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

1 Introduction

2 Theoretical Background

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[1] (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.

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

Tools & Software			
	Versions	**Sources**	**Cost**
Visual Studio Code	1.34.0	See [1]	Free
Git	2.7.4	Built-in Linux program	Free
GitHub	N/A	See [2]	5 free users
Python	3.5	See [3]	Free
Jupyter Notebook	5.7.4	See [4]	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 is simpler and/or easier, or the VE surprisingly performs better[2]. 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]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.

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.

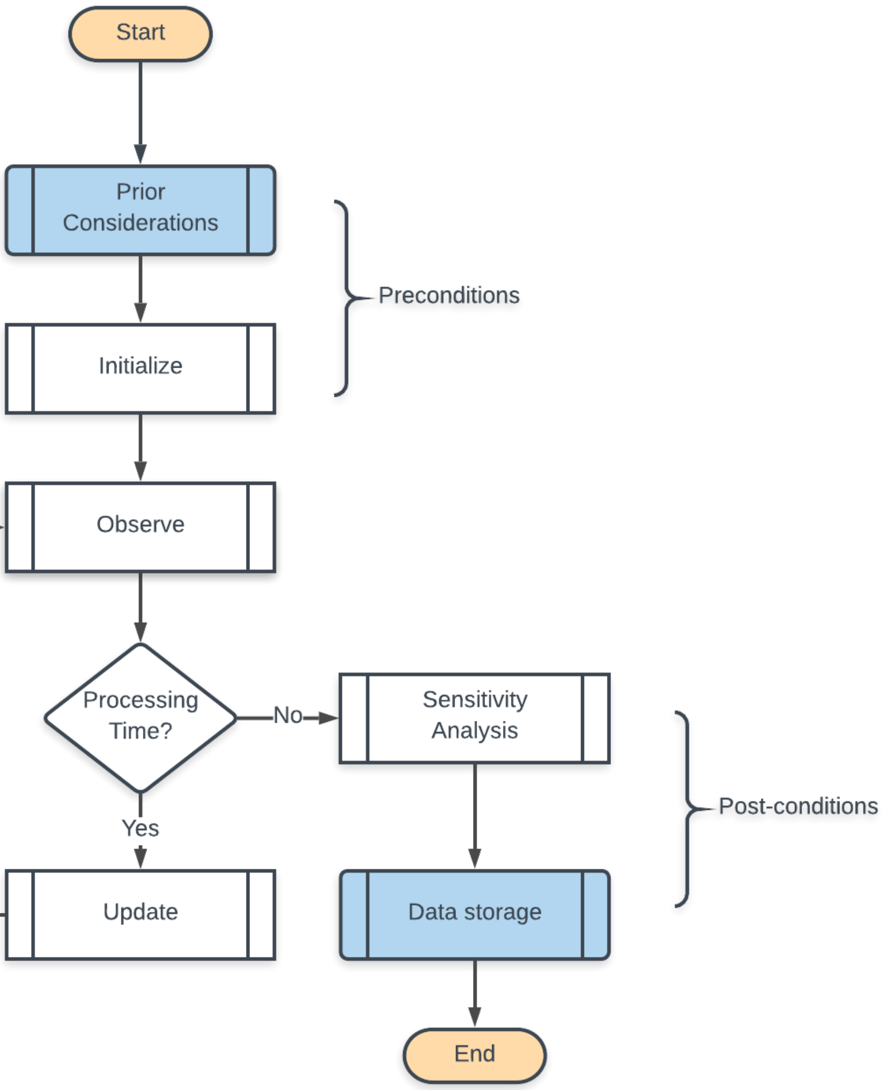


Figure 1: Workflow diagram
(credits: made with *Lucidchart*)

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[3]. 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 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.

[3]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.

- ***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*[4], and the waterbirds, by the term *Agent*. In this case, the 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.

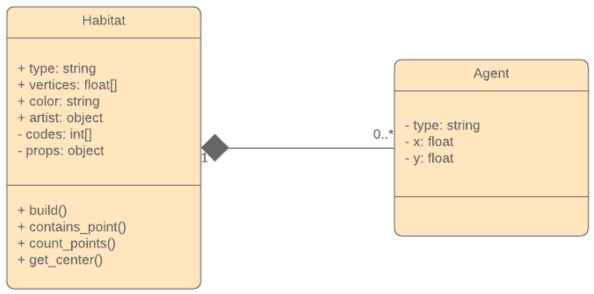


Figure 2: Data structure of *Habitat* and *Agent*
(credits: made with *Lucidchart*)

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

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

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 the Appendix section.*

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[5] 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*[5]. 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);
- ***structured***: the script is written semantically and logically, and organized sequentially (3rd-party libraries import, constants declaration, functions definition, and *main*[6]);
- ***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.

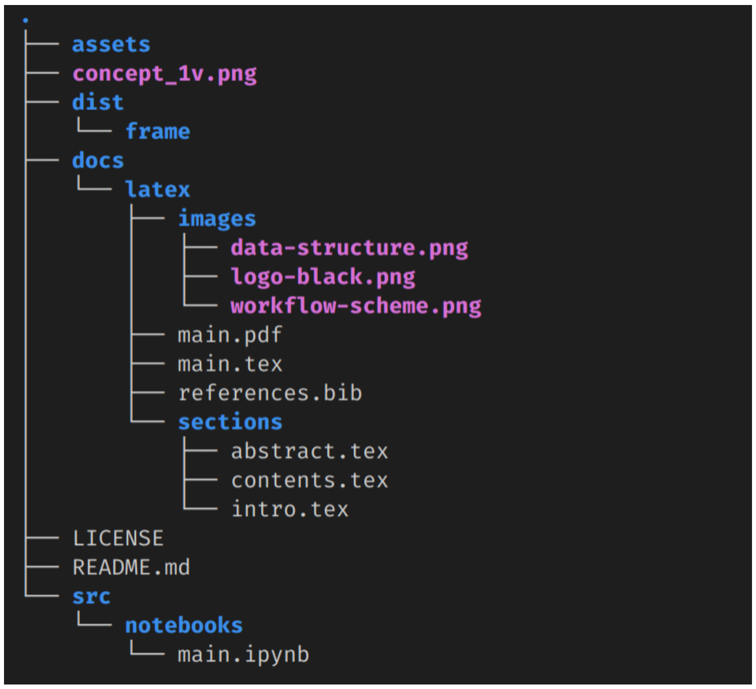
Besides the coding procedure, we also created a commonly-standardized, organized file structure (See Figure 3) for the project. Note the parent folders named *src* and *docs*. The former is for the source code of the project and the latter, for the documentation. The contents under those folders are 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

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

[6]Main entry function to run an application

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.



```
.
├── 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

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 [12]. 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.

Third-Party Libraries			
	Versions	**Sources**	**Features**
numpy	1.15.4	See [6, 7]	*random, linalg.norm*
matplotlib	3.0.2	See [8, 9, 10]	*patches, path, pyplot*
imageio	2.4.1	See [11]	*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. The results of these tests are presented in the subsections that follow.

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 4). 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 [8]). 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 4.

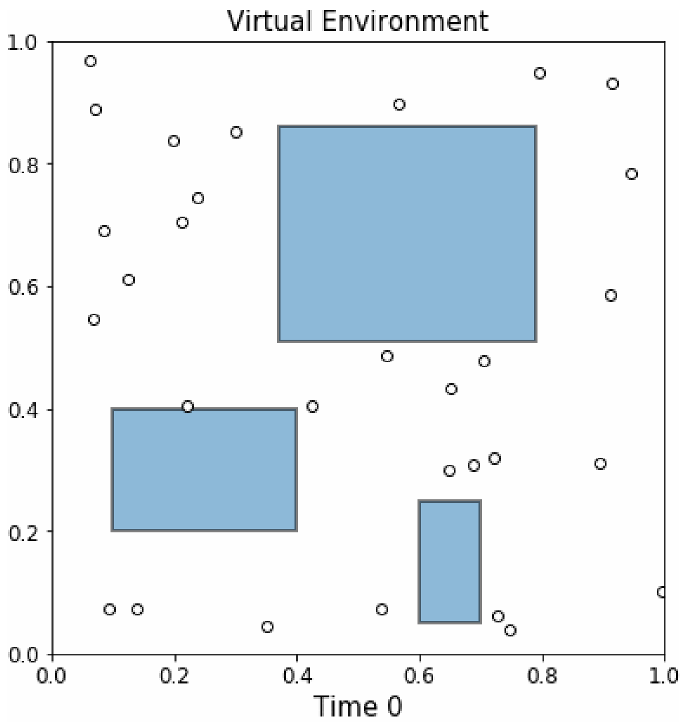


Figure 4: Preview of the first pilot test results: there is a total of 3 static patches (of different sizes) and 30 randomly-positioned short-legged agents. This visualization is the first generated plot of 20. The entire creation process is repeated 20 times and a final GIF image is built up out of 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[7], 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.

[7]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 5).

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 the Appendix section.*

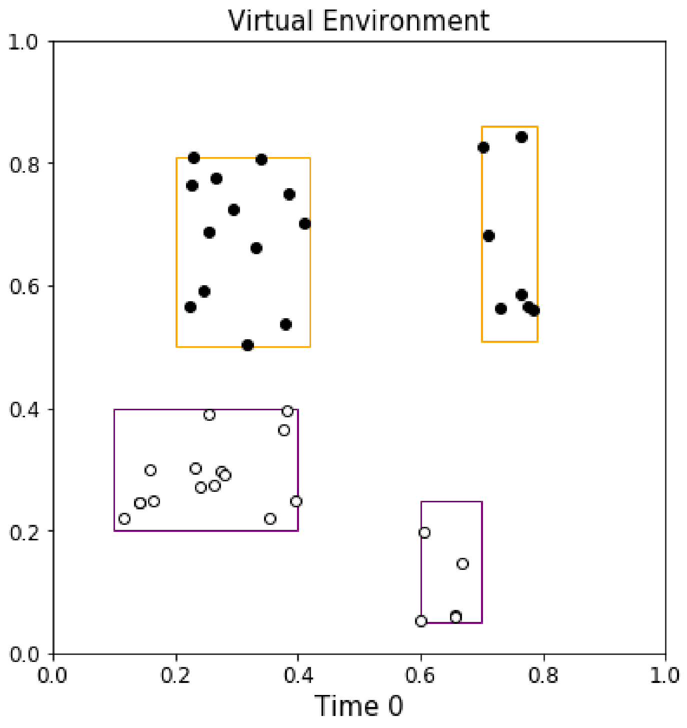


Figure 5: Preview of the second pilot test results: the yellow rectangles (patches) are the areas for long-legged waterbirds (agent) and the purple rectangles, the areas for the short-legged waterbirds. A long-legged agent can only travel between the yellow patches and a short-legged agent, between the 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

6 Conclusion

References

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

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

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

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

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

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

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

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

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

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

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

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