SamPy: A New Python Library for Stochastic Spatial Agent-Based Modeling in Epidemiology of Infectious Diseases

François Viard[1] Emily Acheson[1,2] Agathe Allibert[1]
Caroline Sauvé[1,3] Patrick Leighton[1]

[1]Groupe de recherche en épidémiologie des zoonoses et santé publique (GREZOSP), Faculté de médecine vétérinaire, université de Montréal

[2]Public Health Risk Sciences Division, National Microbiology Laboratory, Public Health Agency of Canada

[3]Pelagic and Ecosystem Science Branch, Fisheries and Oceans Canada, Mont-Joli, Québec, Canada

November 4, 2022

Abstract

Agent-based models (ABMs) are computational models for simulating the actions and interactions of autonomous agents in time and space. These models allow users to simulate the complex interactions between individual agents and the landscapes they inhabit and are increasingly used in epidemiology to understand complex phenomena and make predictions. However, as the complexity of the simulated systems increases, notably when disease control interventions are considered, model flexibility and processing speed can become limiting. Here we introduce SamPy, an open-source Python library for stochastic agent-based modeling of epidemics. SamPy is a modular toolkit for model development, providing adaptable modules that capture host movement, disease dynamics, and disease control interventions. Memory optimization and design provide high computational efficiency allowing modelling of large, spatially-explicit populations of agents over extensive geographical areas. In this article, we demonstrate the high flexibility and processing speed of this new library. The version of SamPy considered in this paper is available at https://github.com/sampy-project/sampy-paper .

Key words: Agent-based-model; epidemiology; python; zoonotic diseases

1 Introduction

The last two years have demonstrated how emerging infectious diseases can have profound impacts on our society, and how mathematical models play a crucial role in predicting the spread of infectious agents, as well as the efficiency of alternative control interventions ([18]). Given that 60% of emerging diseases have a zoonotic origin, and 70% of those come from wild fauna ([9]), it is of fundamental importance to understand

 doi:10.20944/preprints202211.0556.v2

the mechanisms involved in disease transmission within domestic animals and wildlife populations.

In epidemiology, the demographic and behavioural characteristics of host populations, coupled with resource availability and geographic characteristics, drive the dynamics of epidemics as well as the design of control measures ([19]). Thus, for mathematical models to provide a method to simulate disease dynamics, predict the outcomes of alternative control interventions, and allow for optimization of intervention efforts, they have to adequately account for host movement and behaviour parameters, mimic pathogen characteristics and be representative of landscape characteristics and barriers.

The mathematical models most commonly used for the study of epidemics are equation-based models, such as compartmental models [3], or simulation models, such as agent-based models (ABMs) [6]. Equation-based models are computationally less intensive than simulation models but assume homogeneous mixing of agents (e.g., individuals, social groups, or pathogens), where each agent has an equal probability of interacting with other agents. By contrast, agent-based models are computationally intensive but simulate individual agents independently, allowing each to have its own set of characteristics, such as age or immunization status [6]. As a result, each agent can behave uniquely, with consequences for its probability to interact with other agents and, thus, its susceptibility to contract and spread disease. In addition, ABMs can account for the spatial heterogeneity of populations and landscapes ([26]).

As the complexity of our models of disease systems increases, such as with the inclusion of detailed geographic information systems (GIS) data [11] or multiple interacting species, computing power and processing time may become factors limiting our ability to make optimal use of ABMs [6]. Therefore, choosing the right software to develop an efficient ABM is critical for the model success. Many frameworks aimed at developing ABMs are available, like **MASON** [15] and **NetLogo** [30] or the **Python** library **Mesa** [10]. Those frameworks are highly versatile, and have been successfully used to develop ABMs in a wide variety of fields (*e.g.*, Social Science [1], Economy [22], Ecology [17], etc...). These well-known frameworks can have performance issues when the number of agents becomes very large. It is almost impossible to give a precise definition of "very large" since it varies greatly depending on the framework and the ABM itself (even papers comparing frameworks performances such as [13] have to use qualitative terminology to describe ABM scalability). However, when studying epidemics, we need the capacity to modelize populations of million of agents, and this is generally considered too large for the frameworks mentioned above [16]. Other solutions exists, such that framework allowing the development of ABMs for supercomputers or Graphics processing units (GPUs) (*e.g.* [12] or [31]) but these come with their own difficulties, requiring the user to have access to specialized (and expensive) hardware, and such tools are also generally more complex to use than usual frameworks.

In this paper, we introduce an open-source **Python** Library called SamPy (Stochastic Agent-based Modeling with Python). We use **Python** ([25]) as the underlying computing language due to its high readability, flexibility, and broad variety of available scientific libraries such as **NumPy** [5] and **SciPy** [27]. **NumPy** is a performance-focused library

 doi:10.20944/preprints202211.0556.v2

in Python, providing a multi-dimensional array (ndarray) object that is key to efficient computing. We optimize performance using the Numba library [14], which allows compiling functions on the **NumPy** arrays and achieves **C**-like speed. **SciPy** is built on top of **NumPy** by providing more tools for array computing as well as fundamental algorithms for a wide variety of statistical problems [27].

The optimized processing speed coupled with the modular **Python** toolkit structure offers a programmer-friendly, adaptable agent-based modelling method to capture host movement, disease dynamics, and disease control interventions. SamPy is intended for use by researchers or human and animal health professionals wishing to assess the effect of various control strategies (*e.g.*, vaccination programs targeting wildlife, population reduction efforts, mass sterilization) or the influence of particular drivers (*e.g.*, climate change, habitat loss, landscape composition and configuration, animal behaviour) on infectious disease dynamics. SamPy's source code is open and errors can be reported for repair, thus offering a high level of code verification.

In the following section, we provide a high-level overview of the structure and functions of SamPy, along with installation instructions. In Section 2, we give a more detailed overview of SamPy's capabilities. In Section 3, we demonstrate the application of SamPy through a variety of examples. Finally, in Section 4, we conclude and suggest future directions.

1.1 Structure and functions

SamPy is designed for users without advanced programming skills or in-depth knowledge of the internals of the library. To achieve this, we took inspiration from **NumPy** and **SciPy**. **NumPy** and **SciPy** are at the core of the **Python** ecosystem of scientific libraries. They provide a wide range of methods and functions allowing users to easily create and manipulate arrays, and they are designed such that complex and efficient computations can be performed with little coding. For instance, a user can draw millions of samples from various probability distributions using one line of code, without having to worry about pseudo random number generation (which is still an active field of research, see for instance [8]) or how to efficiently sample from a given distribution. **NumPy** and **SciPy** are designed to perform at maximum efficiency on vectorized algorithms. Vectorized algorithms are algorithms centered on arrays instead of independent values, where most manipulations can be done on all the elements of the arrays in parallel. This constraint may appear very restrictive, but it turns out many problems can be tackled in a vectorized fashion. Indeed, users of **NumPy**/**SciPy** can often vectorize part of the program, use the two libraries to treat those sections and develop adapted solutions for the rest. This approach reduces development time and allows users to take advantage of existing already optimized code. Furthermore, **NumPy** and **SciPy** are well tested (see https://NumPy.org/doc/stable/reference/testing.html), and their widespread use means that most bugs are quickly reported and resolved.

We now draw a parallel between these two libraries and SamPy. The concept of arrays, which are structured collections of variables, can be likened to populations in SamPy, which are structured collections of agents. When functions or methods are ap-

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 December 2022 doi:10.20944/preprints202211.0556.v2

plied on a population object, it generally results in some process being applied to all (or a specified selection) of agents in the population (note that the process is generally not applied in parallel on all the agents, but rather applied to the agents in sequence). Though it is possible to work with individual agents in SamPy, users are expected to work mainly with populations objects. A consequence of this is that SamPy is designed to efficiently create ABMs where agents' actions are performed one after the other at the population level. This constraint can be seen as the SamPy equivalent to the Vectorization constraint in **NumPy** and/or **SciPy**. This is best illustrated using a simple example of an ABM constructed with SamPy.

Let's consider an ABM consisting of a population of a theoretical species living on an artificial landscape. We simulate the spread of an infectious disease within the population over ten years, using 520 weekly timesteps. When the population is loaded (first week of our simulation), it contains approximately 110 000 agents. The source code for our example can be found in ex1_paper.py on GitHub. We begin by focusing on the first lines of the file, which contains the imports and the initial setup of the model.

```
from sampy.agent.builtin_agent import BasicMammal
from sampy.graph.builtin_graph import SquareGridWithDiag
from sampy.disease.single_species.builtin_disease import
ContactCustomProbTransitionPermanentImmunity
from constant_paper import (ARR_WEEKLY_MORTALITY,
                            ARR_NB_WEEK_INF,
                            ARR_PROB_WEEK_INF)
import numpy as np

my_graph = SquareGridWithDiag(shape=(100, 100))
my_graph.create_vertex_attribute('K', 10.)

agents = BasicMammal(graph=my_graph)
agents.load_population_from_csv('path_to_pop_csv')

disease = ContactCustomProbTransitionPermanentImmunity(host=agents,
                                        disease_name='disease')
arr_new_contamination = disease.contaminate_vertices([(50, 50), (50, 51)],.5)
disease.initialize_counters_of_newly_infected(arr_new_contamination,
                                              ARR_NB_WEEK_INF,
                                              ARR_PROB_WEEK_INF)
```

This example is constructed using basic built-in objects from SamPy. In SamPy, agents exist on graphs representing landscapes where vertices are spatial sub-units of the landscape (*e.g.* habitat patches). Therefore, we first instantiate a Graph object using the class SquareGridWithDiag. In this case, the landscape is a 2-dimentional square grid of shape 100 x 100. Next, we add an attribute to the graph's vertices representing the

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 December 2022 doi:10.20944/preprints202211.0556.v2

'carrying capacity' (K) of each spatial unit, which represents the average number of agents a given spatial unit can sustain. In this example, we set $K = 10$ for each vertex. We then create the population object, and we populate it with agents using an external source (CSV text file). This 'Population CSV' has been generated using the script ex1_build_up_paper.py, which is similar to this script but without disease and with a few manually created pairs of agents placed on selected vertices of the landscape. Finally, we create a disease object and infect half of the population of agents on the vertices of coordinates (50, 50) and (50, 51). Agents have four possible states with respect to this disease: 1) susceptible (i.e., not yet exposed to the disease), 2) infected, 3) contagious, and 4) immune.

We now move on to the main loop of our model, where the actions are undertaken in sequence at the population level.

```
list_inf_pic = []
nb_year_simu = 10
for i in range(nb_year_simu * 52 + 1): # main loop
    list_inf_pic.append(my_graph.convert_1d_array_to_2d_array(
disease.count_nb_status_per_vertex('inf')))
    agents.tick()
    my_graph.tick()
    disease.tick()

    agents.kill_too_old(52 * 6 - 1)
    agents.natural_death_orm_methodology(ARR_WEEKLY_MORTALITY,
                                         ARR_WEEKLY_MORTALITY)
    agents.kill_children_whose_mother_is_dead(11)

    agents.mov_around_territory(0.5,
condition=agents.df_population['age'] >= 11)

    arr_new_infected = disease.contact_contagion(0.1,  return_arr_new_infected=True)
    disease.initialize_counters_of_newly_infected(arr_new_infected,
                                                  ARR_NB_WEEK_INF,
                                                  ARR_PROB_WEEK_INF)
    disease.transition_between_states('con', 'death', proba_death=0.8)
    disease.transition_between_states('con', 'imm')
    disease.transition_between_states('inf', 'con',
                                      arr_nb_timestep=np.array([1, 2]),
                                arr_prob_nb_timestep=np.array([.5, .5]))

    if i % 52 == 15:
        agents.find_random_mate_on_position(1., position_attribute='territory')
    if i % 52 == 22:
        agents.create_offsprings_custom_prob(np.array([4, 5, 6, 7, 8, 9]),
```

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 December 2022 doi:10.20944/preprints202211.0556.v2

```
                                    np.array([0.1, 0.2, 0.2, 0.2, 0.2, 0.1]))
if i % 52 == 40:
    can_move = agents.df_population['age'] > 11
    agents.dispersion_with_varying_nb_of_steps(np.array([1, 2, 3, 4]),
                                    np.array([.25, .25, .25, .25]),
                                    condition=can_move)
```

Each iteration corresponds to a week of simulation. We begin each week by creating a 2D array giving the number of infected agents per vertex, and then we store this array in the list LIST_INF_PIC. Then we call the tick methods of the objects used in the simulation. In SamPy, the purpose of a tick method is to perform various internal updates needed at each new timestep (for instance, updating the age of each agent), and most of SamPy's objects come with a tick method. The next three method calls deal with natural mortality in our model. The first method call eliminates all agents older than a user-defined threshold (here six years old). The second method call eliminates some agents according to the methodology found in ORM [24] (see the method code for detailed description). The third method call eliminates the dependant juveniles (if any) whose mother is dead. The following method call, MOV_AROUND_TERRITORY, is quite specific to the currently available built-in agents in SamPy. Here, agents represent territorial mammals, and in SamPy they exists in two distinct locations, one called "territory" and the second called "position". The first location is the vertex on which the animal territory is based (depending on the modeled species, this could for instance be the vertex where the agent has its burrow/nest). The second location represents the vertex in which the animal spent most of the current timestep (for instance while foraging). The MOV_AROUND_TERRITORY call updates the 'position' of the agents while keeping their 'territory' the same, and the position of the agents will either be their territory vertex, or one of the neighboring vertices. Note that, in our current example, the only method that uses 'position' instead of 'territory' as default is CONTACT_CONTAGION that propagates the disease by direct contact among agents.

The next five method calls deal with the disease. The first method call transmits the disease to new susceptible agents, and the transmission is done by direct contact. That is, agents sharing the same vertex at the current timestep of the simulation have a fixed probability of transmitting disease to each other, given that one is susceptible and the other is contagious. The other method calls address transition between disease states. Finally, we have a series of yearly events, corresponding respectively to mating (week 15), giving birth (week 22), and dispersal (week 40). Figure 1 shows maps of the spatial distribution of infected agents at various stages of the simulation.

As shown in the example above, agents' actions are performed in sequence at the population level. That is, each method call will loop through the (generally shuffled) list of all agents and attempt to perform the action encoded by the method. The execution of each action is asynchronous uniform when possible (following the terminology of [4]), in the sense that while looping through the population, each agent has a chance to perform the action and update any variable shared between the agents. For instance, the method

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 December 2022 doi:10.20944/preprints202211.0556.v2

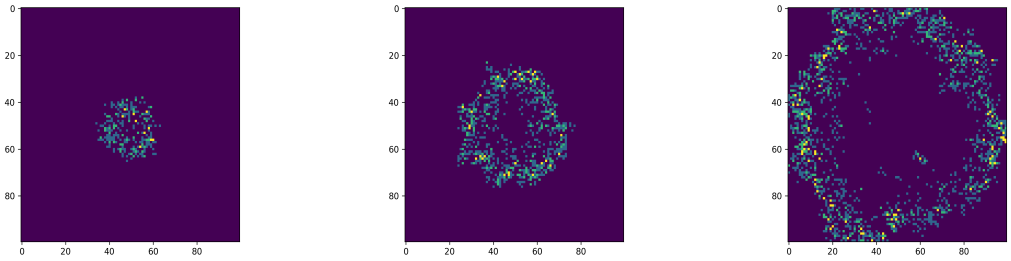

Figure 1: Left to right: map of infected agents after 52, 104 and 208 simulated weeks, respectively, for a simulated population of approximately 100 000 agents of a theoretical species over a synthetic landscape of 100x100 vertices.

NATURAL_DEATH_ORM_METHODOLOGY uses the number of agents per vertex in order to compute a probability of death by natural causes for each agent, and the number of agent per vertex is continuously updated during the method execution as agents are removed. This contrasts with most other ABM frameworks where a user would be able to have actions performed in sequence at the agent level.

A consequence of SamPy's design is that ABMs created using SamPy may be more sensitive to the order in which actions are performed compared to other frameworks. That is, the chosen ordering of the actions may introduce stronger biases. Therefore, working with population objects creates similar constraints as the ones created by arrays in **NumPy** and **SciPy**. When deciding whether SamPy meets the modelling needs of a user, the user should do the following:

- First, run a series of small scale simulations using SamPy;
- If SamPy provides viable models, then one should decide on the order of the actions;
- Finally, depending on the model application's requirements, some code may have to be developed specifically for it. For instance, a user might decide that having to choose the order of execution between the natural mortality and the mortality induced by the disease creates too many biases in the ABM. If so, the user may choose to develop his or her own custom method to deal with both sources of mortality simultaneously.

Similar to **NumPy** and **SciPy**, SamPy provides users with methods that are optimized and tested, and whose use generally results in reasonably readable scripts. Moreover, as discussed in Section 2.1, SamPy has been designed to be modular, and it is reasonably straightforward for a **Python** developer to create their own SamPy components, which can then be easily tested and shared with other users.

SamPy is designed to be purely mono-threaded to allow for the execution of many simulations at once, facilitating large-scale sensitivity or calibration analyses. Therefore,

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 December 2022 doi:10.20944/preprints202211.0556.v2

a typical user would generally be interested in running many simulations efficiently rather than a few at increased speed. Restricting SamPy to mono-threaded computations greatly simplifies the use of multiprocessing, since each simulation can be considered independent from the others.

1.2 Installation instructions

SamPy can be found in the GitHub repository https://github.com/sampy-project/sampy-paper, and is designed to work with the BASE environment of an anaconda3 distribution. If one wants to use his or her own custom environment, here is the list of the dependencies required to use SamPy.

- **Python** Programming language (version 3.8 and higher).
- **NumPy** ($\geq$1.22).
- **SciPy** ($\geq$1.8).
- **Pandas** ($\geq$1.4.3).
- **Numba** ($\geq$0.55.2).

2 Detailed Overview of SamPy's design features

In this section, we detail the design of SamPy.

2.1 Multi-inheritance and modularity

Every aspect of an ABM is subject to many variations, and it would be difficult in practice to recode from scratch each possible variation. Moreover, as explained in Section 1, one of our aims while developing SamPy is to allow users to focus on developing the methods and functions required by their modelling needs. Therefore, we chose to exploit the fact that **Python** allows multiple inheritance to make most objects used in SamPy's scripts composite. To clearly show why this leads to an increased modularity, let us focus on an example from the SamPy source code (at sampy/agent/builtin_agent.py).

```
from .base import BaseAgingAgent
from .mortality import (NaturalMortalityOrmMethodology,
OffspringDependantOnParents)
from .reproduction import (FindMateMonogamous, FindMatePolygamous,
   OffspringCreationWithCustomProb)
from .movement import TerritorialMovementWithoutResistance
from ..utils.decorators import sampy_class


@sampy_class
```

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 December 2022 doi:10.20944/preprints202211.0556.v2

```
class BasicMammal(BaseAgingAgent,
                  NaturalMortalityOrmMethodology,
                  OffspringDependantOnParents,
                  FindMateMonogamous,
                  OffspringCreationWithCustomProb,
                  TerritorialMovementWithoutResistance):
    """
    Agent that represents a basic territorial mammal. This agent is Monogamous.
    """
    def __init__(self, **kwargs):
        pass
```

This is the complete definition of an agent class, and it is entirely constructed using 'class building blocks' gathered from SamPy source code. In many respects, those building blocks are similar to mixins [2], as they are not designed to be instantiated but only to be inherited and give new capabilities to the class inheriting them. The only real difference with mixins lies in the fact that those building blocks come with an INIT method, since they generally require some modification of the child class datastructures.

This organization based on multiple inheritance allows for easy modularity, as one simply needs to develop a new building block to satisfy a specific modeling need. This building block can then be shared, reviewed and tested independently. However, reliance on multiple inheritance generates two problems that we address using the decorator SAMPY_CLASS. First, since each building block has an init method, they need to be called in a specific order. In addition, there may be incompatibilities between building blocks.

Rather than letting the user deal with these difficulties, the decorator SAMPY_CLASS modifies the class by changing the order in which the init methods of the building blocks are called. Note that at the current stage of SamPy's development, there are no incompatibilities between blocks, but this will likely change in the future as new building blocks are developped. Under such circumstances, the decorator will be modified accordingly to raise a meaningful error in case of incompatibility.

By design, many combinations of building blocks are possible. It is therefore difficult to allow the init method of each building block to accept positional arguments. Hence, passing arguments to the constructor should be done using key-word arguments. Furthermore, in order to simplify how the decorator SAMPY_CLASS works, each building block and each SamPy class have to define an init method accepting **kwargs, even if it is not used (as shown above).

2.2 Memory Layout and DataFrameXS

As explained in Section 1.1, SamPy is designed to create ABMs where agents' actions are performed sequentially, at the population level. Generally, a given action does not need to access all of the agents' characteristics. For instance, a method encoding some aspect of agents' migration may only need access to agents' position, and not their age,

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 December 2022 doi:10.20944/preprints202211.0556.v2

gender, infectious status, etc. Therefore, in order to optimize memory access for each method, we chose to store agents' attributes in contiguous one-dimensional arrays. For example, let us consider a population of agents with only two attributes: an age, which is an integer, and a position, which is a couple (x, y) of floats situating the agent on a 2D plane. The attributes of all the agents are stored in the population object as three one-dimensional NumPy arrays, one of integers for the age and two of floats for the position.

For ease of use, attribute arrays are stored within the population object in a custom kind of dataframe, called DataFrameXS (see sampy/pandas_xs/pandas_xs.py). To some degree, our dataframes are designed to feel like **pandas** dataframes [29]. That is, one can set and get columns in the typical way, and it also supports boolean and integer indexing. DataFrameXS also comes with a few useful built-in methods allowing, for instance, to concatenate two dataframes and to shuffle its rows. However, there are several major differences between **pandas**' dataframes and DataFrameXS. First, DataFrameXS columns are always stored as one dimensional **NumPy** arrays, which is generally not the case in **pandas**, where columns of the same type are rearranged in blocks. In addition, the range of **NumPy** dtype accepted by a DataFrameXS is smaller than that accepted by **pandas**. Namely, it is restricted to the types that **Numba** can handle in an optimized manner. The list of allowed types can be found in the DataFrameXS definition (see sampy/pandas_xs/pandas_xs.py):

```
class DataFrameXS:
    """
    [...]
    """
    LIST_ALLOWED_TYPE = ['bool', 'uint8', 'int8', 'uint16', 'int16',
                         'uint32', 'int32', 'uint64', 'int64', 'float32',
                         'float64']
```

Finally, when retrieving a column, the user gets a reference to the underlying **NumPy** array. This allows the user to directly feed a **Numba** compiled function with columns of the dataframe (which is not possible with a **pandas** dataframe, as one gets **pandas** series), and to directly modify the columns within the function.

2.3 Tests and debug mode

The way SamPy is organised abstracts away from the user most of the technicalities of an ABM. This organisation has allowed us to optimize the computations, and to extensively test the components of SamPy. All the tests accompanying SamPy, written using the library **unittest**, can be found in the Github test folder. Given the stochastic nature of the underlying methods, systematic testing of all SamPy's components is a complex task. However, tests are among the most important part of any large-scale scientific libraries, as they significantly reduce the risk of errors. One could argue that they are as important, if not more important, than performances. Consequently, we chose a testing approach

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 December 2022 doi:10.20944/preprints202211.0556.v2

that provided a compromise between performance and ease of testing. Most of Sampy's computations are performed within functions compiled with **Numba**. Proper testing of these functions greatly reduces the risk of computation errors within SamPy. As such, each function of this type is tested at least once in the tests folder available on the Github. We facilitated the tests of those functions by making them purely deterministic. That is, all random number generation is done outside of **Numba** compiled functions, ensuring that all functions can be easily tested without requiring complicated statistical methods. This testing design could have consequences on performances due to additional back and forth between **Python** and '**Numba** compiled' code.

It is to be noted that by default, SamPy's methods perform little to no verification on user inputs. As most ABMs constructed using SamPy will include a 'main-loop' (see the example in Section 1.1) where the same methods are called repeatedly using the same parameters, systematic checking of user inputs would induce a considerable loss of performance. Consequently, input checking has to be purposely activated by the user using the debug_mode which can be activated using the function USE_DEBUG_MODE. An example of its use can be found in the script ex1_paper_with_debug_mode.py on lines 11 to 14. The debug mode is designed to be used when developing a new ABM or when modifying an existing one, and then be disabled when running the simulations of interest.

2.4 Graphs

The integration of the spatial dimension into SamPy's models is currently based on a highly versatile "graph-oriented" approach, although Sampy's design flexibility leaves options for further developments.

A (directed) graph G is mathematically defined as a pair of two sets V and E, V being a finite set called the *vertices* and E a set of pairs of elements of V called the *edges*. Graphs are treated as directed (and stored in memory as such), but whenever an edge

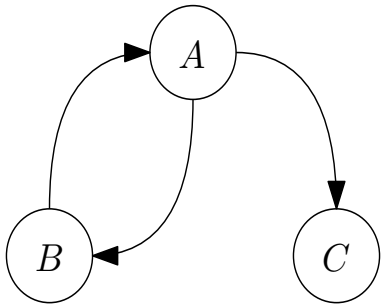

Figure 2: Example with $V = \{A, B, C\}$ and $E = \{(A, B), (B, A), (A, C)\}$.

from a vertex X to a vertex Y exists, then the reverse edge from Y to X exists as well. Therefore, in the following figures, any edge between two vertices is represented by a single solid line without any arrow. In SamPy, graphs are used to represent landscapes, where vertices represent spatial units in which the agents live. In Figure 3, we show

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 December 2022 doi:10.20944/preprints202211.0556.v2

three examples of graphs that can be used to represent two-dimensional landscapes. In

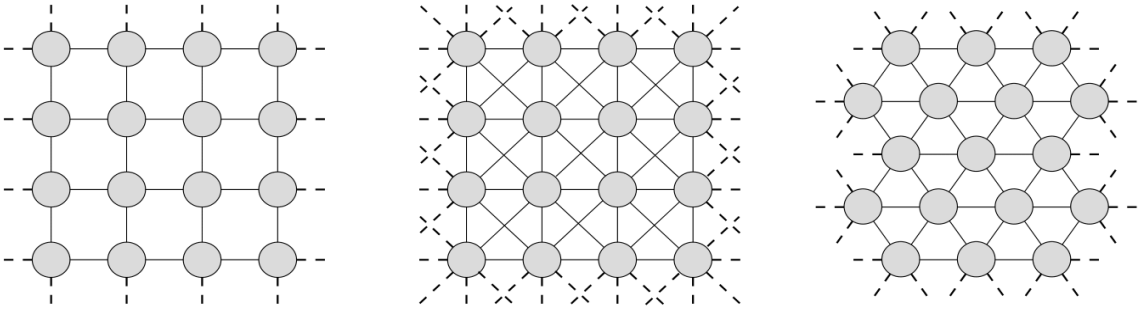

Figure 3: Three examples of "landscape graphs".

SamPy, we assume that the graphs used to represent landscapes satisfy the following properties. There exists an integer N such that:

- any vertex of the graph has degree at most N (here, the degree of a vertex is the number of edges going out of the vertex);
- most of the vertices have degree N.

Such graphs are encoded in SamPy as instanciated objects as follows. Let $G = (V, E)$ be a graph. Each vertex v of G is designated by two different names: an *id*, which is a string and is generally provided by the user, and an *index*, which is an integer generally automatically assigned by SamPy. Indexes and ids are unique, i.e. vertices cannot share the same id and/or index. Finally, if G has k vertices, then the indexes range between 0 and $k-1$. Therefore, given the uniqueness, each value between 0 and $k-1$ is the index of a vertex.

Regarding edge encoding, let N denote the maximum degree of the vertices of G and k the number of vertices of G. The graph object has an attribute called CONNECTIONS, which is a two-dimensional **NumPy** array of integers of shape (k, N). The edges starting from the vertex of index i are encoded in CONNECTIONS[I], which is an integer valued one-dimensional **NumPy** array. Here is an example

```
>>> graph.connections[i]
array([-1, 4, 5, 0, -1, 18])
```

Each non-negative value j in CONNECTIONS[I] is the index of a vertex such that there is an edge from i to j. All negative values are default values which can be ignored. Therefore, in the above example the vertex i is connected to four other vertices, namely 4, 5, 0 and 18.

Closely related to the connections attribute is the attribute WEIGHTS, which associates a probability to each edge of the graph. This may be useful if the user wants to use a landscape graph where the distance between two vertices can vary significantly. For instance, in the center pannel of Figure 3, the diagonal connections are longer than

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 December 2022 doi:10.20944/preprints202211.0556.v2

the horizontal and vertical connections. Hence, an agent moving on such a graph should be less likely to move diagonaly than horizontally/vertically, and the attribute WEIGHTS encodes that as follows. Returning to the previous example with the vertex i, the probability to jump to each of the 4, 5, 0 and 18 vertices is 0.1, 0.2, 0.3 and 0.4, respectively. These probabilities are stored in the attribute weights in a cumulative form:

```
>>> graph.weights[i]
array([0.0, 0.1, 0.3, 0.6, 0.6, 1.0])
```

Values j such that CONNECTIONS[I][J] is a negative integer are treated as edges with 0.0 probability.

Remark – WEIGHTS contains cumulative probabilities rather than actual probabilities for the purpose of optimization. However, this might eventually change to limit confusion, in which case a new attribute called WEIGHTS_CUM could be added.

Finally, graph objects come with a DATAFRAMEXS as the attribute DF_ATTRIBUTES. This dataframe has a row for each vertex in the graph, and is used to store all attributes associated with the graph's vertices. More precisely, the attributes of any vertex of index i are stored in the row i of the dataframe. For instance, in our epidemiological applications, a vertex represents a spatial unit characterized by a *carrying capacity* (K) representing the number of agents the unit can sustain. This is generally encoded in a column named K in DF_ATTRIBUTES, where DF_ATTRIBUTES['K'][I] is the carrying capacity of the vertex of index i.

2.5 Agents

Depending on the ecological system being modeled, agents can vary greatly, for example in their movement behaviour. Therefore, SamPy's modularity is particularly useful when defining agents and there are very few "core" components for SamPy agents. In this section, we introduce those few core agent components, briefly present other options, and refer the reader to Section 3.3 for an example demonstrating how SamPy can be adapted to the user's needs.

As explained in Section 1.1, there is no "single agent" object in SamPy. Instead, one instantiates a population object. The main attribute of such an object is a DataFrameXS called DF_POPULATION. Each row of DF_POPULATION corresponds to an agent, and each column corresponds to an attribute of the agents, such as age, sex, position, *etc.* Actions involving agents are performed using and modifying this dataframe. Attached to the dataframe is the attribute DICT_DEFAULT_VAL, which is a dictionary whose keys are the name of DF_POPULATION's columns, and values are the default parameters used in cases when new agents are created without specifying all of their attributes.

All other characteristics of the agents come from *building blocks* (see 2.1). Currently, the building blocks included within SamPy cover relatively basic capabilities, like the ability to search for a mate and reproduce in its home cell, or the ability to undertake unbiased, uncorrelated random walks on graphs.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 December 2022 doi:10.20944/preprints202211.0556.v2

2.6 Diseases and interventions

Currently, SamPy provides only one mode of disease transmission, which is direct contact transmission. That is, an individual can only transmit the disease to another agent located on the same vertex, which is the main reason why current built-in agents of SamPy have two different location attributes. The TERRITORY, which corresponds to the vertex on which the agent's territory is centered and is used for computing the agent's access to ressources, and POSITION that may vary during the simulation and is used to compute disease transmission. Position represents the fact that the agent's territory does not necessarily match the vertex shape and that the agents may access neighbouring vertices while moving around within their territory. Infected agents are characterized by one of three different states: infectious, contagious, and immune. Several modes of transition between those stages are provided.

Currently, SamPy provides two different types of interventions: culling and immunization. For culling, the user simply specifies the proportion of agents to be eliminated on each vertex and at what time step. Immunization is tied to a disease object and allows the user to protect agents from being susceptible to the disease. Note that this intervention-induced immunity is different from the natural immunity within SamPy, and several of its properties, such as the duration of agents' immunization to the disease, are user-specified. Similarly to culling, the user specifies the portion of agents to be immunized on each vertex, and at what time step.

3 SamPy applications

In this section, we provide examples of SamPy applications.

3.1 Population growth and disease.

In this section we continue with the artificial species and disease used as an example in Section 1.1. We illustrate how the population grows and stabilizes on three different landscapes, and then we infect a few agents located near the center of the landscape with the disease. All landscapes are square grids of shape 100x100, without any barrier to the agent's movement. They only differ by the spatial distribution of carrying capacity (K) across the landscape (see Figure 4). The script used for the map creation, the population build-up and the disease spread are ex2_create_maps.py, ex2_build_up.py and ex2_disease_unleashed.py, respectively. We start by building up a population on the three different landscapes (see Figure 5 for the population build-ups) , by introducing five breeding pairs in the vertex $(50, 50)$ and simulating 80 model years. The first landscape has a uniform K of 10, representing a broad area with relatively low population density. The second landscape represents a transition between two habitats, the lower and upper halves having a K of 50 and 10, respectively. Finally, the third landscape is obtained by applying a gaussian filter from **SciPy** to a white noise and then scaling the obtained smoothed noise by a factor of 20.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 December 2022 doi:10.20944/preprints202211.0556.v2

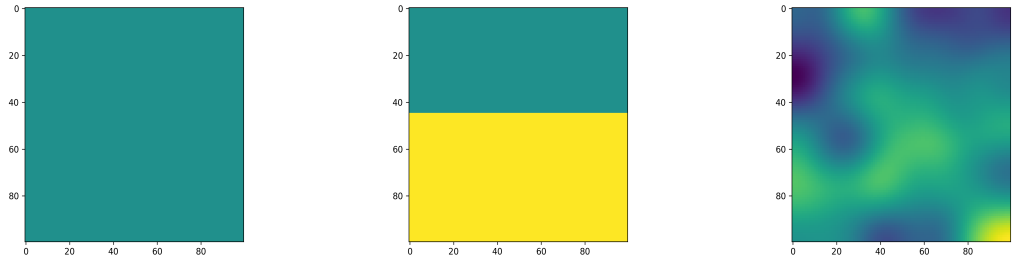

Figure 4: Left to right: map with uniform K of 10, map with upper half with K of 10 and bottom with K of 50, and map with random K between 0 and 20.

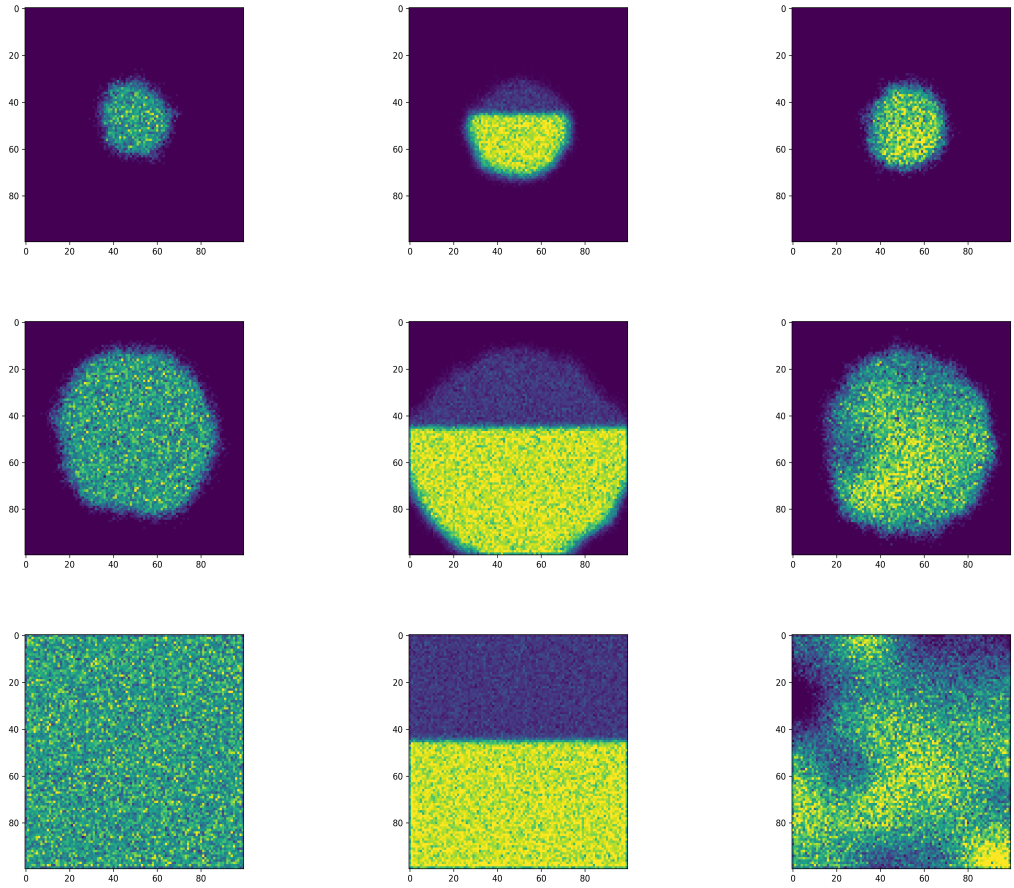

Figure 5: Population build-up on the three landscapes after 20 years (top row), 40 years (middle row) and 80 years (bottom row).

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 December 2022 doi:10.20944/preprints202211.0556.v2

As expected, the colonization of the landscape by the agents is faster in the areas where K is high, which is particularly visible in the second example where the bottom half 'fills' faster than the top half. In all cases, after 100 simulated years, an equilibrium is reached and the population distribution spatially matches K. We then introduce the disease (see example from Section 1.1) by infecting 50% of the agents in cells (50, 50) and (50, 51) in all three scenarios, and simulate three years of the epidemic (see Figure 6). As expected, the disease spreads faster in areas where K is high (i.e, where a higher

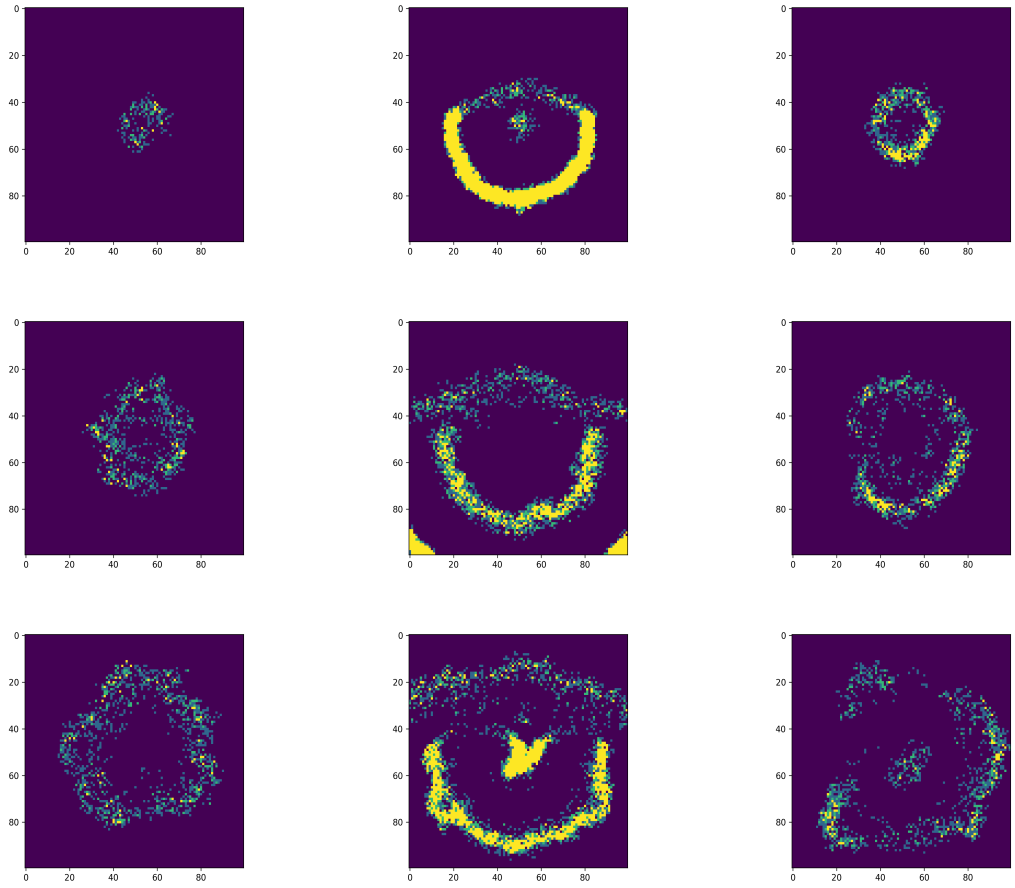

Figure 6: Map of infected individuals after 52 weeks (top row), 104 weeks (middle row) and 156 weeks (bottom row) on the three previous landscapes (uniform k of 10 on the leftmost column, the two halves on the middle and the smoothed noise on the rightmost column).

number of susceptible agents are available). This is particularly visible on the second landscape, where the disease wave front crossed the lower half several times during the three years simulation, while it failed to cross the upper half during the same period.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 December 2022 doi:10.20944/preprints202211.0556.v2

3.2 Intervention

In this section, we study the effect of various immunization strategies on the spread of the disease used for the example in Section 3.1. We use the same artificial species on a square-grid landscape of shape 100 x 100 with a uniform 'K' of 10, which results in a population of approximately 100 000 agents at equilibrium (with large spikes in population abundance at birth weeks). Let us assume that each vertex of the landscape represents a square spatial unit with sides 1 km in length. As in previous examples, we initiaite the epidemic by infecting 50% of the agents in vertices $(50, 50)$ and $(50, 51)$.

Our aim is to evaluate the relative effectiveness of a range of different immunisation strategies applied at three different times of the year. Each strategy consists of a single and uniform immunization of agents located on vertices within a disc-shaped area of the landscape of variable radius, these disks being centered on the vertex $(50, 50)$ (see Figure 7).

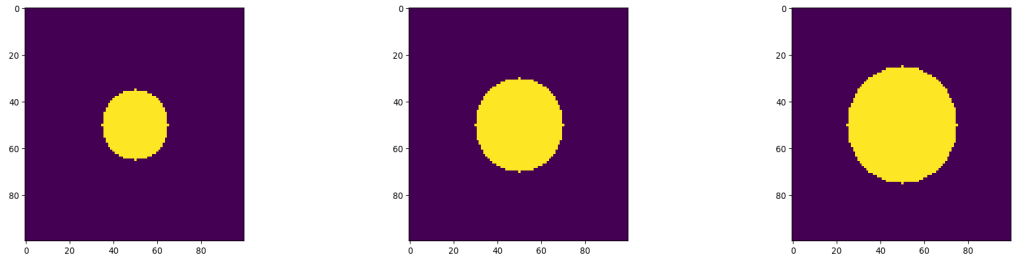

Figure 7: Example of disks where the vaccination is applied. Their radiuses are 15 km, 20 km and 25 km from left to right.

When immunization is applied, an equal proportion of agents from each vertex located within the immunization zone is immunized. This probability is called the 'level' of immunization. In this example, we test all possible strategies with immunization zone radius varying from 10 km to 44 km (with increments of 1 km), immunization levels varying between 0.05 and 0.95 (increments of 0.05), and with application week set to either 16, 32 or 52. Each configuration is simulated 20 times for a simulated period of 156 weeks (*i.e.* three years) of disease spread. The number of active disease cases is extracted after each simulated week. Figure 8 shows some of the curves obtained by varying the immunization level.

Remark – In the example presented in the current section, we ran 42 000 simulations. Each took 6.72 seconds to run on average on a AMD Rome 7532 at 2.40 GHz (see the description of Compute Canada Narval nodes at https://docs.computecanada.ca/wiki/Narval for more information).

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 December 2022 doi:10.20944/preprints202211.0556.v2

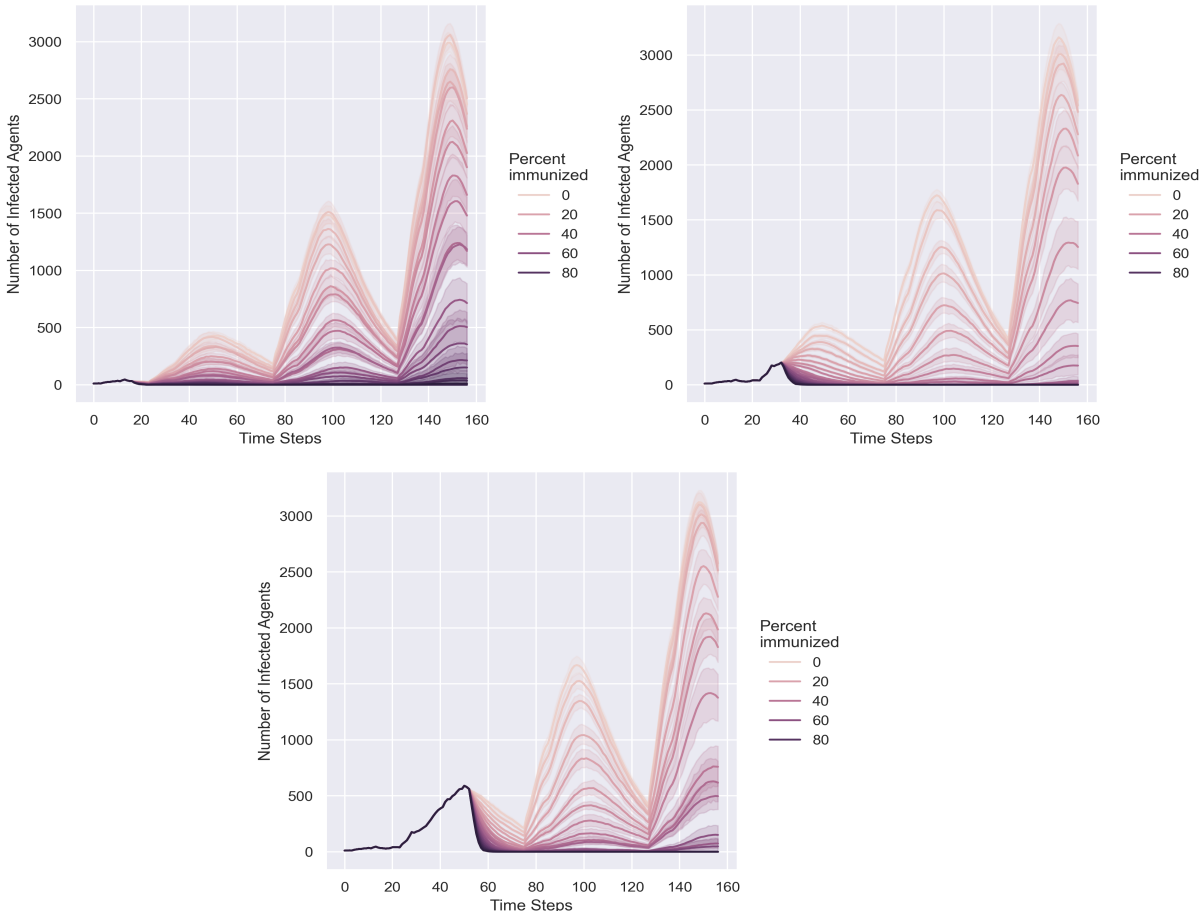

Figure 8: Evolution of the number of active cases when immunization is applied on a disk of radius 20 km at week 16 (top left), week 32 (top right) and week 52 (bottom). These simulations were done with varying level of immunization (from 0% to 95%).

3.3 How to add new functionalities with the example of ORM

In this section, we demonstrate how a user can add new functionalities to SamPy either by creating new building blocks (see Section 2.1), or by using existing building blocks to create personalized SamPy objects. We focus on the example of the Ontario Rabies Model (ORM; see [24]). The ORM has been extensively used and validated to model rabies dynamics in racoons and mongooses ([20], [21], [23]), but speed and further development limitations related to this model have motivated applying SamPy to model rabies dynamics in these systems. As a result, current ORM users expressed the need to translated their ORM landscapes and agents into equivalent objects in SamPy. How this could be done presents a useful illustration of SamPy's flexibility and development potential.

First, we created a new graph object to integrate ORM landscapes into SamPy. In

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 December 2022 doi:10.20944/preprints202211.0556.v2

ORM, landscapes are represented as two-dimensional hexagonal grids (*e.g.*, rightmost picture in Figure 3). ORM landscapes are generally created using QGIS and exported as XML files. Each cell is oriented, i.e. each of the 6 neighbours of a given hexagonal cell is either its north, north-east, south-east, *etc* (Figure 9) neighbour.

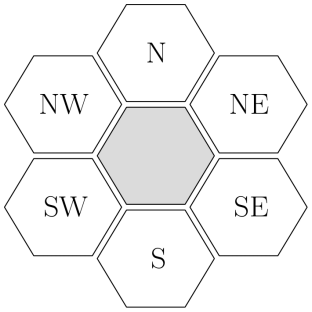

Figure 9: A grid cell with its neighbours.

In addition, the XML encodes other cell-specific information, *e.g.* its geographic coordinates and carrying capacity. In order to integrate these additional features in SamPy landscapes, we developed the graph object GraphFromORMxml (see the file ORM_related_addons/graph_from_ORM_xml.py). To instantiate this object, one provides a path to an ORM XML file using the kwarg PATH_TO_XML. This XML is then translated into a SamPy graph, by assigning an index to each cell, filling the dataframe DF_ATTRIBUTES with all the information stored in the XML, and creating the CONNECTIONS and WEIGHTS arrays. The orientation of the cells is encoded within these arrays as depicted in Figure 10.

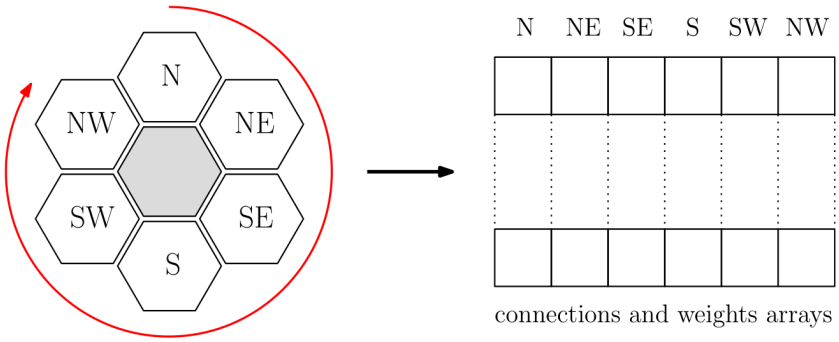

Figure 10: Representation of how the orientation is encoded in connections and weights arrays.

The cell orientation encoded into the arrays connections and weights is essentially invisible for native SamPy agents (*i.e.*, a script using built-in Sampy agents can use an ORM-derived graph without any issue). In contrast, this orientation is used extensively by ORM agents during *dispersal* (*i.e.*, when they disperse from their home cell). Indeed, dispersal is modeled as a discrete correlated random walk in ORM, where agents perform

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 December 2022 doi:10.20944/preprints202211.0556.v2

a series of discrete movement steps, and at each new step they have a 60% chance of continuing in the same direction, 20% chance of turning right and 20% chance of turning left (Figure 11). In addition, movements in ORM can be altered by two resistance parameters related to each cell (InRes and OutRes), which can be used to create topographic or landscape-related barriers to dispersal by adjusting the ease with which agents can leave and enter a given cell.

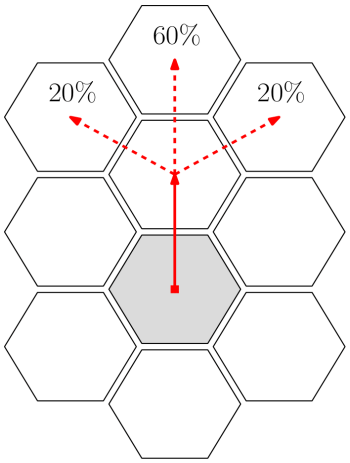

Figure 11: Two consecutive steps of dispersion in ORM.

Then, we created a new building block which can be used to create ORM-like agents, ComponentsFromORM, (see ORM_related_addons/ORM_like_agents.py). This building
block contains an implementation of the ORM dispersal method (orm_dispersion), as well as movement alteration methods which take into account the resistance to enter and leave cells encoded into the ORM landscape XMLComponentsFromORM. We also included a new method for modeling natural mortality of agents taken from the latest version of ORM (method mortality_from_v08).

4 Conclusion and future directions

SamPy provides a flexible and efficient tool to create ABMs for modelling infectious diseases on complexe landscapes and with a large number of agents. SamPy models can involve hundreds of thousands of agents, while running efficiently on a personal computer. This allows users to assess the effects of various disease control strategies in a timely manner, without requiring access to high performance computing systems.

SamPy's design based on building blocks makes it highly flexible, including great potential for future extensions. Future development plans will focus on extending the options available for modelling animal behaviour, mainly by adding complex movement methods. As an example, see the file "sampy/agents/random_walk.py" on GitHub, which contains a preliminary implementation of correlated random walks for SamPy's agents.

 doi:10.20944/preprints202211.0556.v2

In addition, we will develop general extensions, including the ability to generate models in which several species interact directly (currently ABMs with several distinct populations of agents can be created, but these populations cannot interact with one another). Finally, we plan on introducing new disease transmission modes (*e.g.*, vector-borne diseases)

Acknowledgments

The figures shown in this article were made using **Matplotlib** [7] and **Seaborn** [28] libraries. This project was funded by the Ministère de la Santé et des Services sociaux du Québec (MSSS) and a NSERC Discovery Grant (RGPIN-2021-04324) to P. Leighton, and was enabled in part by support provided by Calcul Quebec (calculquebec.ca) and the Digital Research Alliance of Canada (alliancecan.ca).

References

[1] Federico Bianchi and Flaminio Squazzoni. Agent-based models in sociology. *WIREs Computational Statistics*, 7(4):284–306, 2015.

[2] Gilad Bracha and William Cook. Mixin-based inheritance. *ACM SIGPLAN Notices*, 25, 10 1998.

[3] Fred Brauer, Pauline Driessche, and Jianhong Wu, editors. *Mathematical Epidemiology*. Springer, Berlin, 1 edition, 2008.

[4] Kenneth Comer and Andrew Loerch. The impact of agent activation on population behavior in an agent-based model of civil revolt. *Procedia Computer Science*, 20:183–188, 12 2013.

[5] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[6] Elizabeth Hunter, Brian Mac Namee, and John D. Kelleher. A comparison of agent-based models and equation based models for infectious disease epidemiology. *26th AIAI Irish Conference on Artificial Intelligence and Cognitive Science*, 2018.

[7] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[8] Fred James and Lorenzo Moneta. Review of high-quality random number generators. 03 2019.

[9] K. E. Jones, N. G. Patel, M. A. Levy, A. Storeygard, D. Balk, J. L. Gittleman, and P. Daszak. Global trends in emerging infectious diseases. *Nature*, (451):990–993, 2008.

[10] Jackie Kazil, David Masad, and Andrew Crooks. Utilizing python for agent-based modeling: The mesa framework. In Robert Thomson, Halil Bisgin, Christopher Dancy, Ayaz Hyder, and Muhammad Hussain, editors, *Social, Cultural, and Behavioral Modeling*, pages 308–317, Cham, 2020. Springer International Publishing.

[11] Ryan Kennedy, Kelly Lane-deGraaf, S. M. Niaz Arifin, Agustin Fuentes, Hope Hollocher, and Gregory Madey. A gis aware agent-based model of pathogen transmission. *International Journal of Intelligent Control and Systems*, 14, 01 2009.

[12] Lisa Kosiachenko, Nathaniel Hart, and Munehiro Fukuda. Mass cuda: A general gpu parallelization framework for agent-based models. In Yves Demazeau, Eric Matson, Juan Manuel Corchado, and Fernando De la Prieta, editors, *Advances in Practical Applications of Survivable Agents and Multi-Agent Systems: The PAAMS Collection*, pages 139–152, Cham, 2019. Springer International Publishing.

[13] Kalliopi Kravari and Nick Bassiliades. A survey of agent platforms. *Journal of Artificial Societies and Social Simulation*, 18(1):11, 2015.

[14] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, pages 1–6, 2015.

[15] Sean Luke, Claudio Cioffi, Liviu Panait, Keith Sullivan, and Gabriel Balan. Mason: A multiagent simulation environment. *Simulation*, 81:517–527, 01 2005.

[16] Mikola Lysenko and Roshan M. D'Souza. A framework for megascale agent based model simulations on graphics processing units. *Journal of Artificial Societies and Social Simulation*, 11(4):10, 2008.

[17] Adam J. McLane, Christina Semeniuk, Gregory J. McDermid, and Danielle J. Marceau. The role of agent-based models in wildlife ecology and management. *Ecological Modelling*, 222(8):1544–1556, 2011.

[18] J. Panovska-Griffiths. Can mathematical modelling solve the current covid-19 crisis? *BMC Public Health*, 20(551), 2020.

[19] L. A. Real and R. Biek. Spatial dynamics and genetics of infectious disease on heterogeneous landscapes. *Journal of the Royal Society Interface*, 4(16):935–948, 2007.

[20] Erin Rees, Bruce Pond, Catherine Cullingham, Rowland Tinline, David Ball, Christopher Kyle, and Bradley White. Landscape modelling spatial bottlenecks: Implications for raccoon rabies disease spread. *Biology letters*, 5:387–90, 04 2009.

 doi:10.20944/preprints202211.0556.v2

[21] Erin Rees, Bruce Pond, Rowland Tinline, and Denise Bélanger. Modelling the effect of landscape heterogeneity on the efficacy of vaccination for wildlife infectious disease control. *Journal of Applied Ecology*, 50, 08 2013.

[22] E Samanidou, E Zschischang, D Stauffer, and T Lux. Agent-based models of financial markets. *Reports on Progress in Physics*, 70(3):409–450, feb 2007.

[23] Caroline Sauvé, Erin Rees, Amy Gilbert, Are Berentsen, Agathe Allibert, and Patrick Leighton. Modeling mongoose rabies in the caribbean: A model-guided fieldwork approach to identify research priorities. *Viruses*, 13:323, 02 2021.

[24] R. R. Tinline. *The Ontario Rabies Model Guide.* Queen's University, Ontario, Canada, 2007.

[25] Guido Van Rossum and Fred L. Drake. ***Python*** *3 Reference Manual.* CreateSpace, Scotts Valley, CA, 2009.

[26] Srinivasan Venkatramanan, Bryan Lewis, Jiangzhuo Chen, Dave Higdon, Anil Vullikanti, and Madhav Marathe. Using data-driven agent-based models for forecasting emerging infectious diseases. *Epidemics*, 22:43–49, 2018. The RAPIDD Ebola Forecasting Challenge.

[27] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[28] Michael L. Waskom. seaborn: statistical data visualization. *Journal of Open Source Software*, 6(60):3021, 2021.

[29] Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56 – 61, 2010.

[30] Uri Wilensky. Netlogo. http://ccl.northwestern.edu/netlogo/, Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL, 1999.

[31] Peter Wittek and Xavier Rubio-Campillo. Scalable agent-based modelling with cloud hpc resources for social simulations. pages 355–362, 12 2012.