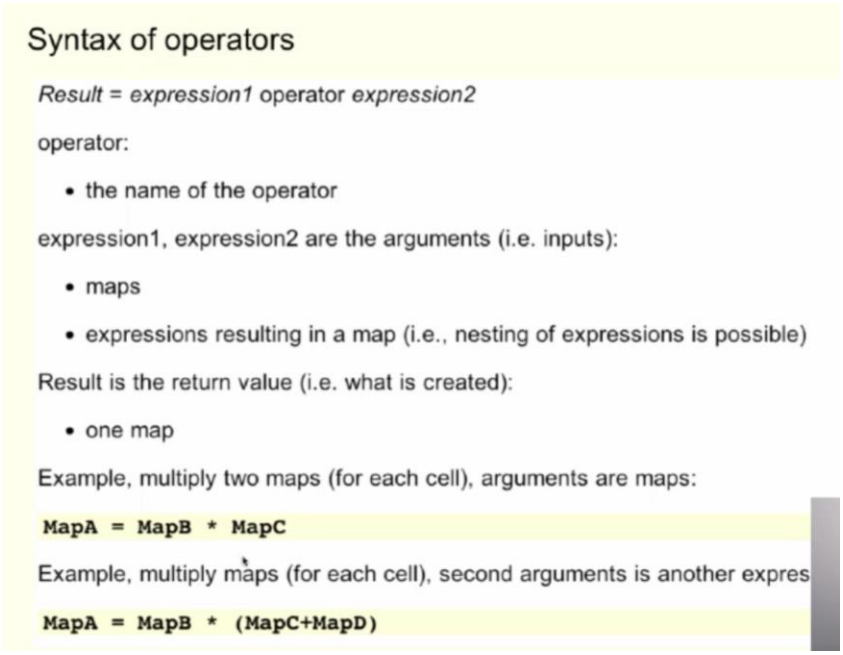
PCRaster has some GIS functionalities, but not a lot as compared to QGIS. Still, you can resample maps to other cell sizes or areas, convert data with standard GIS systems, and interpolate data. PCRaster does not support **digitizing**, **vector data**, or **fancy printing facilities**. As PCRaster cannot store large quantities of data, this will have to be done in for instance R. It is purely intended for **modeling**.

Within models, multiple entities are used.

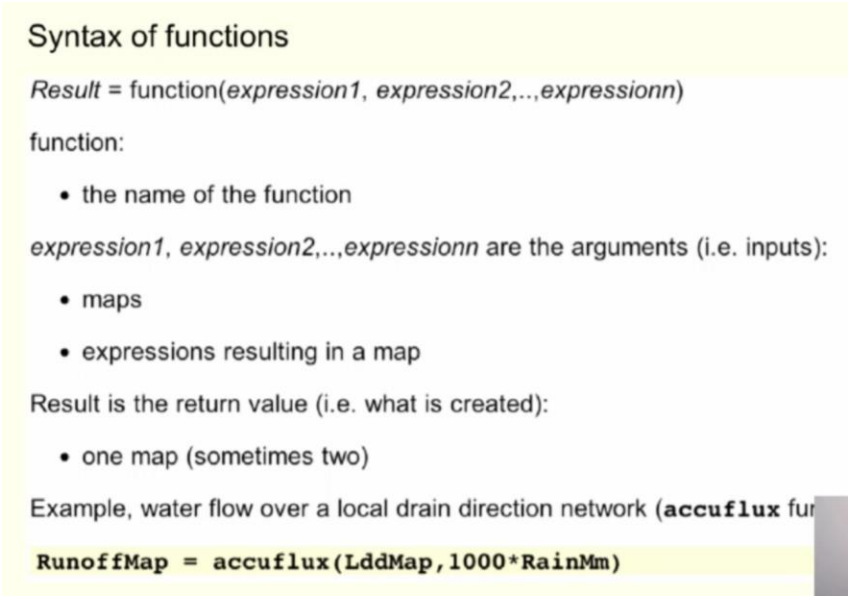
- **Maps** are the main entities that contain the data and save the data in a binary format. These maps are supported in other pieces of software aside from PCRaster, so you should be able to open a similar map within for instance QGIS.
    - The **grid size of cells** in raster maps are **constant**, meaning that over the entire raster the cells must be of the same size. Despite this, cell sizes can be changed to other values, granted that all the map layers in the same model accord to that same cell size (since you cannot layer maps of different cell sizes on top of each other). The cell size is dependent on the source data, but also the scale upon which you want to model the data. In addition, maps are generally required to be of the **same size**. Any values that lie outside of the study area are considered **missing values** and are represented as black cells.
    - PGRaster takes in several data types:
        - **Continuous data**: Maps automatically construct a scale in case you work with continuous raster data. Continuous data include **scalar** (I.e., real values) or **directional data** (I.e., 0 to 2 pi). For instance, when working with elevation areas, the scale will be automatically adjusted based on the available data.
        - **Discrete data**: **Boolean**, **nominal** and **ordinal** data cannot be encoded as strings but are instead transformed to whole values ranging from 0 to 1 and 0 to 255 respectively.
        - **Local drain direction**: For each grid cells, it gives a flow direction to each of its neighbors. Once done for all grid cells, you can make a **flow network** using local drain direction when studying catchment areas or river discharge areas.
- **Timeseries** are entities in models that are used for dynamic models. These entities are used to represent inputs or storing output variables within an .ascii data file. In fact, timeframes can be considered as multiple maps that can be presented as an animation.
- Finally, **tables** are used in point functions to assign new values as a function of several input maps. Tables are manly meant for storing relations between different attributes. They are not as frequently used as maps.

Map algebra can be performed onto these entities to modify the data towards some degree. **Operations** and **functions** are the building blocks of a model, where you can combine map overlays to derive one or two maps from one or more input maps, tables, or timeseries. A basic explanation of map algebra can be found in the lectures in module 3. To reiterate, map algebra are computations performed on raster data that can either take place on a **local**, **zonal**, **focal**, or **global scale**.

In PCRaster, map algebra contains two groups of statements: **operators** and **functions**. Operators (so the modification that you want to apply), can be placed in between the two entities. These operators can be any mathematical application. **Functions** also combine input maps like operators, but the syntax is different. Functions take in multiple expressions which are placed in between brackets. Most of these expressions are spatial operations, contrary to the operations which usually perform grid cell operations. Functions are most suitable when you are working with more complex processes. All these operators and functions can be entered at the command line. More extensive applications of the operations can be found in the lectures on PCRaster Python.



^ An overview of the syntax of **operators**

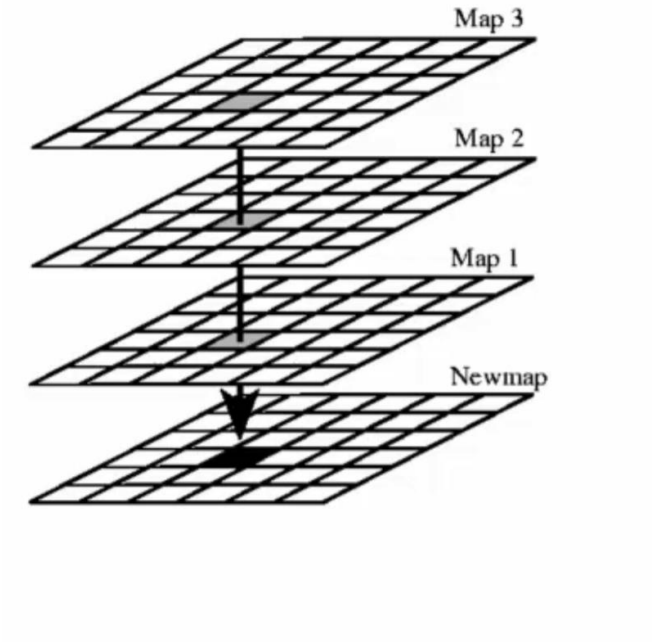


^ An overview of the syntax of **functions**

Lecture 2b. Map Algebra Operations

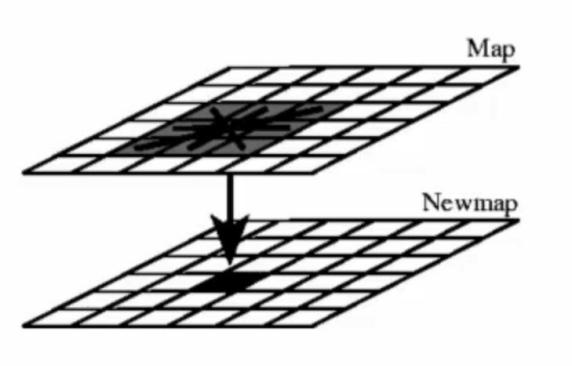
Within the PCRaster software, there are multiple map algebra operations that can be applied to the raster data within a map:

- **Point operations:** In point, or local operations, you only apply particular layers on **one point** on a map. Here, you place the specific operator between the (several, usually two) input layers to create one output layer. Some point operators include **arithmetic**, **trigonometric**, **exponential** and **logarithmic operations**, and can be used by inserting the applicable operator syntax. In addition, **Boolean** and **comparison operators** (<, <=, >, =>) are also applicable on continuous values. Based on such comparisons, a system decides whether a comparison is either **TRUE** or **FALSE**.
    - Boolean maps can also be used in **conditional statements**. Such a function takes a conditional map (so Boolean), followed by a second map of any data type. It uses the first input map to either assign an expression, or a missing value to the second map respective of whether the condition is TRUE or FALSE.
        - For instance:
            - riverw = ifthen(river,riverwidth).
        - Or, for instance when you have a third map:
            - river_or_land = ifthenelse(river,riverwidth,land).
    - Boolean operators always use two inputs, and both inputs have to be of a Boolean datatype, so that TRUE-FALSE datatypes are always combined in a new Boolean map. Boolean operators compare the two Boolean maps and the comparison is based on the Boolean statement (&, ~, |, ^).
        - For instance, when for both values it holds TRUE, there is a bridge:
            - bridge = river & roads



^ Example of a point operation.

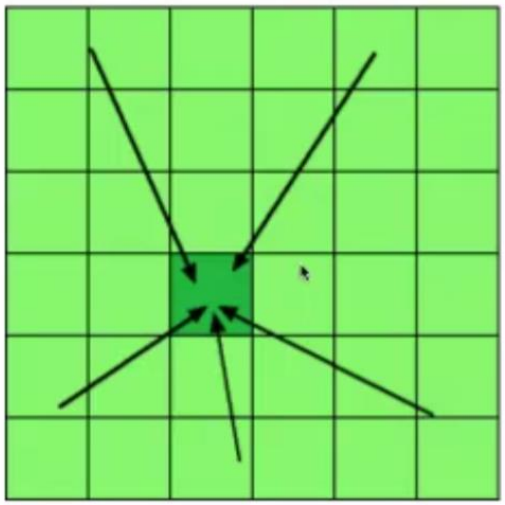
- **Direct neighborhood functions:** Direct neighborhood functions assign a new value to an output map based on the **surroundings** of each grid cell belonging to a neighborhood. Therefore, it is a statement with spatial interaction. Examples of such functions are 'filter' functions and for the measurement of digital elevation models. Such a 'direct neighborhood' consists of multiple cells within a **predetermined area**, and based on the cell values within that area, the system assigns a new cell value to the center cell. This process is applied to each center cell, and therefore it is a **'moving window'**.
    - The syntax of window functions looks like follows:
        - Result = window{insert}(expression, windowlength)
            - **Expression**: variable over which statistics are calculated (e.g., elevation)
            - **Windowlength**: defines the size of the moving window (e.g., 3x3 cells)
            - **Window**{insert}: could be *windowaverage* (mean)*, windowtotal* (sum), or *windowdiversity* (number of unique values), or more.
    - An interesting group of functions are those functions that use elevation models as input. For instance, these can be used to calculate the **slope** of the terrain (I.e., calculating the gradient of a terrain based on the elevation values of surrounding cells in the window). Or, in catchment areas, the **lddcreate** functions generates the **local drain directions** of cells to find the 'flow' of values. This is for example based on the digital elevation model.



^ General example of a direct neighborhood function

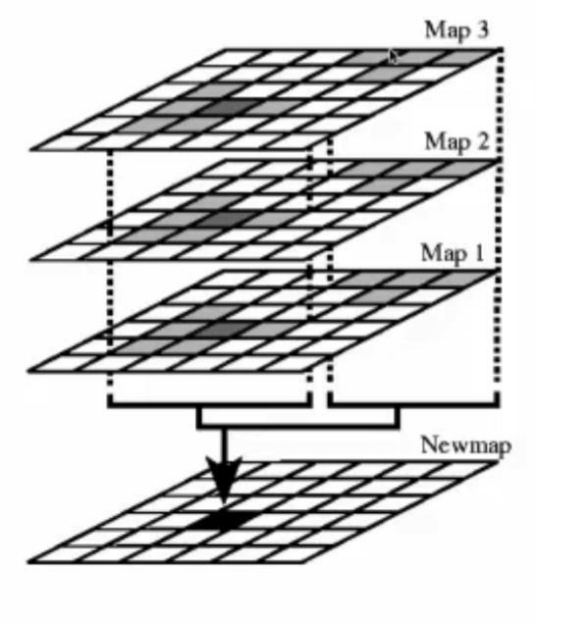
- **Functions with a neighborhood defined by topology:** Local drain direction maps are an example of a function that uses the topology of a map. In other words, it uses the relations that are displayed on the map. Such functions are used to study for instance catchment analysis (I.e. water flow direction) All of these functions make use of the local drain direction map to extract catchment characteristics.
    - Functions to work with ldd, and therefore typology include:
        - Ldd = lddcreate(dem, 1e31, 1e31, 1e31, 1e31) *(to create an LDD)*
        - Pits = pit(ldd) *(to calculate the pits (I.e., outflow locations of LDD))*
        - Catch = catchment(ldd, pits) (*to calculate catchment areas of water using LDD and pits)*

- **Entire neighborhood functions:** Entire neighborhood functions calculate a value based on the **whole area** for each grid cell on the map. There are not many applications, but some of these operations include **distance calculations** (spread), viewshed analysis (visible areas), and groundwater flow.
    - You can calculate the distance towards point by performing the spread function
        - Dist = spread(points, 0, 1)



^ Example of entire neighborhood functions.

- **Functions calculating descriptive statistics:** There are a multitude of functions that calculate the descriptive statistics of areas. They are for the majority described in detail in the lab exercises, so Derrek gave a short summary in the lecture.
    - The syntax of such functions look as follows:
        - AverageInfiltration = areaaverage(Infiltration, Soils)
            - Here, there are two input maps. The second input map is a nominal, discrete data type, and defines the classes over which the calculations are done. The first input, then, is a continuous scalar type. Here, for every soil class, it calculates the average value of the infiltration map and assigns it to all cells in that class.



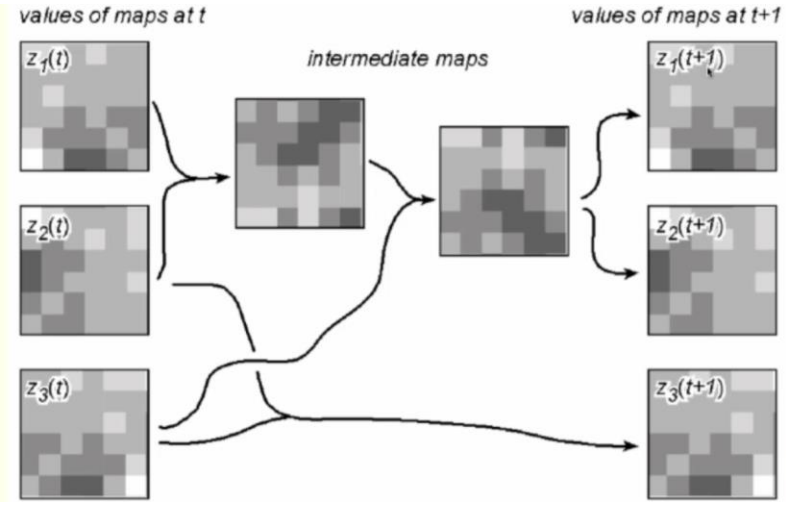
^ Example of how descriptive statistics are used.

Lecture 3a. Dynamic Modelling with PCRaster Python, part 1 - Concepts

N.B. Lot of repetition in this lecture. Previous lecture contains all the necessary information.

As mentioned in a previous lecture, a piece of code needs to be written that stands for the calculations that need to be performed within each instance of the model. This **transition function (f)** should be designed in such a way that it most accurately simulates the real-life process that takes place, and that it is flexible to the various inputs that are being fed into the model so it can calculate an accurate **state**. The transition function has **two inputs**: the **current input (I)**, and the input fed from the **state variables (z)**.

In PCRaster Python software, you can use functions as building blocks that operate on map (data types). The idea is to use the building blocks to update maps, by combining current maps into new maps. These building blocks take input maps, which are combined into intermediate maps, after which new values for the maps are calculated.



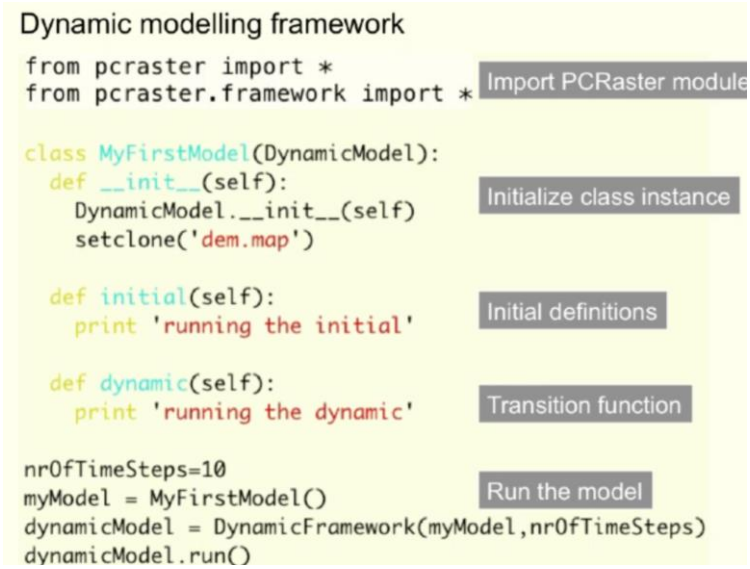
^ A representation of how the transition function is built up, combining the state variables z from the previous state in intermediate maps. Maps are often combined to calculate new state variables z. In simple terms, the map is being 'updated'.

Lecture 3b. Dynamic Modelling with PCRaster Python, part 2 - PCRaster Python Framework

In this lecture, Derrek explains how to use the PCRaster Python Framework to perform iterations over time by demonstrating example code.

He elaborates on an example of Python code containing the import of the required PCRaster module, the initialization of the class, the initial definitions of the functions, the actual transition function, and finally the code to run the model (see images below).

Some parts of the code should not be changed. For instance, the abovementioned pieces of code remain largely the same. However, some other components should be adjusted fittingly to your model. For example, you need to define a **clone map**, which is a raster map that defines the modelling area that you will be working with. You can take any map of your dataset to do this (keeping into account that grid size and map size need to be the same across your maps). Then, the **initial** function refers to the functions that relate to the initial state of your model at the first timestep. On the other hand, the **dynamic** function relates to how the model changes over time. Both pieces of code are relatively short, as these only contain the values that respectively belong to either function. Finally, you have to adjust the **number of timesteps** which determines how many times the model is run. Note that you type **'self'** prior to a variable that needs to be used **interchangeably** between the initial and the dynamic model.



^ What you don't change! An example of code that shows the necessary components when determining a model.

Do not change anything, except..

```
from pcraster import *
from pcraster.framework import *

class MyFirstModel(DynamicModel):
  def __init__(self):
    DynamicModel.__init__(self)
    setclone('dem.map')                    Provide a clone map

  def initial(self):
    print 'running the initial'            Insert map functions

  def dynamic(self):
    print 'running the dynamic'            Insert map functions

nrOfTimeSteps=10                           Provide nr. of timesteps
myModel = MyFirstModel()
dynamicModel = DynamicFramework(myModel,nrOfTimeSteps)
dynamicModel.run()
```

^ What you do change! An example of code that needs to be adjusted dependent on the model.

Example using Python types and operators

```
from pcraster import *
from pcraster.framework import *

class MyFirstModel(DynamicModel):
  def __init__(self):
    DynamicModel.__init__(self)
    setclone('dem.map')

  def initial(self):
    conversionValue = 3.0
    self.reservoir = 30.0 / conversionValue
    print 'initial reservoir is: ', self.reservoir

  def dynamic(self):
    outflow = 0.1 * self.reservoir
    self.reservoir = self.reservoir - outflow + 0.5
    print self.reservoir

nrOfTimeSteps=100
myModel = MyFirstModel()
dynamicModel = DynamicFramework(myModel,nrOfTimeSteps)
dynamicModel.run()
```

^ An example of code where the initial and dynamic section are filled in, showing only a few lines that refer to the values of the state. This model refers to a water basin that is linearly emptying over time, over a duration of 100 timesteps.

Lecture 3c. Dynamic Modelling with PCRaster Python, part 3 - Reading & Writing Model Data

In this lecture, Derrek explains how to read from and write data to disks in PCRaster Python.

- To **store** the data of the map variable aUniformMap, you use the **'report'** statement. You can either do this in the initial section, where only one map is saved, or in the dynamic section, where one map is saved for each timestep. Note that the syntax is the **same** in both the initial as the dynamic section. In the dynamic section, each map is saved according to the **map string**, in addition to a number added at the back of the string to signify the timestep. **Short strings** are recommended when saving files.
    - To store the map 'aUniformMap', you do the following:
        - self.report(aUniformMap, 'uni')
            - The first input is the variable name of the map.
            - The second input is a string under which the map is saved.
- To **import** the data of the map, you use the **'readmap'** statement. The syntax of this statement is the same to the 'report' statement. Here, however, you **do not include** the unnamed variable **map**, so you only use the string that you used to save the map. When loading the maps for multiple timesteps, you can simply load the map and import all the maps according to the number of timesteps assigned to the function.
    - To import the map 'dem', you do the following:
        - self.dem = self.readmap('dem')
            - There is only one input, referring to the string that was used to name the map.

Lecture 4. Neighborhood Interaction

To be added later.

Module 5.

Lecture 1. Spatial Agent-Based Modelling

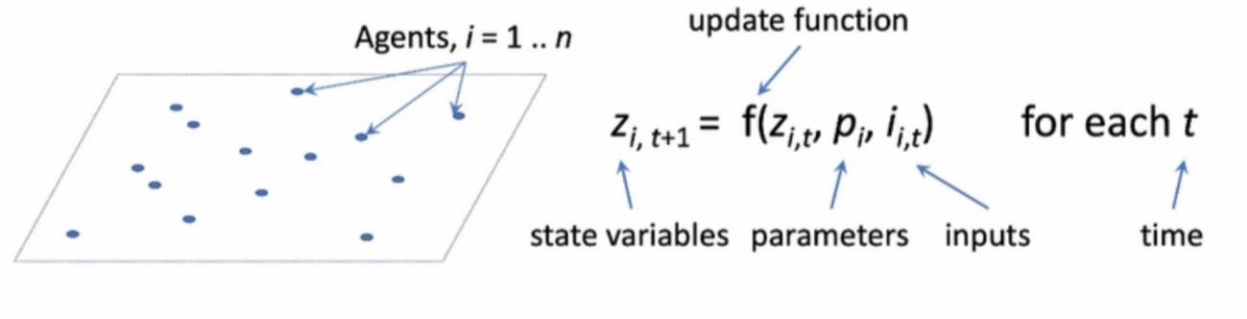
Field-based modelling (I.e., the modelling we discussed in the last module) can be a powerful tool but may exhibit a couple of limitations. These are:

1. Field-based modelling assumes that are modelled phenomena are **spatially unbounded** and **continuous**. This means that for instance, groundwater is everywhere within the modelling area and continuously spread. However, certain items with their own individual behaviors are often bound in space, such as for instance a glacier, or the movement of a ship.
2. Field-based modelling assumes that the **time transfer function** (I.e., the timeframe used for updating a function) is the **same** for all locations and all phenomena within the model. However, a model is often influenced by agents (I.e., farmers exhibit different behavior that led to differences in land use).
3. Field-based modelling assumes that **flow of material or information** occurs across **a two-dimensional field**. However, in reality, there is a flow of material that occurs in for instance as line segments such as water reservoirs, irrigation canals and decision makers who decide when irrigation canals are opened to change the behavior of water reservoirs.

Agent-based modelling (or *individual based modelling, discrete partical simulation*) can facilitate these issues. Here, we assume that there are **spatially bounded agents** (I.e., bounded objects in space that don't exist everywhere). These agents can be **mobile agents** or **agents with evolving geometry**. For instance, farms, water reservoirs, municipalities, glaciers or bird territories can develop over time. These are called **field agents**. Therefore, agents can either be points in space or larger areas, that in turn can be static or mobile. In addition, these agents can **interact** with one another (I.e., agent's decisions influence the other agent's behavior). Finally, agents interact with **continuous fields**, which implies that field- and agent-based modelling can be combined.

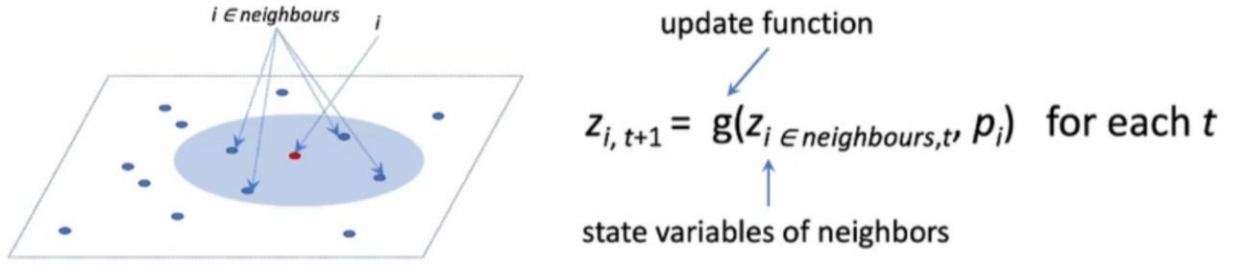
In agent-based modelling, there is a somewhat different transfer function, as the agents often make decisions that are not constrained by timesteps.

Some of the simplest models to simulate in agent-based modelling are **local processes**. This can be compared to the point operations in field-based modelling: there are not interactions between agents. The function looks a bit like the time transfer function from field-based modelling, however now another parameter is added: $\mathbf{z}_{i,t}$ which refer to each agent within the model. Each agent has a set of **state variables** that are property of the agent. These states variables are **updated** that result in state variables for the next time-step. Note that in the function, only agent properties of the agent itself are used as input (so no interactions). Examples are glaciers (snow compaction), companies (reorganization of business), persons (increase in disease risk with age). While the calculations found below apply to point agents, these calculations can similarly be performed to field agents or mobile agents which are understood as larger areas rather than points.



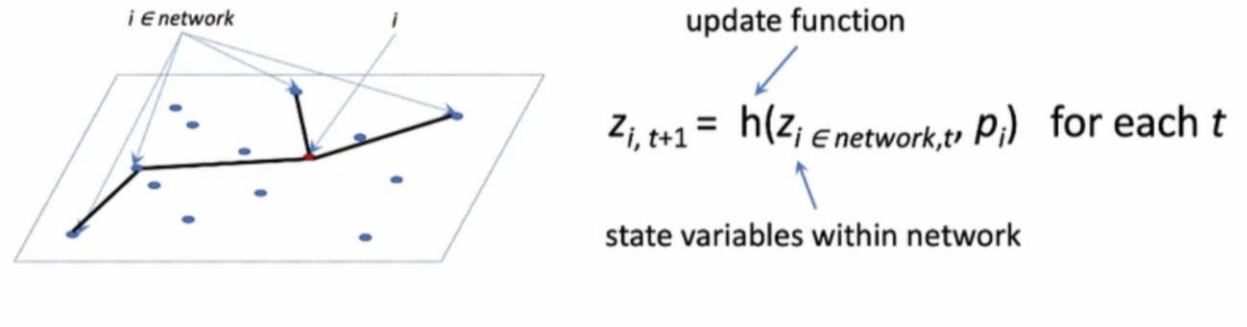
^ A schematized image of local processes.

When these agents do interact with each other, we say that there are **neighborhood effects**. In these neighborhoods, a certain agent is located within a field with a certain distance, and the agents within that field are considered neighbors. The state of each agent in this field is be updated as a function of the properties of these neighbors. Examples are shops (adjusting goods based on goods offered by other shops), house prices (neighborhood house prices affect price of current house), animals (swarm behavior).



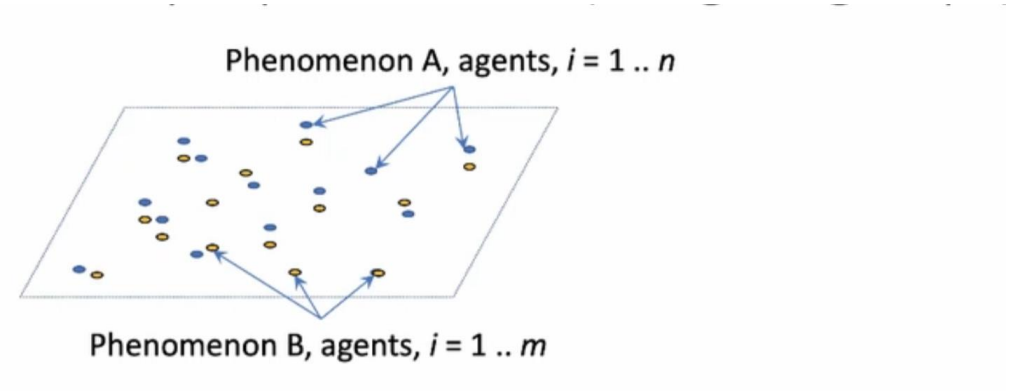
^ A schematized image of neighborhood effects

Where in neighborhood effects we assume that the closest agents within an area affect agents, in **network effects** these agents affect agents that lie closest within a network. These can be direct neighbors in the network, or indirect neighbors in the network. Examples are persons (smoking teenagers influence other teenagers), water reservoirs (inflow determined by management of upstream reservoirs), framers (crop selection based on other farmers' crops).

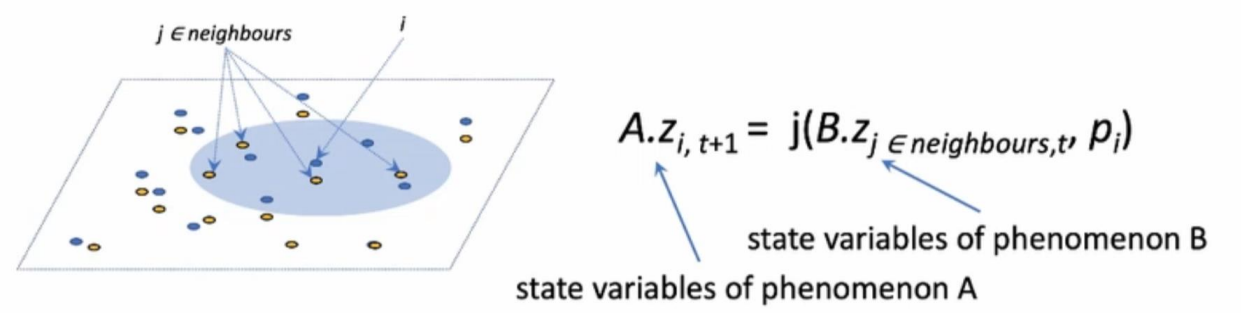


^ A schematized image of network effects

There are often **multiple groups of agents** that interact with other groups of agents. Examples are food outlets and households (food offered influencing diet, diet influencing good offered at shops), predators and prey (food web), individual plants and individual seeds (ecological model).

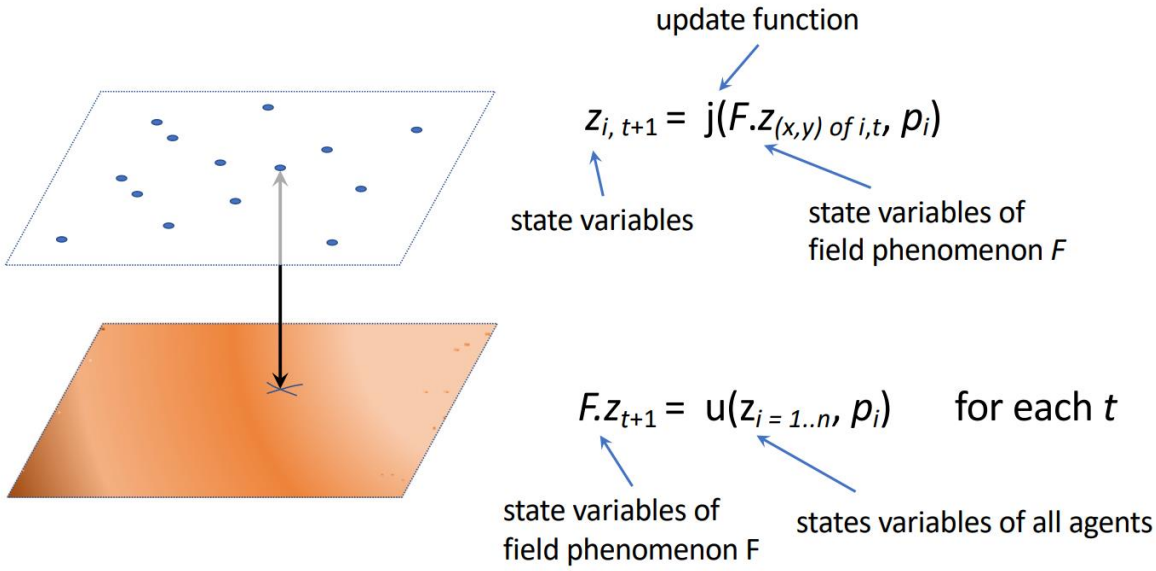


Within a simulation, multiple of the above-described effects can take place at the same time. For instance, **groups** interact only within a **particular neighborhood**. Examples are food outlets and households (people get food available at stores in their neighborhood), predator and prey (predators consume prey in their neighborhood), and water managers and farmers (stakeholder interaction in a water management simulation).



^ A schematized image of how groups interact within a particular neighborhood

Finally, individual agents can also interact with **continuous fields**. By doing so, you can change the state variables of an agent based on the value of the continuous field. In the other way around, you could also change the continuous field based on the agent's state variables. Examples are animals grazing biomass, persons that are exposed to air pollution, or tree biomass as input to soil nutrient stock.



^A schematized image of individual agents interacting with continuous values, and the other way around.

Lecture 2. Campo: Spatial Agent-Based Modeling

Campo is a piece of software which we will use to simulate agent-based models. Campo can be used interchangeably with Python. In this lecture, Derek explains the technical aspects of Campo and how to use it. Note that a lot of the instructions can also be found in the lab manual.

Unlike other agent-based simulation tools, Campo uses two key concepts to approach the simulations. Firstly, it uses fields and agents to simulate software. Secondly, model building is used by applying map algebra on fields and agents. This map algebra is similar to what we have previously seen, where you can simply add, multiply, compare, etc. 'entities', which we will get to know as 'properties' to calculate new 'properties'.

Campo does not really separate between **fields** (I.e., extensive and continuous areas), and **agents** (I.e., bounded and mobile), but instead uses a single **uniform paradigm** to represent both features. This is the case because we could say that 'continuous fields' can also be bounded, although their area will most probably be considerably larger than the agents themselves. Campo distinguishes between fields and agents by **counting** the number of objects within a phenomenon. When the phenomenon contains **more than one object**, these objects are recognized as agents. When the phenomenon contains only **one object**, it is a field.

A phenomenon in Campo is recognized as an entity within a particular environment. These can be defined in a Python environment as follows:

Defining a phenomenon

```
foodenv = campo.Campo()
foodstores = foodenv.add_phenomenon("foodstores")
```

- To define an environment called ‘foodenv’, define it using campo.Campo()
- To add a phenomenon called ‘foodstroes’, define it by adding the phenomenon to the environment.

A phenomenon in Campo contains Property Sets, which are collections of multiple properties of a phenomenon. A property set can be considered as a collection of different properties of a particular phenomenon, which are attached to a particular object in an object to space. To define property sets in Campo:

Defining a property set

```
foodenv = campo.Campo()
foodstores = foodenv.add_phenomenon("foodstores")

foodstores.add_property_set("frontdoor", "foodstores_frontdoor.csv")

print(foodstores.frontdoor)
```

- To define a point property set ‘frontdoor’ of each ‘foodstores’, use the add_property_set function.
- To print the frontdoors, use a dot notation in combination with the phenomenon you are referring to.

Within property sets, you can have multiple properties that can have their individual values.

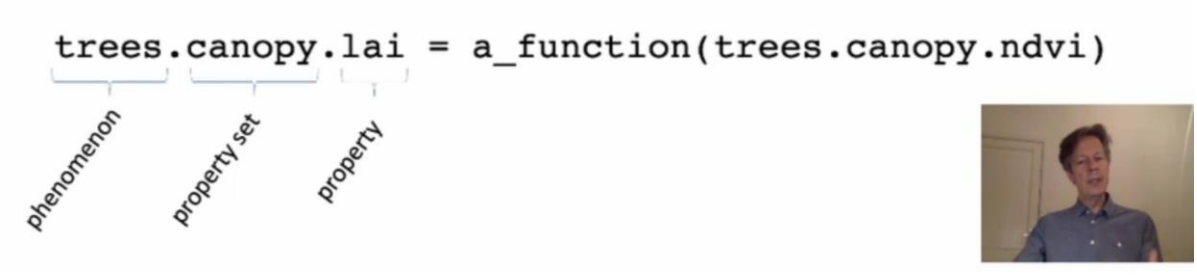
Defining a property

```
…
foodstores.add_property_set("frontdoor", "foodstores_frontdoor.csv")
foodstores.frontdoor.level = 12.1

print(foodstores.frontdoor)
print(foodstores.frontdoor.level)
```

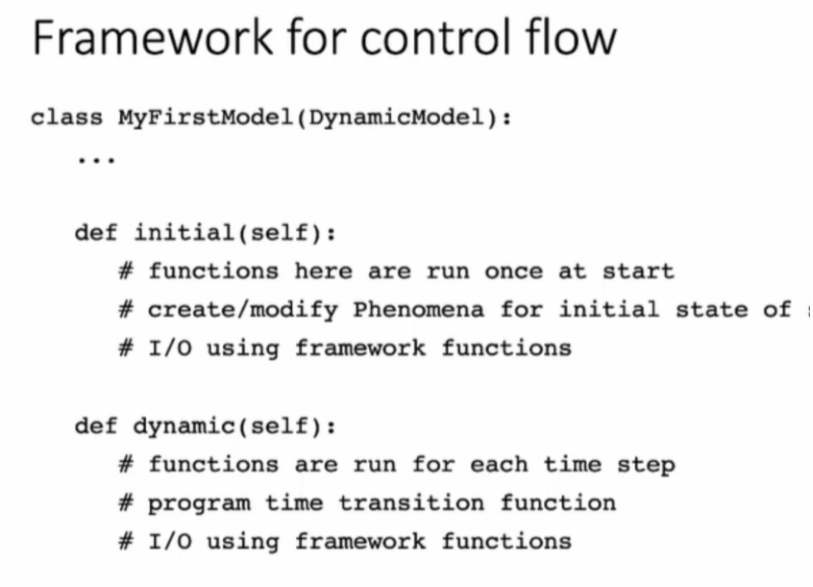
- To define a property for the height (I.e., ‘level’) of a ‘frontdoor’, use a dot notation where you first refer to the phenomenon, then the property set, and then the property that you want to make, following by an equal sign and the respective value that you’d like to assign to it.

To work with these above-described entities in Campo Python, there are multiple Campo operations which you can apply mostly towards the property values of the properties. A benefit of Campo is that you do not have to iterate over all the values within a property, but it will be done for all the values. Functions are various and can be applied to a lot of properties.



^ An example of how to define a new property with values based on a function applied to a previous property (sorry Derek)

The workflow of Campo is very similar to PCRaster, where we have initial and dynamic states.



^ An overview showing the similarities between Campo and PCRaster