**PEOPLE MODULE**

The People module is designed to model the behaviour of citizens within an urban waste collection system. It represents one of the central elements of the project, as it manages the interface between the urban environment (bins, distance, neighbourhood) and human behaviour (production, delivery, environmental awareness).

The aim of this module is to realistically simulate the way in which citizens

* produce waste
* decide whether and how to recycle it,
* interact with each other,
* respond to any incentives from the organisation managing the collection system.

Behavioural decisions are not static, but are the result of a probabilistic combination of internal and external stimuli. In particular, the *green awareness* of each citizen changes over time, depending on neighbourhood behaviour and the level of service received. Behaviours evolve, reflecting a continuous feedback between the environment and the individual.

**Data Class Citizen Type**

This class defines the behavioural profile of a citizen, and is composed of:

* Distinctive group label 'kind' that defines the type of citizen in terms of sensitivity to environmental issues (e.g. eco-friendly, non-eco, etc.).
* Personal attributes of a static nature (e.g. level of education, gender, income, ...)
* Behavioural attributes of a static nature
  + seasonality factor (s) that defines by how much the average waste production increases in seasonal months
  + months of seasonality, a tuple defining the months in which to apply the factor s
  + radius of influence (rof), the maximum distance in metres within which people can influence each other.
  + depth of influence n\_depth, the 'distance' in terms of number of neighbouring blocks within which citizens can influence each other.

Both of these values are given as a range from a minimum to a maximum threshold, when a citizen instance is generated a precise value within this range will be generated. E.g. gaw between [0.7; 0.8], when a citizen instance is generated it may be assigned an initial gaw of 0.75 contained within the specified range.

* Behavioural attributes of a dynamic nature (e.g. green awareness (gaw), a value from 0 to 1 that defines the propensity/probability to recycle)
* Probabilistic functions modelling the behavioural logic of the citizen:
  + gen\_waste: function modelling the amount of waste generated, triggered by a time trigger
  + *green\_inc*: function modelling the increase in gaw based on the level of service, evaluated at each delivery if a bin is available
  + *green\_dec*: function modelling the decrease in gaw based on the level of service, evaluated at each delivery, in the case of a bin not available (full)
  + *f\_influnce*: function that determines the variation of gaw based on neighbourhood influence. Triggered by a temporal trigger; briefly, the logic is as follows, the percentage difference between the gaw of a citizen and the average gaw of the inlfuencers is evaluated. The latter is obtained as a weighted average (based on distance) of the people within the radius of influence (the influencers) of the citizen. If the percentage difference is positive (the citizen has a higher than average gaw) his gaw will tend to decrease, while otherwise his gaw will increase. The specific behaviour is defined by means of probabilistic functions that can be appropriately chosen and parameterised.
  + *f\_incentive*: function that determines the variation (increase) of gaw due to incentives of the collecting agency. Incentives occur at time instants that can be defined at the beginning of the simulation.
  + *f\_dist*: function that determines the variation (decrease) of gaw based on the distance to the nearest bin. This function is the only one that is activated only once, at the beginning of the simulation. The initial gaw of each citizen will then be rescaled (decreased) according to the distance to the nearest bin.
  + *f\_move*: function that determines whether to walk or drive to deliver waste.

**Note.** The functions used can be differentiated and/or parameterised differently depending on the class of the person, the attribute 'kind'. For example, the probability of an eco-friendly person being negatively affected by a service disruption will tend to be lower than that of a non-eco person.

**Class Citizen**

It is the simpy agent that implements the citizen, operationalising the characteristics described by the DataClass Citizen Type. The citizen has spatial, social and behavioural characteristics that determine and influence his or her ability to deliver recycling. Each citizen can be influenced by external factors such as distance to the bins, economic incentives or the social influence of neighbours.

*Attributes and properties*

The citizen has an attribute called 'att' (which stands for attribute) which contains a Citizen Type that distinguishes its social and environmental aspects. In addition to this, each citizen has the following additional attributes that place him/her in the urban environment (the list is not exhaustive, the main ones are given here):

* env, the reference to the simpy environment
* calendar, the reference to a Calendar object which serves solely to convert the simpy time into a concrete date. In practice, therefore, it is possible to track the actions performed by the citizen in calendar time.
* *nd*, the node with which the citizen is associated, a node that corresponds to the centre of gravity of the neighbourhood (or block) of residence.
* *xy*, the Cartesian coordinate that determines its exact location. Location that must remain within a maximum threshold distance with respect to the coordinate of the reference node, (in practice the citizen must remain within the neighbourhood or block to which he has been assigned).
* bins, the dictionary containing all the bins in the network, with their relative distance from the citizen's reference node.
* *infs*, the dictionary containing all citizens that are close enough to influence the current citizen. Each of them is associated with its relative distance. Citizens in this dictionary are referred to as 'influencers'.
* trips, a class that collects data on trips made by the citizen and is able to compile some statistics.
* *gaw*, the actual starting value of green awareness.
* *rof*, the actual value of the radius of influence generated at the time of the citizen's creation and its location in the urban area*.*

There are then some properties/methods that serve to calculate other statistics of interest (e.g. number of times the bin was full, level of service seen by the citizen, etc.). In particular, a special Trip class (assigned to a citizen attribute) is used to map citizens' delivery flows, recording the instant the citizen goes to a bin, the quantity delivered, etc.

*Actions performed when the instance is initialised*

Each citizen is placed in a specific position within the city. The position is determined in the following way: first, the citizen is associated with a node in the graph, representing the centre of gravity of a city block. Next, the citizen's position is determined by placing him or her at a random point within a circle of radius *r*, centred on the reference node. The point obtained represents the Cartesian coordinates (x, y) where the citizen resides. The distance between the node and the point assigned to the citizen determines his or her actual location, which can influence various aspects, such as the distance to be travelled to reach the bins.

Each citizen is assigned all the bins he or she can access, sorted according to distance from nearest to furthest. For each bin, a probabilistic weight is calculated inversely proportional to the distance from the citizen: the nearest bin will have the highest weight, while the furthest bin will have a lower probability of being chosen. These probabilistic weights will be used when the citizen has to decide which dumpster to go to.

Each citizen is also associated with his neighbours, i.e. other citizens who might influence him in some way. For each neighbour, the relative distance to the citizen is calculated, which determines the level of the dynamics that determine the citizens' social behaviour.

As already mentioned, upon creation each citizen is assigned a gaw and rof value consistent with the ranges defined in the corresponding Citizen Type.

*Methods*

The main methods are the following three.

* Confere\_Waste, is the simpy process used to reproduce each citizen's textile waste assignment.
* *Compute\_av\_gaw*, which is used to calculate the average gaw of all influencers.
* *Update\_gaw*, the method that periodically updates the citizen's gaw based on the value of the average gaw of the influencers.

*he* confere\_waste method is a SimPy process that simulates the delivery of waste. The time between two process activations follows an exponential distribution. Each time the event is activated, the citizen decides how to dispose of the waste he has accumulated in the meantime. The amount of waste is generated stochastically, using a triangular distribution, multiplied by (s) in the months of seasonality. The citizen, based on his or her gaw (or rather, with probability equal to his or her gaw) decides whether to recycle or dispose of in the unsorted waste stream. In the latter case, the process ends immediately as undifferentiated waste is collected via door-to-door service. If the citizen makes an ecological choice and decides to recycle, the process continues as follows: (i) the citizen chooses an available dumpster and decides whether to walk or drive to it, (ii) once at the dumpster, if it is not full, the citizen delivers his waste and his gaw potentially increases as a result of the good service experienced, (iii) if the dumpster is full, the citizen goes home and disposes of his waste in the undifferentiated, with a likely consequent decrease in his gaw (related to the disservice experienced).

The *update\_gaw* method is used to update the citizen's gaw based on the neighbourhood influence, word of mouth effect. As already explained, the change in gaw depends on the percentage difference between one's own gaw and the average neighbourhood gaw, calculated as a weighted average, with weight inversely proportional to distance from the influencers. If the percentage difference is positive (i.e. the citizen's gaw is greater than the local average), the citizen will tend to see a decrease in his gaw. Conversely, if the difference is negative, his gaw will tend to increase. The probability of experiencing the effect and the effect is defined by the functions assigned to the Citizen Type that characterises the citizen. This dynamic reflects how social influence and environmental awareness spreads among citizens, contributing to changes in their behaviour in relation to waste management.

**MAP MODULE**

The Map module is used to create the topology of the urban environment in which the textile waste delivery and collection process takes place. The urban map is created by means of a square-shaped grid, in which each node represents the centre of gravity of a neighbourhood or, more appropriately, of a block within the city. Each node is assigned citizens who, as previously explained, are randomly distributed over a radial area centred on the node itself. It is possible to define the number of citizens to be assigned to each node, as well as the percentages of division between the different types of citizens, for example we could say that 50% of the population of a node is an eco-friendly Citizen Type, 30% is a neutral Citizen Type and 20% is a non-eco Citizen Type. In this way, the module does not just generate static maps, but allows complex and realistic urban structures to be modelled. It introduces variability in neighbourhoods, different population densities and probabilistic population distributions, making the urban environment more dynamic and representative of real scenarios.

**Class Urban\_Block**

The Urban\_Block class defines and creates an urban block, describing its main characteristics: including the total population and the distribution of citizens according to their type. For example, we could say that a certain block has a population of enne people (with deterministic or stochastic enne) of which 75% are Eco-friendly and the remainder Neutral.

An example instantiation of an Urban\_Block is shown below:

Very\_Green\_Block = Urban\_Block(population = pop\_nr, type\_prob = {eco:0.75, neutral:0.25}) in which a block is being created with population distributed as defined by pon\_nr (a truncated normal) and a 75% Eco-friendly and the remaining 25% Neutral.

This implementation allows the urban population to be modelled dynamically, allowing each neighbourhood to have different demographic configurations. Furthermore, citizens are positioned in the simulated space in a realistic manner, contributing to a faithful representation of urban dynamics.

*Attributes and properties*

The class has two main attributes, *population* and *pop\_probability*, respectively.

* Population defines the number of inhabitants of the block and can be specified deterministically (as a fixed numerical value) or stochastically, through a probabilistic function that dynamically generates the number of inhabitants of the neighbourhood.
* Pop\_Probability makes it possible to further detail the demographic composition of the neighbourhood; it is in fact a probability distribution that governs the assignment of individuals to different types of citizens (e.g. eco-friendly, non-eco, etc.). The probability is defined via a dictionary such as {eco:0.75, neutral:0.25}: This allows each neighbourhood to have a customised social configuration.

The implementation of the class allows the construction of a dynamic and realistic urban population model, where citizens are positioned in the simulated space in a manner consistent with their characteristics, facilitating analysis and simulations of emergent behaviour on an urban scale.

*Methods*

In addition to the 'copy' method, which allows a block to be copied, the main method is the \_\_call\_\_ method, which is used to populate the block with citizens consistent with what is defined in the Population and Pop\_Probability attributes. Specifically, when an instance of Block\_type is invoked, the \_\_call\_\_ method will

* determine the actual number of inhabitants to be created, assigning each of them a random position within the simulated space, using the random\_circle function, which generates a random co-ordinate within a circle of predefined centre and radius. Note that when the block is created, the centre of the circle will coincide with one of the nodes of the network/grid defining the urban space;
* assign each individual a type (Citizen Type), based on the probabilities defined in the Pop\_probability dictionary;
* assign individuals the list of bins in which textile waste can be delivered.

**Note.** Obviously, the same Urban\_Block object may be assigned to several nodes on the city map. For example, the suburbs could all be equivalent nodes with the same number of inhabitants and the same distribution of non-eco (probably the majority) and eco citizens.

**All\_Blocks class**

This class allows the creation of all neighbourhoods in the city, organising the population within the urban network. As input, the initialiser receives, the list of all block types (Block Types) that may be found within the city and the nodes that will define the city. Note that the nodes are only 2-tuples that define the position occupied by the node within the grid that defines the city map. For example, (0,0) is the node located in the lower left vertex of the grid, just as node (n, n) is the node located in the upper right vertex. The important thing to note is that, at this level the grid has not yet been physically created, we know what position each node will occupy, but we do not yet know what distance there will be between one node and another. As we shall see, the 'physical' creation of the lattice (as a network object is delegated to the City class).

Subsequently, through appropriate methods, the class will assign each node (or set of nodes or 'neighbourhoods') a specific type of block and populate it accordingly.

*Attributes and Properties*

The main attributes are the following three

* block\_types, a tuple containing all block types that may be present in the urban network
* all\_blocks, a dictionary associating each node in the network with the corresponding block type. Note that initially the dictionary will be empty, as specific methods will be used to make the node-island type assignment (and subsequent node population). Also note that the network defining the city map is not a property attribute of the All\_Blocks object, but will be an attribute of the City object (the top-level entity) explained below.

*Methods*

The class provides the following 'magic' methods:

* \_\_getitem\_\_(self, node), retrieves the block type associated with the specified node.
* \_\_setitem\_\_(self, node, block\_type), assigns an isolate type present in block\_types to the specified node, inserting a new node: block\_type pair in the all\_blocks dictionary.
* \_\_iter\_\_(self), makes the object iterable: by iterating over an instance of All\_Blocks, a pair (node, block\_type) is obtained at each step.

To avoid having to populate nodes one by one with \_\_setitem\_\_, the class provides some methods for mass assignment:

* all\_equal(self, nodes, block\_type), given a set of nodes and a single block\_type, populates the nodes after assigning each of them the same type.
* all\_random(self, nodes, block\_types), given a set of nodes and a set of possible block\_types, populates the nodes after assigning each a randomly chosen type from those provided.
* all\_random\_with\_probability(self, nodes, weighted\_block\_types), similar to all\_random, but allows a different selection probability to be specified for each block\_type.

Finally, there are three methods for generating a set of nodes arranged according to a precise spatial arrangement (or topology):

* contiguous\_nodes(self, width, height), generates a set of nodes arranged to form a rectangular grid of dimensions width × height.
* make\_frame(self, width, height), creates a 'ring' of nodes corresponding to the perimeter of a rectangular grid of dimensions width × height, i.e. all external nodes.
* make\_multi\_frame(self, width, height, layers) extends the concept of make\_frame by generating concentric rings of nodes around the perimeter of a grid of dimensions width × height.

These sets of nodes can then be passed, for example, to population methods (all\_equal, all\_random etc.) to automatically assign block types.

**Class City**

This is the class that represents the entire city. The city is defined as a grid of equally spaced nodes with a square shape. Each node is populated with citizens and some nodes house garbage collection bins. As mentioned earlier, the City transforms an All\_Blocks object into a real grid by assigning Cartesian coordinates (x, y) to each node.

*Attributes and properties*

The main attributes are

* env, a reference to the simpy environment,
* dist, the distance between each node, used to create the grid,
* n\_node, the number of nodes on each side of the grid,
* map, an object of type All\_Blocks that defines the possible types of blocks that can be assigned to nodes in the grid
* calendar, an object of type Calendar
* bins, a list of Bins objects defining the bins to be placed in the network
* ctzn, a dictionary that associates each node with the citizens associated with it. This dictionary will be populated using one of the populating methods of the All\_Bloks object
* gf, the graph containing all the nodes in the network. This graph is obtained using the networkx library
* dm, the matrix of minimum distances between each pair of nodes.

*Actions performed when the instance is initialised*

When the city is created (method \_\_init\_\_), the following actions are performed:

* first, the grid defining the city is created,
* for reasons of efficiency, a matrix of the minimum distances connecting each pair of nodes is created (the minimum distance is calculated using Dijkstra's graph algorithm).
* each node is populated according to the structure defined in the map attribute (which contains an object of type All\_Blocks)
* Once the citizens have been created and positioned in a specific grid co-ordinate, each of them is assigned: (i) the list of bins, with an indication of their distance, to which they can deliver their textile waste, and (ii) the set of influencers, i.e. the people who are within the radius of influence and who can therefore influence the citizen's behaviour.

*Main methods*

The main methods implemented are as follows.

* \_gen\_network, uses the networkx library to generate an undirected square-shaped graph composed of nxn nodes, each equi-spaced by a distance d from its eight neighbouring nodes.
* \_gen\_dmat, translates the network (city graph) into a distance matrix that associates each pair of nodes with the minimum distance separating them. The distance is obtained via Dijkstra's minimum path optimisation algorithm. The matrix returned is a DMatrix object, defined in the following.
* \_get\_influencers, finds the influencers of each citizen using the radius of influence (rof) and depth (n) defined for each citizen type.
* show, shows the city graph indicating the number of citizens of each node and the position of the bins.

*Other methods*

The class also has the following additional methods

* Iter\_citizens, which returns the citizens of the city one by one,
* n\_citizens, which returns the number of citizens, possibly filtered by node and type.

**Class DMatrix**

Support class that manages a distance matrix. As input, it must receive a dictionary composed as follows:

* the key is a 2-tuple representing two nodes
* the associated value is the minimum distance (on the original graph) to connect the two nodes

As mentioned, this dictionary is calculated by the \_gen\_dmat method of the City object. This dictionary is the only attribute of the class, which also has the following methods:

* \_\_get\_item\_\_, extracts the distance between two nodes from the dictionary,
* All\_nodes, returns a list with all nodes in the network
* As\_DF, turns the dictionary into a DataFrame pandas with the individual nodes used as row and column headers and their minimum distances as values.

**SIM\_OBJECTS MODULE**

The Objects module contains all simulative objects (outside of citizens, which are contained in the People module). The objects in question are:

* Trucks, which model collection vehicles
* the Bins, which model the collection bins,
* the simulation manager.

**Class Truck**

The Truck class models the vehicles used to collect textile waste and take it to the point of separation and disposal. Each truck is associated with a routing that defines the sequence of bins it must visit; this routing is fixed and defined so as to minimise the total distance covered. In practice, the truck starts from a parking area, visits the first bin, empties it, drives to the second and so on. Once the tour is over, the truck drives to the disposal point, unloads the collected waste and finally returns to the staging area. Should the capacity of the lorry not be sufficient to accommodate all the waste in the bins assigned to it, the lorry makes several partial rounds. As soon as it becomes saturated, the lorry drives to the disposal area, unloads the waste and returns to the last bin visited (if not completely emptied) or to the next one and resumes the tour. Note that the bin has an upper threshold level (close to maximum saturation), at which it goes directly to the disposal area without even visiting the next bin (since it would not have enough space to empty it). Similarly, bins have a lower threshold level, the operation of which is as follows: if the truck fails to empty the bin completely, but succeeds in lowering the level below the lower treshold, then the bin is considered 'as empty' and the truck will not return to empty that bin on the next sub-turn.

Attributes

The main attributes are as follows.

* env, the reference to the simpy environment
* map, the reference to the city graph distance matrix
* cap, the capacity in tonnes
* thr, the upper threshold level
* vel, the speed
* tld, a function defining the loading time as a function of the quantity taken (usually a fixed time plus a variable part proportional to the quantity taken)
* tul, a function defining the unloading time at the disposal centre
* bins, a dictionary associating nodes and bins
* f\_route, a list defining the routing to be followed by the truck. This list contains all the nodes where the bins to be unloaded by the truck are located, as well as the input/output node where the staging area is located and the node to which the sorting area is assigned.
* stats, a dictionary which associates at the instant a pick-up mission starts, a Stat object containing all the statistics of the trip (kg collected per bin, loading times, sub-routing performed, etc.).

Methods

The truck has only one main method, the collect\_from\_bins method which, in fact, is the simpy process used to reproduce a pick-up tour which, as mentioned, takes place in the following way: the truck visits all the bins in the order assigned (and defined in its routing) and, if it does not have enough space, it goes to the unloading, and then returns to the bin that has not been completely emptied before heading to the next bin. Once the routing has been completed and the waste has been unloaded (at the sorting station), the truck returns to the station. This process is triggered externally (e.g. at a fixed call time, or in the case of smart bins, when the level of a bin exceeds an upper threshold point).

This process takes advantage of the following sub-processes:

* \_sub\_route which handles any sub-routes. Let us assume that the routing is as follows: [i/o, b1, b2, b3, b4, b5, as, i/o] where i/o is the input output node (staging area), as the sorting area node, b1 the node where the first bin is located, and so on. Suppose that at b3 the truck becomes saturated without the level of b3 falling below the lower threshold. In this case the process is interrupted, and the truck is sent directly to 'as', after which the routing changes as follows: [b3, b4, b5, as, i/o].
* \_trip the single move between two nodes,
* \_load the load process,
* \_unload the unload process

Finally, the truck has a number of properties that allow basic statistics to be calculated, such as the total number of trips made, the average number of sub-tours made on each mission, etc.

**Bin Class**

This class allows bins to be modelled, which may be standard or smart bins.

Attributes

The main attributes are as follows.

* env, the reference to the simpy environment
* crd, the bin's Cartesian coordinate
* lvl, the instantaneous level
* cap, the maximum capacity,
* thr, the upper threshold that triggers the call, in the case of smart bins.
* cap, the capacity in tonnes
* thr, the upper threshold level
* l\_thr, the lower threshold level, below which the bin is considered empty.
* stat, a dictionary with basic statistics (number of times it was completely emptied, number of times it reached full saturation before being emptied, etc.).

Methods

In addition to a number of representation methods, the two main methods (actually simpy processes are as follows:)

* Collect which, if possible, empties the bin. Specifically, this method receives as input a value corresponding to the kilograms to be collected. Typically, this value is the remaining capacity of the truck, the maximum that the truck could take away. The method returns this quantity if present in the bin, or the entire contents if not. Obviously the method updates the level of the bin.
* Put the kg passed as an input argument into the bin, provided that this quantity can be put into the bin (there is sufficient space). If the bin is intelligent, this method sends a signal to the bin manager as soon as the upper threshold is reached, so that the manager starts a pick-up mission as soon as possible.

**Class Sim\_Manager**

The Sim\_Manager class is used to manage and synchronise all simpy processes that make up the simulation.

Attributes

The main attributes are as follows:

* env, a reference to the simpy environment
* ctz, a tuple containing the types of citizens considered
* tr\_bin, a dictionary that associates each truck with the bins that the truck must visit
* node\_pop, a nested dictionary that associates to each node all the citizens assigned to that node, grouped by type
* calendar, a reference to the calendar object
* ctz\_stat a Citizen\_Stats object, which is responsible for generating citizen statistics
* bin\_truck\_stat a BT\_Stats object to generate bin and truck statistics

Methods

The main methods are as follows

* gen\_waste\_single\_citizen, simple process which simply runs the gen\_waste process of each citizen.
* apply\_incentives, a simpy process that applies incentives at specific time instants (e.g. January of the second year), probabilistically increasing the gaw of citizens (of all or only those in specific categories and/or specific nodes/neighbourhoods).
* update\_gaw, a periodic simpy process that updates the gaw of each citizen, depending on the mutual influence exerted between citizens belonging to the same area, according to the operation described above.
* schedule\_truck, a periodic simpy process that serves to assign new pick-up missions to each truck in the fleet. Generally, this process has a timed activation (fixed-interval missions), but in the case of smart bins, the process can be activated on call.

Completing the module are the two classes Citizen\_Stats and BT\_Stats, which collect data (by processing transactions) to generate summary statistics for citizens and bins.

The main methods are as follows:

* avg\_gaw\_trend which generates the historical series of the gaw\_average trend over time, calculated at the level of the individual node and by type of citizen. Obviously, the method also allows aggregation by multiple citizen types and over multiple nodes (at the limit at the level of the entire city).
* tr\_stat, generates the main statistics for each truck: number of trips made, number of call trips, total km, loading and unloading times, average number of sub-tours per mission, average saturation level, etc.
* b\_stat, generates the main statistics for the bins: number of times the bin was emptied, average level at the time of emptying, number of times the bin was filled number of calls made (if smart), etc.
* cts\_stat, generates cumulative citizen statistics calculated per individual node and per citizen type. As before, these can also be aggregated for multiple types and across multiple nodes. The main statistics are:
  + the change in gaw broken down into its determinants: incentives, distance to bin, reciprocal influence effect, service effect
  + the average amount of waste produced,
  + the quantities delivered,
  + those disposed of in unsorted waste,
  + those that the citizen would have liked to dispose of but threw in the unsorted waste because he found the bin full,
  + the level of service
  + the kilometres travelled by car to dispose of by each citizen

Finally, there is a method that converts organises these statistics into a pandas data frame, for ease of reading.

**SEQUENCE OF CREATION OF SIMULATIVE OBJECTS**

The creation of a simulative scenario requires the following steps:

* The simpy environment is created
* Create the necessary citizen types (instances of the Citizen\_Type class), specifying the attributes that distinguish each type.
* You define the blocks you intend to create (instances of the Urban\_Block class), specifying the population of the block (deterministic or stochastic) and the distribution of the types of citizens (among those defined above) that will populate the block. E.g. Very\_Green\_Block = Urban\_Block(population = pop\_nr, type\_prob = {eco:0.75, neutral:0.25}) in this case we have a very "green" block with a population distributed as defined in the function pop\_nr (truncated normal distribution) eh with 75% eco people and 25% standard type (neutral).
* We create the city map (instance of the class All\_Blocks), defining the number of nodes that will make up the city map and assigning each of them (or groups of ess that form neighbourhoods) a block type.

For example:

The\_Map = All\_Blocks((Very\_Green, Green, Neutral, Low\_Green, Non\_Green)) creates a map that will consist of three types of blocks

Extreme\_Periphery = tuple(The\_Map.make\_multi\_frame(mx = mx, layers = 0)) returns the ods of the outer ring that will form the 'extreme periphery' neighbourhood

The\_Map.all\_equal(Extreme\_Periphery, 3) assigns the extreme periphery the type Non-green, index 3 of those passed to the All\_Blocks injector

* Bins are now created and assigned to the nodes
* The city (an instance of the class City) is created by passing it as input the list of bins, the All\_Blocks object and the distance between nodes. In this way, the class creates the spatial grid (placing each node in a Euclidean space and connecting neighbouring nodes) and calculates the corresponding distance matrix (DMatrix object). At this point, since each node has a co-ordinate, the nodes are populated, each citizen is assigned a position (x, y) and the identification of influencers for each citizen can proceed. The City class finishes its functionality by placing bins on the grid.
* Trucks are created by defining their route (routing), i.e. associating them with a set of bins.
* The SIM\_MANAGER object is created, which receives all the previous objects, creates and sets up the objects needed to run the statistics (Citizen\_Stats and BT\_Stats) and launches and orchestrates the simpy processes of the various agents (citizens, trucks and bins).

**EXAMPLE OF LAUNCHING A SIMPY RUN**

An example of the launch of a simulation run considering a city consisting of a square-shaped grid of 18x18 = 324 nodes is shown here. In practice, we therefore have a structure consisting of 9 rings, the outermost of which has 68 nodes, the innermost of which has only 4.

Each node has an average number of inhabitants of 80 for an expected total of 25920 inhabitants. Five neighbourhood typologies are also defined from non-green (high percentage of people with low gaw) to very-green (high percentage of people with high gaw). These typologies are arranged concentrically, according to a gradient from very low sustainability in the periphery to high sustainability in the city centre (inner ring). In particular:

* the outer ring is composed of non-green blocks,
* the next two rings are composed of low-green blocks,
* the next three are composed of neutral type blocks,
* the next two rings are composed of green-type insulators,
* the middle ring is composed of very-green type isolates.

Citizens generate an average of 15 kg of textile waste per year; considering 2 months of 'high seasonality', which increases the conferment of the remaining 10 months by 2.5 times, this corresponds to an average monthly conferment of 1kg. Furthermore, citizens do not deliver if they do not have at least 1.5 kg of fabric to throw away. In case the amount produced in the month is less, they accumulate it waiting to confer it the next time.

The system is sized to handle demand rather easily in the months of low seasonality, but not those of high seasonality. Specifically, 16 bins are arranged in the network, 12 of which are on the third ring (starting from the outside) and 4 on the seventh, each with a capacity of 500 kg. The outer bins are entrusted to a truck with a capacity of 5 tonnes, the inner ones to a smaller truck with a capacity of 2.5 tonnes. Both carry out two pick-up missions (to the bins) each month.

The code used to run the simulation is shown below

# WASTE GENERATION

f\_gen\_waste = f\_waste\_gen(0.75, 1, 1.25) # kg generated in individual months of low seasonality, triangular distribution

s\_factor = 2.5 # seasonality factor

seasonal\_months = (5, 11)

kg\_min = 1.5 # minimum quantity to be conferred

# DISTANCE DISINCENTIVE EFFECT

# with these values the disincentive effect is negligible up to 200 metres, whereas it cancels the willingness to contribute with distances over 2.5 km

f\_dist = f\_dist\_sig(alpha = 1.6, beta = 3.2, max\_ds = 2.5, in\_km = True)

# DIFFERENT DISTRIBUTIONS BY CITIZEN TYPE

# Let's start with incentives

incentive\_times = [t\*365\*24 for t in range(5)] # the possible instants at which an incentive is activated

f\_inc\_low = f\_incentive\_tr(0.1,0.15,0.2) # Incentive has little effect, at most it increases the gaw by 20%.

f\_inc\_std = f\_incentive\_tr(0.2,0.3,0.4)

f\_inc\_high = f\_incentive\_tr(0.3,0.4,0.5)

# setting everything to None no incentive will take effect

INC\_EC = None # are never affected

INC\_NT = None

INC\_NE = None

# EFFECT OF EMPTY BID, apha and beta form factors of sigmoid curves, delta\_inc the expected increase in gaw

f\_EB\_EC = f\_influence\_sig(alpha = 2.5, beta = 5, delta\_inc = 0.015, delta\_dec = None , inf\_type = Influence.Bin\_empty) # this will be the effect on eco citizens

f\_EB\_NT = f\_influence\_sig(alpha = 2.5, beta = 5, delta\_inc = 0.015, delta\_dec = None , inf\_type = Influence.Bin\_empty)

f\_EB\_NE = f\_influence\_sig(alpha = 2.5, beta = 5, delta\_inc = 0.015, delta\_dec = None , inf\_type = Influence.Bin\_empty)

# FULL BIN EFFECT

f\_FB\_EC = f\_influence\_sig(alpha = 2.5, beta = 5, delta\_inc = None, delta\_dec = -0.15 , inf\_type = Influence.Bin\_full)

f\_FB\_NT = f\_influence\_sig(alpha = 2.5, beta = 5, delta\_inc = None, delta\_dec = -0.15, inf\_type = Influence.Bin\_full)

f\_FB\_NE = f\_influence\_sig(alpha = 2.5, beta = 5, delta\_inc = None, delta\_dec = -0.15 , inf\_type = Influence.Bin\_full)

# WORD OF MOUTH - NEIGHBOURHOOD

roi = (0.25, 0.35) # radius of influence3

n\_rings = 2 # neighbourhood rings

# NEIGHBOURHOOD EFFECT

f\_PP\_EC = f\_influence\_sig(alpha = 2.5, beta = 5, delta\_inc = 0.2, delta\_dec = -0.05, inf\_type = Influence.People)

f\_PP\_NT = f\_influence\_sig(alpha = 2.5, beta = 5, delta\_inc = 0.15, delta\_dec = -0.3, inf\_type = Influence.People)

f\_PP\_NE = f\_influence\_sig(alpha = 2.5, beta = 5, delta\_inc = 0.1, delta\_dec = -0.15, inf\_type = Influence.People)

# CREATION OF THREE CITIZEN TYPES

eco = Citizen\_type(kind = 'eco', s\_factor = s\_factor, m\_season = seasonal\_months, min\_waste = kg\_min,

green\_awareness = (0.7, 0.9), radius\_of\_influence = roi, n\_depth = n\_rings,

gen\_waste = f\_gen\_waste,

green\_inc = f\_EB\_EC,

green\_dec = f\_FB\_EC,

f\_influence = f\_PP\_EC,

f\_incentives = INC\_EC,

f\_dist\_penalty = f\_dist,

f\_move = f\_mv)

neutral = Citizen\_type(kind = 'neutral', s\_factor = s\_factor, m\_season = seasonal\_months, min\_waste = kg\_min,

green\_awareness = (0.4, 0.6), radius\_of\_influence = roi, n\_depth = n\_rings,

gen\_waste = f\_gen\_waste,

green\_inc = f\_EB\_NT,

green\_dec = f\_FB\_NT,

f\_influence = f\_PP\_NT,

f\_incentives = INC\_NT,

f\_dist\_penalty = f\_dist,

f\_move = f\_mv)

non\_eco = Citizen\_type(kind ='non\_eco', s\_factor = s\_factor, m\_season = seasonal\_months, min\_waste = kg\_min,

green\_awareness = (0.15, 0.35), radius\_of\_influence = roi, n\_depth = n\_rings,

gen\_waste = f\_gen\_waste,

green\_inc = f\_EB\_NE,

green\_dec = f\_FB\_NE,

f\_influence = f\_PP\_NE,

f\_incentives = INC\_NE,

f\_dist\_penalty = f\_dist,

f\_move = f\_mv)

# CREATION OF ISOLATES

pop\_nr = normal\_pop(mu = 80, sigma = 8, min\_pop = 50) # population per isolate

# 5 TYPES OF BLOCKS

Very\_Green = Neighbourhood(population = pop\_nr, type\_prob = {eco:0.75, neutral:0.25})

Green = Neighbourhood(population = pop\_nr, type\_prob = {eco:0.375, neutral:0.5, non\_eco:0.125})

Neutral = Neighbourhood(population = pop\_nr, type\_prob = {eco:0.25, neutral:0.5, non\_eco:0.25})

Low\_Green = Neighbourhood(population = pop\_nr, type\_prob = {eco:0.125, neutral:0.5, non\_eco:0.375})

Non\_Green = Neighbourhood(population = pop\_nr, type\_prob = {neutral:0.25, non\_eco:0.75})

# THE CITY MAP DIVIDED BY RINGS

n\_nodes = 18 # maximum number of nodes per side

node\_distance = 0.15

mx = n\_nodes - 1

The\_Map = All\_Neighbourhood((Very\_Green, Green, Neutral, Low\_Green, Non\_Green))

# We create the sets of nodes defining the rings, which we will use to create the population

Extreme\_Periphery = tuple(The\_Map.make\_multi\_frame(mx = mx, layers = 0)) # extreme (first) ring

Periphery = tuple(The\_Map.make\_multi\_frame(mx = mx, layers = (1, 2)) # second, third

Zone\_Residential = tuple(The\_Map.make\_multi\_frame(mx = mx, layers = (3, 4, 5)) # three centre rings

Centre = tuple(The\_Map.make\_multi\_frame(mx = mx, layers = (6, 7)) # third and penultimate ring

Centre\_historical = tuple(The\_Map.make\_multi\_frame(mx = mx, layers = 8))# first ring (centre ring)

The\_Map.all\_equal(Extreme\_Periphery, 4) # type Non\_Green

The\_Map.all\_equal(Periphery, 3) # type Low\_Green

The\_Map.all\_equal(Zone\_Residential, 2) # type Neutral

The\_Map.all\_equal(Centre, 1) # type Green

The\_Map.all\_equal(Centre\_Historical, 0) # type Very\_Green

# BINS AND COLLECTION MEANS

b\_cap = 500 #kg

b\_tr = None # call treshold null, therefore fixed time

b\_ltr = 50 # kg low treshold, that for which the bin is considered empty

b\_nr = 16

# The coordinates where we will place the outer and inner bins

b\_coord\_est = ((2,2), (2,6), (2,11), (2,15), (6,15), (11,15)

(15, 15), (15, 11), (15, 6), (15, 2), (11, 2), (6,2))

b\_coord\_int = ((6,6), (6,11), (11,11), (11,6))

b\_all\_coord = b\_coord\_est + b\_coord\_int

# input and ouput points

start\_point = (0, 0)

end\_point = (6, 0)

# van going to the outer bins

big\_van\_cap = 5000 #kg

big\_van\_tr = 4800 #kg

speed = 25 #km/h

tc\_bidone = lambda kg: 0.02 + 0.00016\*kg # bin loading time, fixed + proportional to kg, if bin full 6 minutes

tsc\_island = lambda kg: 1 # fixed time of one hour

# van going to internal bins

sm\_van\_cap = 2500 #kg

sm\_van\_tr = 2300 #kg

turnaround time = 14\*24 # 14 days, 2 weeks

# SIMULATION: TIMES IN HOURS, DISTANCES IN KM, QUANTITIES IN KG

seeds = (3, 5, 7, 9, 11) # the seeds that will be used

a\_day = 24 # 24 hours

days\_year = 365

duration\_in\_years = 4 # 4 years

days\_month = (31, 28, 31, 30, 31, 30, 31, 30, 31)

inter\_time\_gaw = 30\*one\_day # average inter\_time of conferment

intertempo\_agg\_gaw = tuple(g\*one\_day for g in days\_month) # you can also pass a number

distance\_matrix = None # this does not change, so we keep it good

for sd in seeds: # the seeds that will be used

s = f'Start simulation with seed {sd}'

print()

print('\*'\*30)

print(s)

print()

random.seed(sd)

env = sp.Environment()

calendar = Calendar(env) # the calendar

# CREATION OF BINS

Std\_bin = Bin(env = env, capacity = b\_cap, threshold = b\_tr, low\_treshold = b\_ltr, calendar = calendar, full\_percent = 0.95)

bins = Std\_bin.gen\_n\_at(b\_all\_coord)

# CITY AND DISTANCE MATRIX

the\_City = City(env = env, n\_nodes = n\_nodes\*\*2, distance = distance\_nodes,

the\_map = The\_Map, bins = bins, in\_out = (start\_point,end\_point),

calendar = calendar)

if not distance\_matrix: # create the distance matrix, only the first time

print('Creating distance matrix, will only be done once')

distance\_matrix = The\_City.gen\_dmat(show\_increase = 0.1)

print('End of creation of distance matrix')

# TRUCK CREATION

trucks = [

Truck(env = env, the\_map = distance\_matrix, capacity = big\_van\_cap, threshold = big\_van\_tr,

velocity = speed, n\_in = start\_point, n\_out = end\_point,

bins = tuple(b for b in bins if b.nd in b\_coord\_est),

t\_load = tc\_bidone, t\_unload = tsc\_island),

truck(env = env, the\_map = distance\_matrix, capacity = sm\_van\_cap, threshold = sm\_van\_tr,

velocity = speed, n\_in = start\_point, n\_out = end\_point,

bins = tuple(b for b in bins if b.nd in b\_coord\_int),

t\_load = tc\_bid, t\_unload = tsc\_island),

]

Mg = Manager(env = env, the\_city = The\_City, trucks = trucks, ctz\_types = ('eco', 'neutral', 'non\_eco'), all\_ctz = 'all', calendar = calendar)

# The processes it handles

env.process(Mg.calendar.make\_calendar(show = True))

Mg.gen\_waste\_single\_citizen(mean\_time = inter\_time, rnd\_bin = True)

env.process(Mg.update\_green\_awareness(dt =intertempo\_agg\_gaw , min\_dist = 0.05, wgh = True))

env.process(Mg.apply\_incentives(incentive\_times[1:]))

for tr in Mg.tr\_bin.keys():

env.process(Mg.schedule\_trucks(tr, dt = intertempo\_turn ))

print("Effective start of simulation")

env.run(until = a\_day\*days\_year\*duration\_in\_years + 5)

print()

s = f'End of simulation with seed {sd}'

print(s)