PyLogo:
A Python reimplementation of (much of) NetLogo

Russ Abbott and Jung Soo Lim
Department of Computer Science, California State University, Los Angeles,
5151 State University Drive, Los Angeles, California, USA
{*rabbott, jlim34*}*@exchange.calstatela.edu*

Keywords: ABM, agent-based modeling, NetLogo, PyLogo, Python, simulation

Abstract: IN the world of Agent-Based Modeling (ABM) NetLogo reigns as the most widely used platform. The NetLogo world of agents interacting in a two-dimensional space seems to provide just the right level of simplicity and abstraction for a wide range of models. Yet, the NetLogo language makes model development more painful than necessary.

This combination—widespread popularity accompanied by unnecessary coding pain—motivated the development of PyLogo, a NetLogo-like modeling and simulation environment in which developers write their models in Python. Although other NetLogo-like systems exist, as far as we know PyLogo is the only NetLogo-like system in Python at this level of completeness.

This paper examines a number of issues with the NetLogo language and then offers a simple, illustrative PyLogo example model.

PyLogo is also a tribute to Python. Work began after the Fall 2019 semester ended. A fully operational version was ready a month later when the Spring 2020 semester began. The models used in class were all written in what might be considered academic "real time", i.e., as needed during the Spring 2020 semester. Students, about half of whom were new to Python, also wrote their own models.

PyLogo is open source. The code is available at this GitHub repository. We welcome collaborators.

1 Introduction

For a senior-level modeling and simulation class we developed a pure-Python version of NetLogo.

Why NetLogo?

- Due to its widespread adoption, NetLogo has become the standard platform for communicating and implementing ABMs (Thiele, 2014).
- NetLogo is recognized as robust and powerful but nevertheless easy to learn (Badham, 2015).

Why Python?

- The NetLogo language makes the development of non-trivial models unnecessarily painful.
- Python is one of the most straightforward programming languages. It is often used to introduce students to programming and yet has become the *de facto* scripting language for many areas of computer science (Nagpal and Gabrani, 2019).

Our objectives for this paper

- To document some of NetLogo's awkward features. (Section 3). (Given space limitations, this discussion constitutes the bulk of the paper.)
- To discuss a small PyLogo model. (Section 4).
- To offer PyLogo to the ABM community as an open-source resource and to invite others to contribute to PyLogo's further development.

2 Related Work

There are, of course, many modeling and simulation systems, both agent-based and non-agent-based. The following discusses only agent-based systems.

First the classics. *Repast* (North et al., 2013) and then *MASON* (Luke et al., 2005) established agent-based modeling as a distinct discipline. They remain among the most widely-used ABM systems, especially for complex models or models with many agents. These were followed by *AnyLogic* (Grigoryev, 2015). All use Java for model development.

More recent agent-based systems include:

- *ABCE* (Taghawi-Nejad et al., 2017), a Python-based platform tailored for economic phenomena;
- *Agents.jl* (Vahdati, 2019), a Julia-based relative newcomer; and
- *GAMA* (Taillandier et al., 2019), which describes itself as *an environment for spatially explicit agent-based simulations*. GAMA models are written in GAML, a scripting-like language.

NetLogo (Tisue and Wilensky, 2004a; Tisue and Wilensky, 2004b) teased the possibility of writing models in pseudo-English.

> It stands apart as the best platform for both beginners and serious scientific modelers. Many if not most public scientific ABMs are implemented in NetLogo, which has become a standard platform for agent-based simulation (Railsback et al., 2017).

NetLogo's popularity triggered work in multiple directions. For example, mechanisms exist for interacting with NetLogo from other languages.

- NetLogo Mathematica allows Mathematica to control NetLogo. (Bakshy and Wilensky, 2007).
- *NL4Py* (Gunaratne and Garibay, 2018) and *Pynetlogo* (Jaxa-Rozen and Kwakkel, 2018) offer access to and control over NetLogo from Python. A NetLogo extension allows calls to Python.
- *RNetLogo* (Thiele, 2014) enables one to write *R* code that calls NetLogo. A NetLogo extension allows one to call *R* from NetLogo.

Finally, a number of NetLogo-like systems allow model developers to write in a language other than the standard NetLogo script.

- *ReLogo* (Ozik et al., 2013), an offspring of *Repast*, replicates many NetLogo features. It uses *Groovy*, a script-like version of Java, as a model-development language.
- *Mesa* (Masad and Kazil, 2015) comes closest to PyLogo. Although Mesa is less complete, e.g., no links, Mesa models are written in Python. Mesa consists of a Python-based "headless" modeling component, which can run either autonomously or with a JavaScript visualization component.

NetLogo critiques. We found no earlier work critiquing NetLogo as a language.

3 The NetLogo language

This section discusses the NetLogo language and some of the issues that motivated PyLogo.

We wish to acknowledge (again) NetLogo's extraordinary success. Wilensky (Wilensky, 2020) reports that since January 2019 there have been:

- 700k unique visitors to the NetLogo website,
- 130k NetLogo downloads, and
- 7000 authors of papers that use NetLogo.
- 600 courses that use NetLogo.

Wilensky believes these are significant undercounts and would feel comfortable increasing these numbers by an order of magnitude.

3.1 A brief history of NetLogo

Tisue and Wilensky (Tisue and Wilensky, 2004a) and (Tisue and Wilensky, 2004b) discuss NetLogo's history.

> Logo, as originally developed by Seymour Papert and Wally Feurzeig (Feurzeig and Papert, 1967), is derived from Lisp but has a syntax that seems to non-programmers to be similar to English. It is best known for its "turtle graphics," in which a virtual being or "turtle" moves around the screen drawing figures by leaving a trail behind itself.
>
> NetLogo generalizes this concept to support hundreds or thousands of turtles all moving around and interacting. Different "breeds" of turtle may be defined, and different variables and behaviors can be associated with each breed. In effect, NetLogo adds a primitive object-oriented overlay to Logo.
>
> Turtles inhabit a grid of programmable "patches." Collectively, the turtles and patches are called "agents." Agents can interact with each other.
>
> A collection of agents—e.g., the set of all turtles or the set of all patches—is known as an *agentset*. One can make custom agentsets on the fly: for example the set of all red turtles or the set of patches with a given *X* coordinate. One can ask all agents in an agentset, to perform a series of operations.

NetLogo is specified in a clearly-written user manual (Wilensky, 2019).

Logo's debt to Lisp is clear from introductory articles, e.g., (Harvey, 1982), which focus on examples that use recursive list manipulation. Like Logo, one of NetLogo's goals was to make Lisp accessible to middle school students. Unfortunately (in our view) NetLogo's commitment to being "English-like" produced an imperative-style language. The remainder of this section discusses concerns about the language.

3.2 Keywords

NetLogo is somewhat ambiguous about keywords. The Programming Guide says,

> The only keywords are **globals, breed, turtles-own, patches-own, to, to-report,** and **end**, plus **extensions** and the experimental **__includes**.

But tacking *-own* onto a breed name makes the combination a keyword: if *cats* is a breed, **cats-own** functions like **turtles-own**.

NetLogo has a number of such "keyword-constructors." For example, breeds of *links* are declared using the "keywords" **undirected-link-breed** and **directed-link-breed**.

Furthermore, according to the **Keywords** section of the Programming Guide,

> Built-in primitive names may not be shadowed or redefined, so they are effectively a kind of keyword.

The NetLogo dictionary lists (very approximately) 500 built-in names. In most cases, this is less of a problem than one might imagine. Most of the built-in names are either constants (like *true*) or function names (like *sin*), which are not problematical.

More confusingly, control constructs such as *if*, *ifelse*, *ifelse-value*, and *while* appear as if they were keywords. According to the Programming Guide,

> Control structures such as *if* and *while* are special forms. You can't define your own special forms, so you can't define your own control structures.

Infix operators such as *of* and *with*, and identifiers such as *self* and *myself* also function like keywords.

It would appear that *and*, *or*, and other functions are also defined via special forms: *and* and *or* use short-circuit evaluation; *set* and *let* do not evaluate their first arguments; *filter* and *map* treat their first arguments as function names rather than function calls.

It adds confusion to deny keyword status to identifiers defined via such special forms.

Finally, *breed* is both a built-in *own* variable, which can be set to change an entity's breed, and a keyword used to declare breeds.

3.3 A primitive object-oriented capability

NetLogo offers a primitive form of object-oriented programming through its **breed** mechanism. If one thinks of NetLogo's **turtle** as similar to Python's *object*, a breed is something like a subclass of **turtle**. Just as **turtle**s may have the equivalent of instance variables—declared through the **own** keyword constructor—breeds may also have the equivalent of instance variables—again declared using **own**.

But breeds differ in important ways from classes in most object-oriented languages.

- one can't declare a sub-breed of a breed;
- breed methods are not marked as restricted to breed instances;
- even though patches are agents, one can't create a breed of patches; and
- when an entity's *breed* variable is changed, say from *breed-1* to *breed-2*, it loses its *breed-1* instance variables and gains those of *breed-2*.

3.4 Sets and lists

*Set*s and *list*s are treated as unrelated data structures. This raises a couple of issues.

- Why limit sets to agents? Just as one can have lists of both agents and numbers, one should be able to have sets of numbers as well as sets of agents.
- More importantly, given a NetLogo function defined for one, there is generally a similar *but different* function defined for the other—creating confusing redundancy. The following pairs of functions operate on sets and lists respectively.
  - ask and foreach
  - with and filter
  - of and map

  In each case, the two functions provide very similar functionality. Replacing each pair with a single function that operates on both sets and lists would simplify the language and reduce the memory burden on users. This is straightforward in Python since sets and lists are both types of collections.

Following are some simple illustrative examples.

3.4.1 ask and foreach

Consider a world of five turtles and two lists:

- a list of degrees, one for each turtle ordered by *who*-number, and
- a list of distances, also one for each turtle ordered by *who*-number.

We want our turtles to turn by the indicated number of degrees and then to move forward the indicated distance. This seems tailor-made for *foreach*.

```
( foreach sort-on [who] turtles
          [30 40 120 50 270]
          [40 20 50 10 30]
      [ [t deg dist] ->
        ask t [ rt deg fd dist ] ] )
```

This code seems overly complex. Since *turtles* is an agentset and *foreach* requires lists, we used

```
sort-on [who] turtles
```

to convert an agentset to an ordered list of turtles. Then, since we are requiring the turtles to perform turtle methods, we embeded an *ask* in the *foreach* anonymous procedure. Not very pretty code.

A Python version is much simpler. (We assume that *turtle(n)* retrieves the turtle with *who*-number *n*, that *rt()* and *fd()* are turtle methods, and that *count()* has been imported from *itertools*).)

```
for who, deg, dist in
            zip(count(),
                [30, 40, 120, 50, 270],
                [40, 20, 50, 10, 30] ):
  t = turtle(who)
  t.rt(deg)
  t.fd(dist)
```

3.4.2 with/filter, of/map, and list comprehensions

One of Python's most powerful and intuitive constructs is the list comprehension. Wikipedia lists nearly three dozen languages that include it. Because of its power and simplicity, Python style guides, e.g., GitHub Python Style Guide, generally recommend list comprehensions over *map* and *filter*.

Yet NetLogo does not offer list comprehensions. It is possible, if somewhat ugly, to create a simple NetLogo equivalent using *of* and *with*. For example, assuming *t.who* retrieves turtle *t*'s *who*-number, one can mimic Python's

```
[t.who * t.who for t in turtles
               if t.who % 2 == 1]
```

with

```
[who * who] of turtles
               with [who mod 2 = 1]
```

Although this is possible, a general list comprehension would be welcome. More importantly, the Python list comprehension is very widely used and well-known to virtually everyone familiar with Python. Few NetLogo developers would easily come up with the NetLogo equivalent.

3.5 Reporters

NetLogo uses the term *reporter* in multiple—and often confusing—ways.

- **Primitive reporters**, such as *turtle* and *list*.
- **User-defined procedures** created with *to-report*.
- ***ask*-reporters**. Many NetLogo primitives—such as *all? max-n-of* (and similar), *of*, *sort-on*, *while* (strangely), and *with* (and similar)—provide *ask*-like contexts for reporters. By that we mean that these reporters run in the same sort of context in which *ask* command blocks run, i.e., as something like "methods" of the agents running them. In the example at the end of the previous section, the *who* in the reporter *[who * who]* referred to the *who*-number of the then-active turtle.
- **anonymous reporters**. These may or may not work as one would expect.

Consider the following example. (The parentheses are required for correct parsing.)

```
turtle ([who] of a-turtle + 1)
```

Assuming *a-turtle* refers to a turtle, this *reports* the turtle with the successor *who*-number.

One can define a reporter, named, say, *next-turtle*, to perform this function.

```
to-report next-turtle [a-turtle]
  report turtle ([who] of a-turtle + 1)
end
```

When applied,

```
next-turtle some-turtle
```

produces the correct answer. But attempting

```
[next-turtle] of some-turtle
```

produces the error message: *NEXT-TURTLE expected 1 input.* (See Section 3.8 for why.) Furthermore, one may not wrap *next-turtle* in an anonymous reporter.

> Agent primitives such as *of* and *with* don't accept anonymous procedures.

Making a possibly long story short, the following are all equivalent.

```
0. next-turtle some-turtle   ;; as above
1. [next-turtle self] of some-turtle
2. first map next-turtle
             (list some-turtle)
3. (runresult
     [t -> next-turtle t] some-turtle)
```

Let's consider 1 - 3 in order.

1. This illustrates why we call the reporter associated with *of* an *ask*-reporter. That's how *self* gets *some-turtle* as its value.
2. This illustrates how *map* is special. Although its first argument plays the same role as the *of* reporter, the two reporters take different forms: *[next-turtle self]* vs *next-turtle*.

3. *runresult* is required because it is not permissible to apply anonymous procedures to their arguments in the same way that one applies standard procedures to their arguments. That's the case even if the anonymous procedure is given a name.

```
[n -> n + n] k
```

and

```
to-report test-named-anon [k]
  let double [ n -> n + n ]
  report double k
end
```

both generate the (unhelpful) error message: `Expected command.` Also see Section 3.8.

3.6 The *if* and *while* constructs require different condition forms

The *if* family of statements requires a boolean expression as condition; *while* requires a reporter block.

```
ifelse 3 > 4 [show 3] [show 4]
```

parses and produces 4.

```
while [3 > 4] [show 3]
```

parses and correctly produces nothing. Why require the user to remember when each form is required?

3.7 Non-standard terminology

NetLogo uses the terms *reporter* and *report* for concepts for which virtually all other languages use *function* and *return*.

The functions *word* and *sentence* are also non-standard and confusing.

- The function *word* converts its argument(s) to strings, which it concatenates.
- The function *sentence* combines the functionality of *list* and what might traditionally be called *flatten*. Confusingly, *sentence* has no specific connection to words or strings.

3.8 Higher-order functions

Since almost the very beginning, NetLogo has included the standard trio of higher-order functions: *filter*, *map*, and *reduce*. However, it appears to be quite awkward for model developers to create and use their own higher order functions.

Consider an attempted definition of the trivial function *application-of* that expects a reporter procedure as its first argument and an argument for that reporter procedure as its second. *application-of* applies its first argument to its second and returns the result. (One might want something like this if one has a list of functions and wants to select one to be applied to some element.)

```
to-report application-of [fn arg]
  report fn arg
end
```

The preceding fails to compile. But as we did in Section 3.5, we can use the *map* trick.

```
to-report application-of [fn arg]
  report first map fn (list arg)
end
```

But attempted execution

```
application-of last a-list
```

produces the error message: *APPLICATION-OF expected 2 inputs*. That's the case even though the body of the function executes without complaint.

```
first map last (list [1 2 3])   ;; => 3
```

The problem is that it's not possible to refer by name to a function as an object—except as the first argument of *map* and other special cases.

Making the parentheses explicit, NetLogo parses

```
application-of last a-list
```

as

```
application-of(last(a-list))
```

So (almost) any time a procedure name appears, the NetLogo parser takes it as a procedure call.

One work-around is to use anonymous procedures. Suppose we define *double* and *square*.

```
to-report double [n]
  report n + n
end

to-report square [n]
  report n * n
end
```

For a list containing those function, write

```
list [n -> double n] [n -> square n]
```

rather than

```
list double square
```

We could test this as follows.

```
to test-application-of [k]
  let lst list [ n -> double n ]
               [ n -> square n ]
  foreach lst [fn ->
                show application-of fn k]
end

test-application-of 6   ;; => 12 36
```

3.9 Identifiers and shadowing

In the following, NetLogo's shadowing rules disallow the parameters *x* (which is global) and *dx* (which is a primitive) and the local variables *y* (which is global) and *z* (which is a parameter name).

```
globals [x y]

to test-scope [x dx z]
  let y 3
  let z 4
  ...
end
```

3.10 Square brackets and parentheses

Square brackets are required for the following.

- command block components of higher-order (and control) constructs such as *ask*, *carefully*, *crt*, *cro*, *hatch*, *sprout*, *if*, *loop*, *repeat*, *while*;
- reporter components of functions such as *all?*, *every*, *max-n-of*, *of*, *sort-on*, *with*;
- anonymous expressions, which *must* be surrounded by square brackets;
- "containers" for components of keyword constructs such as *at-points* (two levels), *breed*, *extensions*, *globals*, *histogram*, *_includes*, *_-own*;
- *lists* (but only of literals) in the code or of any values in the output. The list `[1 2]` is ok, but the would-be list `[(turtle 0) (turtle 1)]` is not.
- parameter lists;

That's too much of a burden for square brackets to carry comfortably.

Parentheses are required around any construct that includes a function (such as *foreach*, *ifelse*, *ifelse-value*, *list*, *map*, *patch-set*, *run*, *runresult*, *sentence*, *turtle-set*, and *word*) that (a) may take a variable number of arguments and (b) is given two or more.

Parentheses are not required if only one argument appears—except for *list* which requires parentheses for one argument but not for two.

3.11 Stack trace for run-time errors

NetLogo is inconsistent with respect to run-time errors—sometimes displaying a stack trace and at other times only an error notice, with no hint about how the program got there. If a stack trace can be provided in some situations, why not in all?

3.12 Dictionaries

Dictionaries are one of Python's most useful features. NetLogo's *table* extension functions like a Python dictionary. But *table* operations must include the prefix `table::`, which makes for cluttered code.

3.13 Dot notation for accessing instance variables and methods

Most object-oriented languages use dot-notation for instance variables and methods. NetLogo foregoes dot notation: `[who] of turtle-x` rather than `turtle-x.who`. Dot notation is simpler and cleaner.

4 Starburst: a PyLogo model

This section examines a simple PyLogo model.

Starburst starts with a user-selected number of agents at the center of the grid. When run, 20% of the agents move toward each corner. The final 20% remain stationary. Since the agents in each group have identical coordinates, each group appears to be a single agent. The agents reflect back from the corners. At a user-selected tick, the agents start moving in random directions, producing a "starburst" effect. From then on the agents experience a repulsive force from each other, producing attractive swerving trajectories.

Figure 1 is a screenshot. The bottom two rows of buttons are standard on all PyLogo models. The penultimate line includes *setup*, *go once*, and *go* buttons, similar to NetLogo models. The *go* button, initially green, turns red and is renamed *stop* when clicked. The *exit* button terminates the model. The top five lines allow the user to set model parameters.

Starburst consists of a *Starburst_Agent* subclass of the PyLogo *Agent* class and a *Starburst_World* subclass of the PyLogo *World* class. (See Listing 1.)

- *setup()* consults the gui to determine the number of agents. It creates those agents and sets their velocities as described earlier. PyLogo, which provides substantial agent-motion support, includes *velocity* as an agent attribute.
- *step()* executes at each tick. If the number of ticks has exceeded the user-settable "burst tick,"

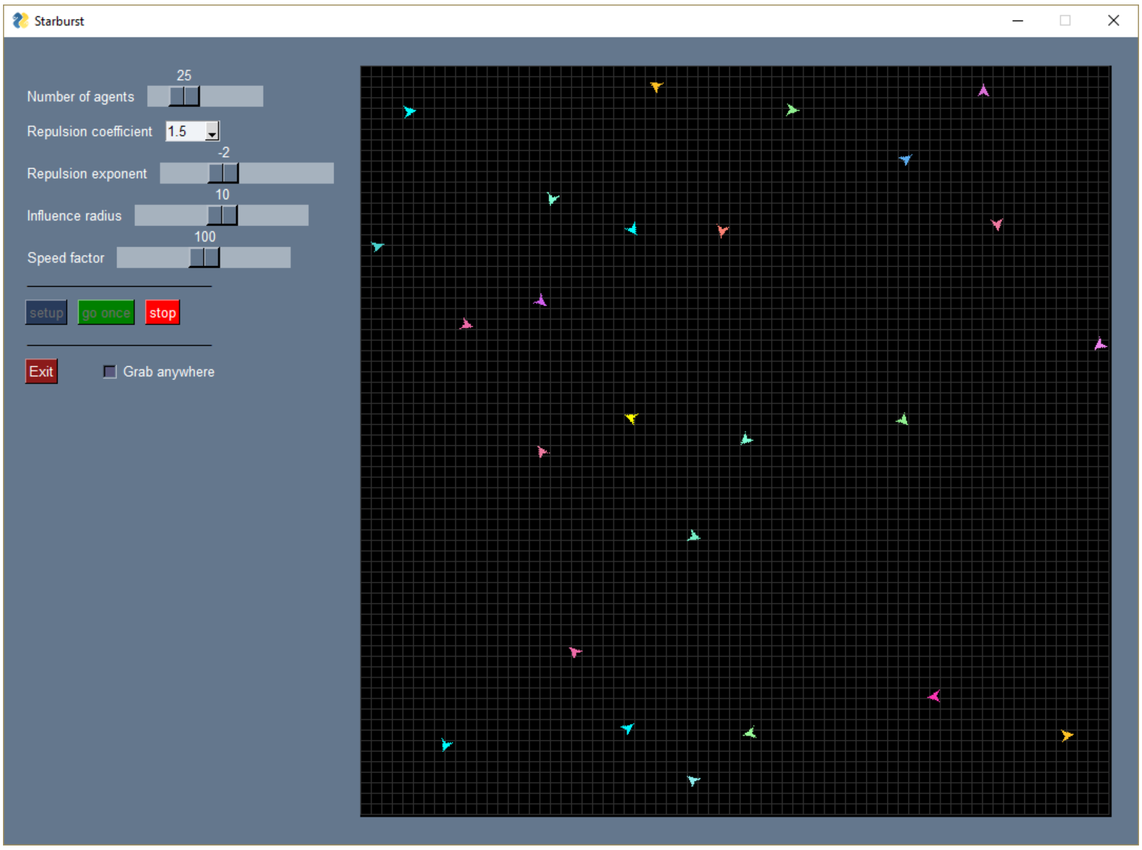

Figure 1: Starburst

(not displayed), the *static* method *Agent.update_-agent_velocities()* is called. That, in turn, calls *update_velocity()* for each *Starburst_Agent*.

update_velocity() updates an agent's velocity as a function of its distances from its neighbors. The *influence radius* slider determines how close agents must be to effect each other.

Agent.forces_cache stores the inter-agent forces so that they are computed only once each tick.

Agent.update_agent_positions() updates agent positions based on their positions and velocities.

5 Conclusion

PyLogo has two major weaknesses.

- PyLogo is not optimized. It runs more slowly than NetLogo, and it can gracefully handle only a smaller number of agents. Even so, it handles the standard NetLogo examples quite well.
- PyLogo doesn't offer graphing.

We welcome the participation of anyone knowledgeable about Python optimization or visualization.

Notwithstanding these limitations, PyLogo was more than adequate for teaching an ABM course.

PyLogo's primary advantage is that model developers write in Python. For people accustomed to writing software, Writing in Python is much less frustrating than writing in NetLogo. Experienced programmers often feel that they are fighting the language when writing in NetLogo. The opposite is generally true when writing in Python.

From a broader perspective, PyLogo corroborates the NetLogo model for agent-based modeling. Something as simple and intuitive as agents interacting on a grid (using tick-based scheduling) accommodates a very wide range of models.

The development of PyLogo—a fully operational core completed in a month, with additional features plus a range of models added while using the system for a class—demonstrates that Python enables rapid development of a fairly sophisticated system.

PyLogo is comparatively small: 10 core files and 15 models of 2,000 SLOC and 3,000 SLOC respectively. It is available for download.

```
# imports not shown ...

class Starburst_Agent(Agent):

    def update_velocity(self):
        velocity = self.velocity
        influence_radius = gui_get('Influence radius')
        neighbors = self.agents_in_radius(influence_radius * BLOCK_SPACING())
        for neighbor in neighbors:
            force = Agent.forces_cache.get((neighbor, self), None)
            if force is None:
                force = force_as_dxdy(self.center_pixel, neighbor.center_pixel)
                Agent.forces_cache[(neighbor, self)] = force * (-1)
            velocity = velocity + force
        speed_factor = gui_get('Speed factor')
        self.set_velocity(normalize_dxdy(velocity, 1.5*speed_factor/100))


class Starburst_World(World):

    def setup(self):
        nbr_agents = gui_get('nbr_agents')
        for _ in range(nbr_agents):
            self.agent_class(scale=1)
        vs = [Velocity((-1, -1)), Velocity((-1, 1)), Velocity((0, 0)),
              Velocity((1, -1)), Velocity((1, 1))]
        for (agent, vel) in zip(World.agents, cycle(vs)):
            agent.set_velocity(vel)

    def step(self):
        burst_tick = gui_get('Burst tick')
        if World.ticks >= burst_tick:
            Agent.update_agent_velocities()

        Agent.update_agent_positions()


# PySimpleGUI definitions. PyLogo uses the very straightforward
# PySimpleGui (https://pysimplegui.readthedocs.io/) as its GUI framework.

# As in many Python programs, the following starts the model.
if __name__ == "__main__":
    PyLogo(Starburst_World, 'Starburst', gui_left_upper, agent_class=Starburst_Agent,
           bounce=(True, False), patch_size=9, board_rows_cols=(71, 71))
```

REFERENCES

Badham, J. (2015). An introduction to agent-based modeling (review).

Bakshy, E. and Wilensky, U. (2007). Netlogo-mathematica link. *Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL*.

Feurzeig, W. and Papert, S. (1967). The logo programming language. *ODP-Open Directory Project*.

Grigoryev, I. (2015). Anylogic 7 in three days. *A quick course in simulation modeling*, 2.

Gunaratne, C. and Garibay, I. (2018). Nl4py: Agent-based modeling in python with parallelizable netlogo workspaces. *arXiv preprint arXiv:1808.03292*.

Harvey, B. (1982). Why logo. *Byte*, 7(8):163–193.

Jaxa-Rozen, M. and Kwakkel, J. H. (2018). Pynetlogo: Linking netlogo with python. *Journal of Artificial Societies and Social Simulation*, 21(2).

Luke, S., Cioffi-Revilla, C., Panait, L., Sullivan, K., and Balan, G. (2005). Mason: A multiagent simulation environment. *Simulation*, 81(7):517–527.

Masad, D. and Kazil, J. (2015). Mesa: an agent-based modeling framework. In *14th PYTHON in Science Conference*, pages 53–60.

Nagpal, A. and Gabrani, G. (2019). Python for data analytics, scientific and technical applications. In *2019 Amity International Conference on Artificial Intelligence (AICAI)*, pages 140–145. IEEE.

North, M. J., Collier, N. T., Ozik, J., Tatara, E. R., Macal,

C. M., Bragen, M., and Sydelko, P. (2013). Complex adaptive systems modeling with repast simphony. *Complex adaptive systems modeling*, 1(1):3.

Ozik, J., Collier, N. T., Murphy, J. T., and North, M. J. (2013). The relogo agent-based modeling language. In *2013 Winter Simulations Conference (WSC)*, pages 1560–1568. IEEE.

Railsback, S., Ayllón, D., Berger, U., Grimm, V., Lytinen, S., Sheppard, C., and Thiele, J. (2017). Improving execution speed of models implemented in netlogo. *Journal of Artificial Societies and Social Simulation*, 20(1).

Taghawi-Nejad, D., Tanin, R. H., Chanona, R. M. D. R., Carro, A., Farmer, J. D., Heinrich, T., Sabuco, J., and Straka, M. J. (2017). Abce: A python library for economic agent-based modeling. In *International Conference on Social Informatics*, pages 17–30. Springer.

Taillandier, P., Gaudou, B., Grignard, A., Huynh, Q.-N., Marilleau, N., Caillou, P., Philippon, D., and Drogoul, A. (2019). Building, composing and experimenting complex spatial models with the gama platform. *GeoInformatica*, 23(2):299–322.

Thiele, J. C. (2014). R marries netlogo: introduction to the rnetlogo package. *Journal of Statistical Software*, 58(2):1–41.

Tisue, S. and Wilensky, U. (2004a). Netlogo: A simple environment for modeling complexity. In *International conference on complex systems*, volume 21, pages 16–21. Boston, MA.

Tisue, S. and Wilensky, U. (2004b). Netlogo: Design and implementation of a multi-agent modeling environment. In *Proceedings of agent*, volume 2004, pages 7–9.

Vahdati, A. R. (2019). Agents.jl: agent-based modeling framework in julia. *Journal of Open Source Software*, 4(42):1611.

Wilensky, U. (2019). Netlogo user manual.

Wilensky, U. (2020). Private communication.