JACOBS UNIVERSITY

ADVANCED PROJECT I

Project Report

Supervisor: Prof. Dr. Agostino Merico

Contributor: Dr. Davi Tavares

Virtual Environment for Individual-Based Modeling

By Ralph Florent

May 29, 2019

Abstract

This report provides some facilities for understanding the virtual environment implemented to study the habitat use by waterbirds in coastal lagoons of the tropics. This virtual environment is based on an Agent-Based Modeling system using some assumptions that are derived from previous observations of waterbirds' behaviour within some habitats in the tropics. Further, some detailed information is given about the carried-out tests and their results.

1 Introduction

Complex Systems is a field of science studying how individual components of a system give rise to collective behaviours and how the system interacts with its environment [1]. One of the approaches to study a system interaction with its environment is through Agent-Based Modeling (ABM), a generalized framework for modeling and simulating dynamical systems.

In our case, we intend to study the habitat use by waterbirds in coastal lagoons of the tropics. ABM, besides being a computational simulation, is the closest modeling assumptions capable of providing a deeper understanding and interpretability of such a system.

This project report outlines the different steps to achieve a Virtual Environment (VE) using an ABM technique to characterize the waterbirds behaviours, adaptation and evolution within a set of habitats with distinguishable properties. These steps are a reference to the Python code implementation of this VE, which serves as a demonstration basis to run and simulate the ABM. Finally, the results are analyzed and discussed in accordance with the algorithmic methods, the content structure, and the workflow scheme that are derived mostly from the representative traits of each component of the system.

2 Theoretical Background

2.1 Waterbirds and Environmental Factors

Tropical coastal lagoons are shallow aquatic ecosystems located at the boundary between terrestrial and marine environments [2]. The high environmental heterogeneity of coastal lagoons, in both temporal and spatial scales, provides habitats for aquatic bird species with different ecological needs [3, 4, 5]. These habitats are mainly characterized by seven environmental variables and classified into three groups:

1. *structural*: vegetation height, lagoon size and water depth;

2. *hydrochemical*: water salinity and pH;

3. *anthropogenic*: livestock grazing pressure and distance from human settlements;

and water depth is the most important variable influencing the waterbird assemblage [2].

The aquatic birds or *waterbirds* inhabiting the coastal lagoons were, according to a survey conducted by Tavares D.C. et al in 2015 [2], grouped into guilds[1] reflecting species' foraging habits and morphology. The six identified guilds were: diving birds (grebes), dabbling ducks (belonging to the

[1]The guild concept was proposed by Blondel in 2003 [6].

genera Dendrocygna and Anas), large wading birds (herons, egrets and storks), vegetation gleaners (jacanas and gallinules), fishing birds (gulls and terns) and small wading birds [7].

2.2 Agent-Based Modeling

Agent-Based Models are computational simulation models that involve many discrete agents [1]. This computational simulation is usually based on intense processings and algorithmic calculations due to the fact the typical context in which the ABM is used is to study the collective behaviour of large number of components or agents.

An *agent* is a component or an entitiy of the system and contains usually the following properties: internal states, spatial locations, interaction with the environment, interaction with each other, behaviour rules, adaptation and evolution. Depending on the goal of the ABM, some additional properties may or not be incorporated into the model. For instance, certain agents can be attributed the role of *central controllers*.

The code implementation of an ABM can particularly be as heavy as its model complexity increases. Hence, it is viable to start off with some uncomplicated settings and assumptions in order to favour a straightforward analysis of the results that are obtained after running the simulation. Afterwards, one can subsequently tranform the model by adding more complexities. On the other hand, from a programming point of view, the code maintenance and organization are a relevant factor that contributes to debug relatively faster as the amount of coding increases.

3 Instrumentation

The VE, as specified literally, is developed in a complete *virtualized* workspace. This virtualized workspace is made up of tools and software used to carry out this project to its current release. In this section, a brief overview of those tools and software is provided to help to reproduce or replicate the exact setup of the development environment put in place at the time of implementing the project.

3.1 Tools and Software

There are several currently-available programming tools that may achieve the same VE goal. The reason to believe so is that it turns out that today's open source community has grown larger and, subsequently, has been more actively involved in software improvements and new releases. As a result, accessing those online tools is no longer an issue, at least in terms of low-money budget, since they are publicly available (under free or moderately limited license).

Given the availability of several options, enlisted below are the most regular choices of tools and software for a developer with mere knowledge in programming:

- GNU/Linux Ubuntu 16.04 (operating system)
- Visual Studio Code (text editor for the documentation)
- Git[2] (version control)
- GitHub (web-based hosting service for versioning system)
- Python (programming language for the scripting)
- Jupyter Notebook (workspace for the VE simulation)

Obviously, it is not a concern to access and use a set of randomly compatible versions of the above-mentioned tools and software. However, in case a developer wants the exact versions, Table 1 lists more detailed information on both the versions and sources for future downloads.

Tools & Software			
	Versions	**Sources**	**Cost**
Visual Studio Code	1.34.0	See [8]	Free
Git	2.7.4	Built-in Linux program	Free
GitHub	N/A	See [9]	5 free users
Python	3.5	See [10]	Free
Jupyter Notebook	5.7.4	See [11]	Free

Table 1: Detailed information on the tools and software used for the VE

3.2 General Comments

The tools and software discussed in the previous subsection are chosen by a matter of personal preference. No further comparison or parallelism procedure has been carried out to assess the most convenient option. That is to say, it might exist a better work environment where the VE simulation

[2] Also available as a bash emulation for other platforms for free (e.g. Git Bash for Windows).

is simpler and/or easier, or the VE surprisingly performs better[3]. But, given that this first release is most importantly seen as a prototype, more tools and software can be tested out in a near future so that we end up with a so-called optimal workspace for the VE.

4 Methodology

This section will explore the methods used to implement the core functionality of this project. This exploration includes the mention of the workflow scheme, the third-party libraries usage and options, the algorithm and content structure, and finally the programmatically-implemented coding procedure.

4.1 Workflow Scheme

This project's workflow scheme consists of 3 main steps:

1. *Initialize*: stands for initial conditions
2. *Observe*: handles the graphical parts
3. *Update*: computes random movements based on the probability distribution of the corresponding factors.

where each step contains itself a series of internal subprocesses aiming a specific goal.

Important: *Observe in Figure 1 the remaining steps categorized as* Preconditions *and* Postconditions*. They represent respectively the* Before *and* After *the 3 main steps* `Initialize, Observe`*, and* `Update` *are executed. Note also that the* `Initialize` *process is considered part of the Preconditions semantics. That is because it only prepares the basic conditions for the components of the system, which are the habitats and the birds.*

Analyzing the workflow diagram in Figure 1, we denote the following fields:

- ***Start***: indicates the starting point of the VE simulation.
- ***Prior Considerations***: are the basic setup necessary to fulfill the initialization phase requirements[4]. This setup spans the following elements: the geometry of the habitats and the human settlements; the functions defining the probability distribution of the random movements (driven

[3]In the outlook section, "simpler" and "easier" simulation is explained with the perspective of an ideal use case scenario. Similarly, a better performance of the VE refers to reduction in processing time, resource consumption in an easy-to-follow simulation platform.

[4]These considerations, mostly based on the concerned entities (waterbirds, coastal lagoons), the environmental variables, and any additional properties contributing to the setup phase of the VE simulation, are also discussed in this document in the theoretical section.

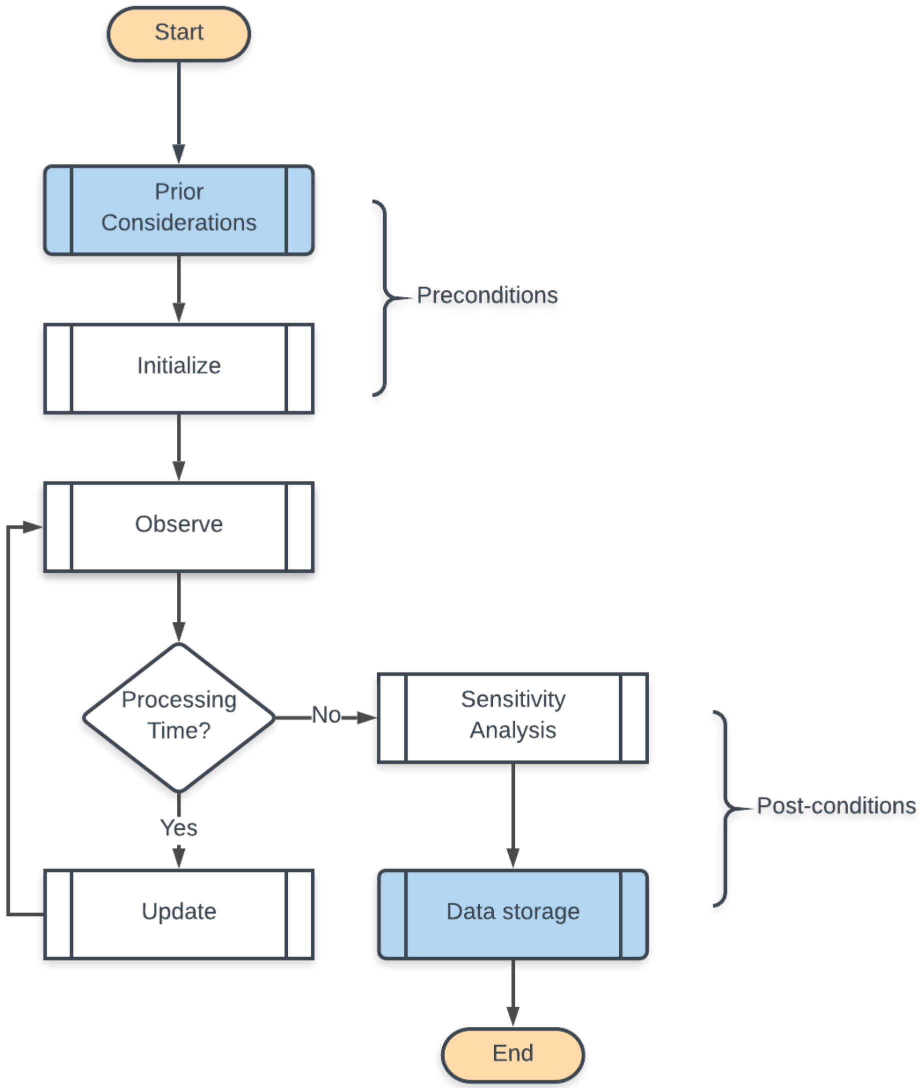


Figure 1: Workflow diagram
(credits: made with *Lucidchart*)

by the water salinity, water depth, and food availability factors); the duration of the overall simulation process; and a reasonable threshold to handle the feasability of the random movements for a given seabird under certain conditions.

- ***Initialize***: creates the initial conditions of the system based on prior considerations mentioned above. That is, the patches (habitats) and agents (seabirds) creation.

- ***Observe***: generates a 2-dimensional plot whose scale goes from zero to one$(0-1)$ in both axes (x, y). The rendered plot helps to visualize both the patches' and agents' positions.

- ***Processing Time?***: focuses on updating the agents' positions' as long as the conditional parameter for the processing time holds. That is, the iteration is exclusively based on a specific number of times without accounting for other parameters that might influence the habitats and the birds. Note that, in this current version, the iteration is set statically during the prior considerations process.
- ***Update***: randomly assigns an agent to new positions within the existing habitats, considering a given threshold and the other aspects of the probability distribution.
- ***Sensitivity Analysis***: collects the probability values to form a set of probability distributions, which later can be analyzed and compared to each other with the expectation to draw conclusions on the final output.
- ***Data Storage***: given the generated plots, collects them as PNG images and then generates a GIF out of the entire dumped images. This is relevant to provide the end-user useful insights on the collected data.
- ***End***: indicates the ending point of the VE simulation.

Recalling that this Virtual Environment constitutes essentially a digital representation of an Agent-Based Modeling system, each component of such a system relies on the interaction and interconnection with other involved components in an organized flow. Therefore, the diagram in Figure 1 shows a workflow scheme that intends to provide with a visual aid for a better understanding of the system's behaviour.

4.2 Algorithm & Data Structure

The VE simulation implies the use of well-coordinated processes and subprocesses, which, once computed, will eventually attempt to explain the agents' behavior ant their mutual interactions with the environment in which they coexist. This section discusses the algorithm and data structure applied to contruct these processes and subprocesses.

4.2.1 The *Habitat* and *Agent* data structure

In the VE simulation, both the wetland areas and the human settlements of the coastal lagoons are represented by the term *Habitat*[5], and the waterbirds, by the term *Agent*. In this case, the

[5]Note that human settlements are simply less appealing habitats for the waterbirds due to the humans' threatening characteristics

concept ”Habitat” is a 2-dimensional *static* polygonal shape drawn from certain given geometrical measurements (see Figure 2). Similarly, the concept ”Agent” is simply the representation of the waterbirds with some of its characteristics or attributes.

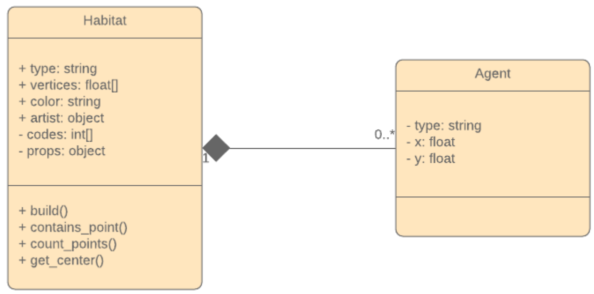


Figure 2: Data structure of *Habitat* and *Agent*
(credits: made with *Lucidchart*)

Observe that in Figure 2 we use the class diagram named *UML (Unified Modeling Language)* to model and document the properties of the components: Habitat and Agent. On the one hand, we construct a Habitat class definition with the following properties:

- ***type***: the type or category name of the habitat;
- ***vertices***: the coordinates of the patch representing the habitat;
- ***color***: the color (edge and face) to apply or distinguish a habitat from another;
- ***artist***: the patch-based polygonal shape to draw on a given figure;
- ***props***: the dictionary-like additional properties that characterize this habitat;
- ***build()***: constructs the *artist* or patch on a figure;
- ***contains_point()***: determines whether or not an x-y coordinate (point) belongs to a patch;
- ***count_points()***: counts the total number of agents located within the patch based on their x-y positions.
- ***get_center()***: obtains the center point (x-y coordinate) of this habitat.

On the other hand, we construct an Agent class definition with the following properties:

- ***type***: the type or category name of the agent species;
- ***x***: the x-coordinate of the agent within an area;

- ***y***: the y-coordinate of the agent within an area.

Important: *Keep in mind that some of the methods in the class definitions use helper functions to do their specific task. These helpers can be found in the Python scripts located in Appendix A.*

4.2.2 The overall algorithm

The overall algorithm is quite based on the step-by-step flow chart described in Figure 1. In other words, it corresponds to the descriptive, logical aspects of the core functionality of the VE. The steps are as follows:

1. ***Given***: given a collection of geometrical measurements (design) of the existing habitats and human settlements in a specific environment, a finite number (relatively small, 20 for example) of seabirds, and a set of predefined probability distribution functions (PDF) whose arguments are the characteristics of that environment;
2. ***Initialization***: represent digitally (virtually) that environment by creating patches and agents;
3. ***Update***: randomly choose an agent, then assess the probability of it moving to a random destination, and finally move the agent (if doable);
4. ***Observe***: snapshot the current state of the plotted environment, then save figure as a PNG image;
5. ***Iterate***: Repeat steps **3** & **4** for n times;
6. ***Stop***: collect the dumped images and form GIF final image to visualize the random movements of the agents.

4.2.3 The *Update* algorithm

Some of the processes are really straightforward and do not demand a time-, or energy-consuming logic to build them. For instance, the initialization phase is one of the common cases where the developer only needs to take care of statically sets of values required as prior considerations for the initial conditions. But, as for the *Update* process, a thoughtful, analytical solution is needed.

This algorithm basically defines an asynchronous approach to randomly update an agent's status, namely its geolocation. Thus, the set of instructions that follows below is the algorithm used to accomplish the "*Synchronous Update*" functionality of the VE simulation:

1. ***Given***: given a randomly selected agent;

2. ***Initialization***: randomly choose a new destination within an "acceptable" habitat (an area where this agent can move to, given the environmental conditions);

3. ***Computation***: compute the probability of that new destination use for this agent.

4. ***Update***: finally, move the selected agent to that new destination if the calculated probability complies with the threshold.

Recalling that this version of the project is a prototype whose purpose is to virtualize a static Agent-Based Modeling system, these algorithms are defined in their most simplistic mode. For this reason, they are subject to change in the future when it comes to updating the dynamics of the system or adding more complex variations.

4.3 Implementation

As mentioned in the *Tools and Software* subsection in *Instrumentation*, the VE simulation is implemented in a Jupyter Notebook workspace using the Python programming language. They are many reasons for choosing this particular setting to develop this workspace and the free cost is one of them.

The code implementation is based on the flow diagram presented in Figure 1 as well as the algorithms and data structure described in the previous subsection. Here, we mostly focus on the coding procedure and the programming standards to facilitate other colloborators' contributions and support in the future.

A standard programming workflow, if it does not involve too much of team management, demands to follow a set of intended principles[6] that takes a system from a development stage to a production stage. For instance, the code should be: *architected, modular, standardized, structured, scalable, secure, performance-oriented, tested, testable, collaborative, time-estimated, documented, and so on*[12]. These principles are very common among big tech companies' projects and can also be used for smaller, or startup projects.

Since this actual version of the VE simulation is an early prototype, we focus on following part of these principles so far. Among them, figure:

- ***architected***: the overall system follows a series of well-planned, modularized, interconnected conceptual tasks describing the interaction of the components within the system;

- ***standardized***: the script follows the rules for the naming conventions in Python (variable names, function and class definition, etc);

[6] Those principles vary among institutions. So far, there is not yet a clear proposed draft describing them. Therefore, following them remains subjective.

- ***structured***: the script is written semantically and logically, and organized sequentially (3rd-party libraries import, constants declaration, functions definition, and *main*[7]);
- ***scalable***: the script can be easily extended for new releases and anticipates enhancements in the future;
- ***collaborative***: the code is version-controlled using Git and GitHub online hosting service;
- ***documented***: the script is well-documented and describes the coding content in a very human-friendly way.

Besides the coding procedure, we also created a commonly-standardized, organized file structure (See Figure 3) for the project. Note the parent folders named *src* and *docs*. The former is for the source code of the project and the latter, for the documentation. The contents under those folders are backed-up and synced with an online GitHub repository for versioning and collaboration reasons.

Writing test is currently out of the scope of this release. We understand that using *Unit Testing* and *Integration Testing* for the implemented code is relatively important and should be covered. For now, the code is maintained and tested throughout the outputs and the visualizations as expected. But as for future updates or releases, the new implementation should be written using test-driven scripts methodology as the scalability of the project will make the code cumbersome to maintain and test.

4.4 Third-Party Libraries

Like in most of software-based systems, not all the components (or parties) of such systems are built from scratch. That is also true for our VE simulation prototype. It is built on top of certain third-party libraries as illustrated in Table 2.

One of the core principles in Software Engineering is *DRY (**D**o not **R**epeat **Y**ourself)*. This acronym encourages developers to avoid code duplication and focus on configurable and reusable components [19]. With that being said, we focus mainly on some third-party component reusability. That, of course, comes with its pros and cons:

- ***pros***: time saving, pre-tested code usage, modular code usage, etc.
- ***cons***: dependency, lack of support, overuse, security issues, etc.

The key point behind this brief pros-and-cons topic is to signal that it important to carefully select the right libraries to use, which is what we did in the first place.

[7] Main entry function to run an application

```
.
├── assets
├── concept_1v.png
├── dist
│   └── frame
├── docs
│   └── latex
│       ├── images
│       │   ├── data-structure.png
│       │   ├── logo-black.png
│       │   └── workflow-scheme.png
│       ├── main.pdf
│       ├── main.tex
│       ├── references.bib
│       └── sections
│           ├── abstract.tex
│           ├── contents.tex
│           └── intro.tex
├── LICENSE
├── README.md
└── src
    └── notebooks
        └── main.ipynb
```

Figure 3: File structure of the project

Third-Party Libraries			
	Versions	**Sources**	**Features**
numpy	1.15.4	See [13, 14]	*random, linalg.norm*
matplotlib	3.0.2	See [15, 16, 17]	*patches, path, pyplot*
imageio	2.4.1	See [18]	*imread, mimsave*

Table 2: Detailed information on the third-party libraries used in the VE simulation

5 Results & Discussions

The virtual environment prototype is the end result of 2 combined pilot tests following the general model concept illustrated in Figure 4. The results of these tests are presented in the subsections that follow.

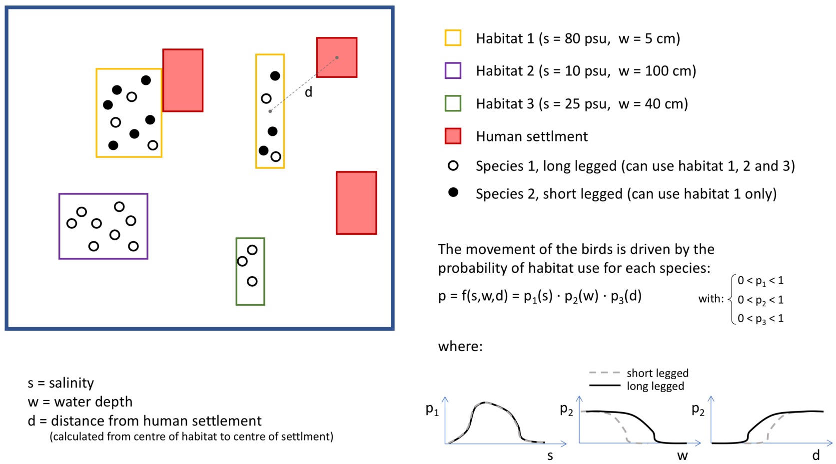


Figure 4: General model concept for studying habitat use by waterbirds in coastal lagoons of the tropics. The functions for calculating the probability of birds moving using a given water depth, distance from human settlement and salinity can be found in Appendix A.

5.1 First Pilot Test

In this very first test, we aim for the most possible, simplistic Agent-Based Modeling simulation. The objective of this test is to simulate a physical environment of static habitats and a few randomly-positioned waterbirds with the expectations of moving around over time. In addition to that, we create a simple restriction rule, which is "*the waterbirds are not allowed to visit the habitats*".

Simulating this startup environment requires a focus on drawing specific patches (representing the habitats) and generating a finite number of agents (representing the waterbirds). In this case scenario, we use a square figure scaled from 0 to 1 in both sides as the limited area for the environment (see Figure 5). Then, with the help of the *matplotlib* 3rd-party library, we produce the patches. Similarly, we use object-based definitions to create the agents and randomly position them within the environment using some helper functions. No prior considerations.

5.1.1 The patches

A patch is a drawing version of a habitat. It has a non-, or polygonal shape and tries to represent as closely as possible a 2-dimensional structure in reality. The shape can be very simple, as in the case of a simple square; and very complex, as in the case of a curvy closed-line. Besides its structural representation, a patch has some other properties such as *facecolor*, *edgecolor*, *lineweight*, and so forth, that are related to the setting of the drawing itself (see more details in [15]). But in our case, we choose to go with the rectangular shape, which we consider sufficient for the demo.

In earlier sections of this document, we introduced the Habitat class definition with some attributes (*type, vertices, color, etc.*) that is to digitally represent a physical habitat or human settlement. But in this first pilot test, we use only a lightweight version of these attributes, which are the vertices. To illustrate our point about drawing a patch (or an artist) within an area, see the Python scripts below:

```
1  import matplotlib.patches as Patches
2  from matplotlib.path import Path
3  
4  # vertices of the rectangle
5  verts = [
6      (0.2, 0.2), # left, bottom
7      (0.2, 0.4), # left, top
8      (0.4, 0.4), # right, top
9      (0.4, 0.2), # right, bottom
10     (0.2, 0.4), # ignored
11 ]
12 
13 # how to draw the lines
14 codes = [
15     Path.MOVETO, # start designing here
16     Path.LINETO, # draw line to
17     Path.LINETO, # draw line to
18     Path.LINETO, # draw line to
19     Path.CLOSEPOLY,# finish polycurve here
20 ]
21 
22 # create the final plottable rectangle
23 path = Path(verts, codes)
24 patch = Patches.PathPatch(path, facecolor='b', alpha=0.5, lw=2)
25 
26 # omit scripts for plotting
```

Listing 1: Script for creating a patch using *matploblib*

Observe how the codes in lines 23-24 in Listing 1 create a final plottable object that can be later drawn in a figure.

5.1.2 The agents

An agent is a drawing version of a waterbird. It can be of the type *short-legged* or *long-legged* species. This little nuance is what constitutes the core distinction in the waterbirds' behaviour within the habitats. Therefore, it remains relevant to set a clear cut in the design so that a short-legged type visually differs from a long-legged type.

Recall that the Agent class definition is really simple in terms of properties: *type and x-y positions*. The *type* attribute takes the value of either "short-legged" or "long-legged" and the agent's position is the x-y coordinates that take decimal values between 0 and 1 within a 2-dimensional area. As a result, creating an agent in the VE simulation is simply to instantiate an object of the on-the-fly class Agent, then define its type as "short-legged" or "long-legged", and finally assign a random position to that agent. However, due to the restriction rule mentioned previously, the randomly-generated positions cannot fall into the area occupied by the habitats.

```python
import numpy as np
from matplotlib.path import Path
import matplotlib.patches as Patches

# omit scripts for patch creation...

def gen_random_point(patches):
    x, y = np.random.rand(2) # initialize random point(x, y): [0-1, 0-1]
    while True:
        found = False # flag to determine when to stop iterating
        for p in patches:
            if p.get_path().contains_point((x, y)):
                found = True
        if not found: break # ice breaker
        x, y = np.random.rand(2) # update point(x, y)
    return (x, y)

# on-the-fly agent class definition
class Agent:
    pass

# create agents
def create_agents(n_agents):
    global patches # make previously created patches available
    agents = []
    for i in range(n_agents):
        agent = Agent()
        agent.type = "short-legged"
        x, y = gen_random_point(patches) # that is not in patch
        agent.x, agent.y = x, y # new position being assigned to this agent
        agents.append(agent) # append (i.e. add) the ith agent into the array 'agents'
    return agents

# omit scripts for plotting...
```

Listing 2: Script for creating an agent using the *gen_random_point()* helper

Finally, the results of creating both patches and agents for the first pilot test are illustrated in Figure 5.

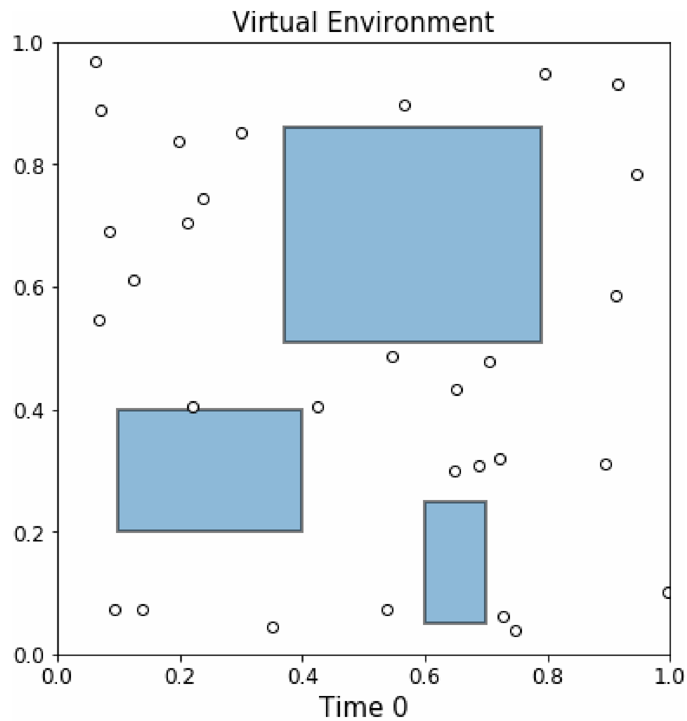


Figure 5: Preview of the first pilot test results: there is a total of 3 static patches (of different sizes) and 30 randomly-positioned short-legged agents. This visualization is the first generated plot of 20. The entire creation process is repeated 20 times and a final GIF image is built up out of the 20 generated images for a better visualization.

5.2 Second Pilot Test

In the first pilot test, we try to wrap up some key concepts and terminologies to avoid confusion with the interpretations of the end results. The second pilot test, built on top of what is mainly explained in the first pilot test[8], adds a little bit more complexity to the simulation.

In the second pilot test, we focus on creating an approach that intends to bring the VE simulation closer to a real-life case scenario by randomizing the movements for the agents based on resource availability. That is, we clearly differentiate which patch an agent can move to by taking into account the maximum number of agents that can cohabit a patch at once. This particular denotation, resource availability, is the core principle used to determine whether or not a waterbird should move to another habitat to look for food since in real life the amount of waterbirds is directly proportional to food consumption. Obviously, other environmental aspects such as the nature of the habitats are not being considered yet.

[8]If not read yet, it is highly recommended to read the First Pilot Test part in order to grasp the whole idea of the second pilot test. Both are loosely coupled.

Another important point to mention in this test is that it is related to the flowchart shown in Figure 1, excluding the *Sensitivity Analysis* step. Likewise, the test follows the general algorithm discussed in previous sections, with the exception of the *Update* process. Consequently, this alters the implementation as well as the scripting.

```python
# omit scripts for library import, patch and agent creation

# START: simplest approach for synchronous updates based on lack of resources
def update():
    global patches, agents # make previously created patches and agents available
    sh_patches, lg_pathes = patches[0:2], patches[2:4] # distribute patches for short and long legs
    ag = agents[np.random.randint(len(agents))] # randomly choose an agent to update its status

    # simulating random movements based on agent's type
    if ag.type == 'short-legged':
        _x, _y = gen_random_point(sh_patches)
        # agent is moving within the same area
        if is_in_patch(sh_patches[1], (ag.x, ag.y)): # resourceless patch
            ag.x, ag.y = _x, _y # this agent belongs to the small patch, therefore he can move anywhere
        else: # agent coming from a long-legged patch
            if is_in_patch(sh_patches[0], (_x, _y)): # moving within the same is fine
                ag.x, ag.y = _x, _y
            else:# moving to small-legged patch requires resource availability checks
                pos = [(ag.x, ag.y) for ag in agents if ag.type == 'short-legged']
                count = count_points(sh_patches[1], pos)
                if count < 5: # maximum capacity for short-legged waterbirds
                    ag.x, ag.y = _x, _y
    else:
        _x, _y = gen_random_point(lg_pathes)
        # agent is moving within the same area
        if is_in_patch(lg_pathes[0], (ag.x, ag.y)): # resourceless patch
            ag.x, ag.y = _x, _y # this agent belongs to the small patch, therefore he can move anywhere
        else: # agent coming from long patch
            if is_in_patch(lg_pathes[1], (_x, _y)): # moving within the same is fine
                ag.x, ag.y = _x, _y
            else: # moving to small patch requires resource availability checks
                pos = [(ag.x, ag.y) for ag in agents if ag.type == 'long-legged']
                count = count_points(lg_pathes[0], pos)
                if count < 7: # maximum capacity for long-legged waterbirds
                    ag.x, ag.y = _x, _y
    # END: update
```

Listing 3: Script for updating an agent's position randomly

The script in Listing 3 is not considered as the best approach to reflect the second pilot test scenario because it violates most of the programming standards mentioned in previous sections. Although the *update* function does what it is supposed to do, it is implemented in a hard-to-follow, not-so-well-structured way. Note the *magic* values in lines 6, 13, 16, 26, 29, and 33 used to locate the array's positions (a particular patch). Note also the patches' maximum capacity values in lines 21 and 34. For this reason, we only intend to explain how to interpret not the implemented code, but the visualizations (see Figure 6).

Important: *This code implementation may look tedious in the first place but no refactoring remains necessary since it is just a step that leads to our final goal. Keep in mind that the definition of the helper functions* `gen_random_point()` *and* `count_points()` *can be found in Appendix A.*

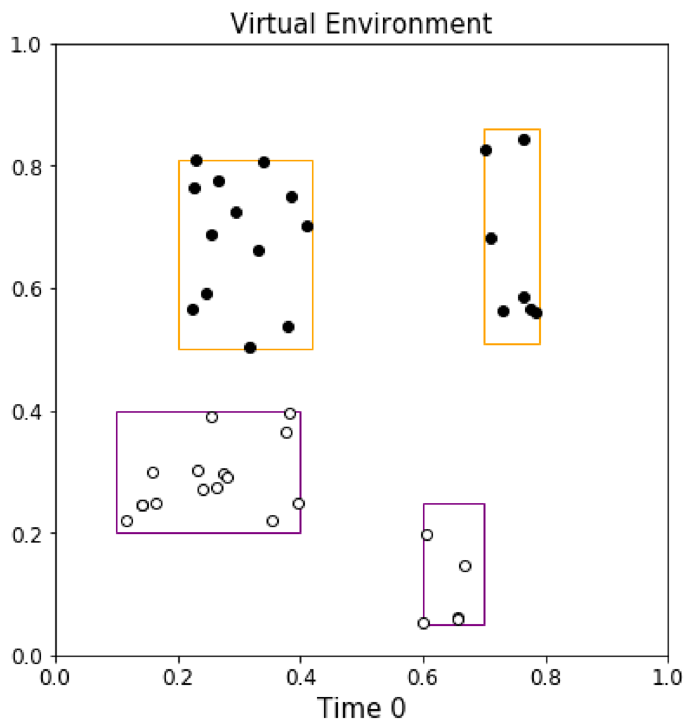
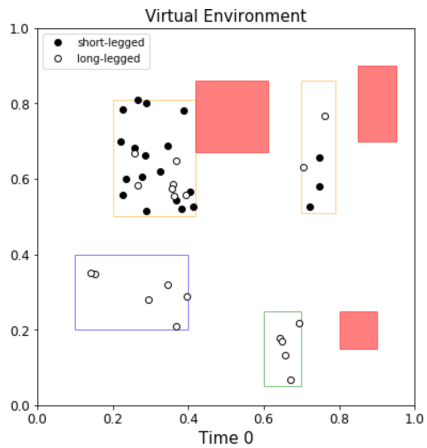


Figure 6: Preview of the second pilot test results: the yellow rectangles (patches) are the areas for long-legged waterbirds (agent) and the purple rectangles, the areas for the short-legged waterbirds. A long-legged agent can only travel between the yellow patches and a short-legged agent, between the purple patches. The initial conditions for the small rectangles (yellow and purple) are set according to the maximum capacity. Once the small patches reach the maximal capacity, an allowed agent can no longer travel across them unless the corresponding number of agent of the patch in question reduces over time.

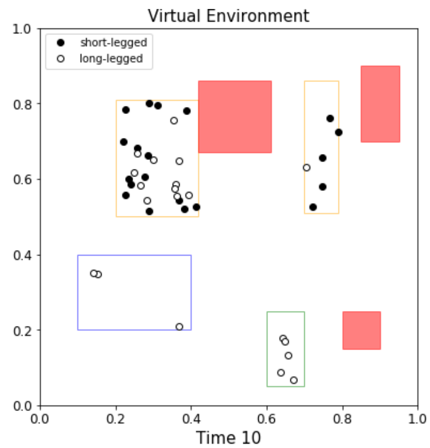
5.3 The VE Prototype

This part of the document discusses the current release of the ABM project, which is considered as the first prototype of the virtual environment simulation. The reason to state that is because this version goes beyond the first 2 pilot tests and covers important features of the VE system. It includes a higher degree of complexity, which brings the VE simulation much closer to the lifestyle of the waterbirds inhabiting the coastal lagoons of the tropics.

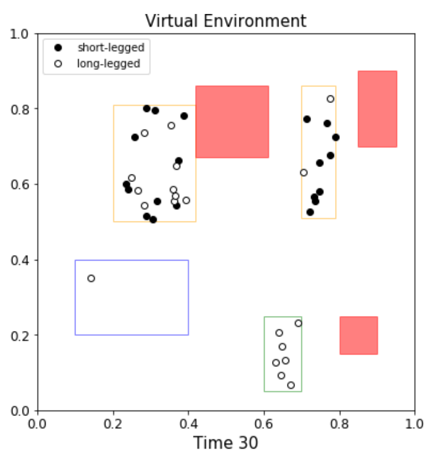
Again, this prototype relies on the terms and concepts clarified in the early tests. Besides the other tests, this version follows all the programming principles, the algorithms, the workflow scheme, previously discussed in this document. It also uses the predefined probability distribution functions (PDF) for the environmental characteristics that play an important role on the waterbirds' behaviour within the habitats and their interactions with each other.



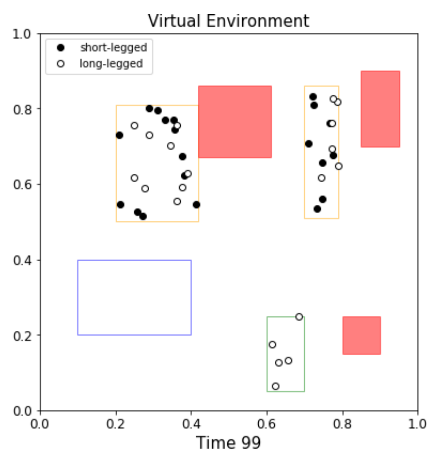
(a) Sample 1



(b) Sample 2



(c) Sample 3



(d) Sample 4

Figure 7: Preview of the VE simulation results: this visualization shows an extract of 4 samples out of 100 "processing random movements" units based on the PDFs. *Sample 1* and *Sample 4* are respectively the starting and ending point of the entire process. In one processing unit, only one agent gets to move, if the conditions are fulfilled to do so.

5.3.1 General comments

Unlike the other tests, the VE simulation setting of this prototype contains some updates, most importantly in the design. Firstly, there are more defined habitats: (edge-colored) `yellow, blue, green`, and (fully-colored) `pink` with some distinctive environmental characteristics each.

- `yellow`: categorized as *one*, these habitats represent the common places where the short-legged wading waterbirds can only wander for food.
- `blue`: categorized as *two*, this habitat contains certain properties that benefit only the long-legged waterbirds. For instance, the water depth of this habitat can be within one meter range.
- `green`: categorized as *three*, this habitat, besides its geometry (smaller in size), contains similar characteristics that restrain the short-legged birds from visiting it.
- `pink`: categorized as *human*, these habitats represent the human settlements. This factor also influences the waterbirds' behaviour, given the distance between a human settlement and a particular appealing, resourceful habitat.

Important: *The habitats' category names* `one, two, three`*, and* `human` *are just arbitrary names. Note that they could be changed in the future, if judged necessary.*

Next, the white dots and black dots are now labelled respectively as the long-legged waterbirds and short-legged waterbirds. Finally, the waterbird movements are driven by the category type of the habitat. For instance, the long-legged waterbirds can move to any habitat except for the human settlements whereas the short-legged waterbirds can only move to the *one* (`yellow`) habitats.

5.3.2 Analysis of the results

The results shown in Figure 7 are a single-run[9] of the scripts in Appendix A. We only extract 4 samples out of the 100 generated plots to explain the waterbirds' behaviour and interactions under a specific initial condition (see Table 3). And this initial condition can only generate a single-point dataset for each PDF over time. In order to achieve a data set, we need to collect a set of initial conditions, especially for the habitats' characteristics and tune their values over time. For instance, simulating a water depth reduction or tweaking the food availability factors are a good example of how to achieve a multi-point dataset for each PDF. But since this is out of the scope of this release, we only mention it as our next considerations for future releases.

The first sample, *Time 0*, is the first generated plot with a random distribution of the waterbirds within the allowed areas. Note that the long-legged birds (white dots) are spread out in all 4 habitats

[9] Running the script yourself will not achieve the same results due the random initialization state.

Initial Conditions	
Total of long-legged waterbirds	20
Total of short-legged waterbirds	20
Processing times	100
Threshold for	$1 * e^{-7}$
Areas for short-legged waterbirds	`one`
Areas for long-legged waterbirds	`one, two, three`
Habitat one (big)	s = 80psu, w = 5cm, f = 0.3
Habitat one (small)	s = 80psu, w = 5cm, f = 2.56
Habitat two	s = 10psu, w = 5cm, f = 6.41
Habitat three	s = 25psu, w = 40cm, f = 11.53

Table 3: Default values and parameters for the VE prototype's initial conditions. Note that the geometrical measurements of the habitats are part of the initialization process, but omitted here (refer to Appendix A for more details on them).

whereas the short-legged birds (black dots) are only located in the habitats categorized as *one*. Then in the second sample, *Time 10*, we notice that the long-legged birds start leaving Habitat *two*. And finally, in the last 2 samples, Habitat *two* remains empty. The reason for this is that the movements of the long-legged birds in this particular case are reasonably driven by the PDFs and the chosen threshold. That is, the chosen threshold factor does not have a fair probability (50% each) of whether a bird should move or not.

6 Conclusion

This document reports the analysis and results of the Agent-Based Model when studying habitat use by waterbirds in coastal lagoons of the tropics. We have built a virtual environment that is computationally inexpensive by simply considering a few designing aspects to simulate the ABM under certain conditions. So far, the realization of this virtual environment is very promising and brings us closer to the real-life situation of the waterbirds' behaviour.

The current version of the virtual environment is a prototype that has room for a lot of improvements. All along the document, some of these improvements are mentioned as well as some important points that are expected to be tackled in future releases.

References

[1] Agostino Merico. *Complex System and Agent-Based Modelling*. Jacobs University Bremen, Nov. 2018. (accessed: 29.05.2019).

[2] Davi Castro Tavares et al. "Environmental and anthropogenic factors structuring waterbird habitats of tropical coastal lagoons: implications for management." In: *Biological Conservation* 186 (2015), pp. 12–21.

[3] YAA NTIAMOA-BAIDU et al. "Water depth selection, daily feeding routines and diets of waterbirds in coastal lagoons in Ghana." In: *Ibis* 140.1 (1998), pp. 89–103.

[4] Mariano Paracuellos and José L Tellería. "Factors affecting the distribution of a waterbird community: the role of habitat configuration and bird abundance." In: *Waterbirds* (2004), pp. 446–453.

[5] Davi Castro Tavares and Salvatore Siciliano. "An inventory of wetland non-passerine birds along a southeastern Brazilian coastal area." In: *Journal of Threatened Taxa* 5.11 (2013), pp. 4586–4597.

[6] Jacques Blondel. "Guilds or functional groups: does it matter?" In: *Oikos* 100.2 (2003), pp. 223–231.

[7] Davi Castro Tavares and Salvatore Siciliano. "Temporal variation in the abundance of waterbird species in a coastal lagoon in the northern Rio de Janeiro state." In: *Biotemas* 27.1 (2014), pp. 121–132.

[8] Microsoft Corporation. *Visual Studio Code*. Apr. 2019. URL: https://code.visualstudio.com/.

[9] GitHub Inc. *The world's leading software development platform*. Apr. 2019. URL: https://github.com/.

[10] Python Software Foundation. *Welcome to Python.org*. Mar. 2019. URL: https://www.python.org/.

[11] Project Jupyter. *Project Jupyter*. Jan. 2019. URL: https://jupyter.org/. (Last updated April 12, 2019).

[12] Smashing Magazine. *The Nine Principles Of Design Implementation*. Aug. 2017. URL: https://www.smashingmagazine.com/2017/08/nine-principles-design-implementation/.

[13] The SciPy community. *numpy.random.rand*. Jan. 2019. URL: https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.rand.html.

[14] The SciPy community. *numpy.linalg.norm*. Jan. 2019. URL: https://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.norm.html.

[15] The Matplotlib development team. *matplotlib.patches.Patch*. Mar. 2019. URL: https://matplotlib.org/3.1.0/api/_as_gen/matplotlib.patches.Patch.html.

[16] The Matplotlib development team. *matplotlib.path*. Mar. 2019. URL: https://matplotlib.org/3.1.0/api/path_api.html.

[17] The Matplotlib development team. *matplotlib.pyplot.plot*. Mar. 2019. URL: https://matplotlib.org/api/_as_gen/matplotlib.pyplot.plot.html.

[18] The Python community. *imageio*. Jan. 2019. URL: https://imageio.readthedocs.io/en/latest/userapi.html.

[19] Aris Papadopoulos. *Why should developers use third-party libraries?* Mar. 2018. URL: https://www.smashingmagazine.com/2017/08/nine-principles-design-implementation/.

Appendix A Code Repository

All the code implemented and utilized during the execution of the data workflow described in this report is available on the GitHub repository https://github.com/systemsecologygroup/BirdsABM.