PyLogo

A Python reimplementation of (much of) NetLogo

Russ Abbott*, Jungsoo Lim*, and Ricardo Medina*

* California State University, Los Angeles, USA

Abstract In the world of Agent-Based Modeling (ABM), and especially ABM education, NetLogo reigns as the most widely used platform. According to the NetLogo website, NetLogo *is used by many tens of thousands of students, teachers, and researchers worldwide.*

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, as this paper discusses, the NetLogo language makes model development more painful than necessary.

This combination—widespread popularity accompanies by unnecessary coding pain—motivated the development of PyLogo, a NetLogo-like modeling and simulation environment. The system is written in Python; 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.

The paper focuses on three areas.

- Issues with the NetLogo scripting language, i.e., the language in which one develops NetLogo models.
- The design and organization of PyLogo.
- The PyLogo models we developed.

PyLogo is also a tribute to Python. Work began after the Fall 2019 semester ended. A fully operational version was ready for use a month later when the Spring 2020 semester began. The models discussed in this paper were all written in what might be considered academic "real time", i.e., as needed for class 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.

ACM CCS 2012

- Computing methodologies ~ Simulation environments
- Software and its engineering ~ Multiparadigm languages;

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

The Art, Science, and Engineering of Programming

Perspective The Art of Programming

Area of Submission Simulation environments and languages, Computer Science education, General-purpose programming

1 Introduction

For a senior-level modeling and simulation class we developed a pure-Python version of NetLogo. We had three primary objectives:

1. To offer students a platform for working with agent-based models—especially models with features of complex systems [1]. See Section 5 for example models.
2. To explicate the relatively straightforward structure of a modeling platform. See Section 4 for a discussion of the structure of PyLogo.
3. To help students improve their Python coding skills. (Our department uses Java as its introductory programming language.)

- Why NetLogo?
  - Due to its simplicity and large user community, NetLogo is becoming the standard platform for communicating and implementing ABMs [27].
  - NetLogo is recognized as robust and powerful but nevertheless easy to learn [4].
- Why Python?
  - The NetLogo language makes the development of non-trivial models unnecessarily painful. Section 3 offers a discussion.
  - Python has become the *de facto* scripting language for many areas of computer science, including AI, Machine learning, data science, and others [18]. Computer Science students should develop strong Python skills and intuition.
  - Python has a proud history of giving birth to widely used open-source libraries.

    > *SciPy*, for example, was started in 2001 mainly by graduate students—many without a computer science education. It was preposterous to imagine that a small group of 'rogue' student programmers could upend the well-established ecosystem of research software—backed by millions in funding and many hundreds of highly qualified engineers. Yet a fully open tool stack, combined with an excited, friendly community have proven auspicious in the long run [31].

Our objectives for this paper are:

- ➢ To announce the availability of PyLogo and to offer it to the ABM community as an open-source resource. It is currently available as a GitHub repository.
- ➢ To document some of NetLogo's awkward features. (Section 3).
- ➢ To explain PyLogo's organization and design. (Section 4).
- ➢ To offer examples of models developed in PyLogo—both to illustrate PyLogo's flexibility and as a head-start for new PyLogo model developers. (Section 5).
- ➢ 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 discussion is limited to agent-based systems.

First the classics. *Repast* [19] and *MASON* [15] established agent-based modeling as a distinct discipline. They remain among the most widely-used agent-based modeling

systems, especially for complex models or models with a great many agents. These were followed by AnyLogic [10]. All use Java for model development.

Some more recently developed agent-based systems include:

- *ABCE* [24], a Python-based platform tailored for economic phenomena,
- *Agents.jl* [30], a Julia-based relative newcomer, and
- *GAMA* [25], which describes itself as *an environment for spatially explicit agent-based simulations*. GAMA models are written in GAML, a scripting-like language.

NetLogo [28, 29], a *Repast* and *MASON* contemporary, teased the possibility of writing models in pseudo-English. It grew in popularity to the point that it now

> 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 [22].

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

- The NetLogo Mathematica link allows Mathematica to control NetLogo. [5].
- *NL4Py* [11] and *Pynetlogo* [13] offer access to and control over NetLogo from Python. NetLogo includes an extension that allows calls to Python.
- *RNetLogo* [27] enables one to write *R* code that calls NetLogo functions. A NetLogo extension allows one to call *R* code.

Finally, a number of attempts have been made to build NetLogo-like systems in which the model developer writes in a language other than the NetLogo script.

- *ReLogo* [20], an offspring of *Repast*, replicates many NetLogo features. It uses *Groovy*, a script-like version of Java, as a model-development language.
- *Mesa* [16] comes closest to PyLogo, although it is less complete, e.g., no links. Mesa enables users to write NetLogo-like models in Python. Mesa consists of a Python-based "headless" modeling component, which can either run autonomously or connect to a JavaScript visualization component on a browser.

3 The NetLogo language

This section discusses the NetLogo language and some of the issues that motivated PyLogo. This discussion is not intended as a full analysis.

Before beginning, we wish to acknowledge (again) NetLogo's extraordinary success. In an email, Wilensky [32] reports that since January 2019 there have been approximately: 700k unique visitors to the NetLogo website, 130k NetLogo downloads, and 7000 authors of scientific papers that use NetLogo. The NetLogo website links to approximately 600 courses that use NetLogo.

Wilensky believes that these are significant undercounts. He says he feels comfortable changing *tens of thousands* (on the NetLogo website) to *hundreds of thousands*.

3.1 A brief history of NetLogo

In [28] and [29] Tisue and Wilensky discuss the history of the NetLogo language.

> Logo, as originally developed by Seymour Papert and Wally Feurzeig [8], 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 it.
>
> 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.
>
> The world in which the turtles move is a grid of "patches," which are also programmable. Collectively, the turtles and patches are called "agents." Agents can interact with each other and can perform tasks.
>
> 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, in a pseudo-random order, to perform a series of operations.

Like the original Logo, one of NetLogo's goals was to make Lisp accessible to middle school students. Logo's affinity for Lisp is clear from introductory articles, e.g., [12], which focus on examples that use recursive list manipulation. Consequently, it seems strange that Logo and its descendants resemble traditional imperative languages. Consider this extract from the Logo manual [2].

> All complete Logo statements are imperative. If you type:
>
> SUM 17 26
>
> LOGO will respond with the error message
>
> YOU DON'T SAY WHAT TO DO WITH 43
>
> In contrast:
>
> print SUM 17 26
>
> is a complete statement. The computer will print 43.

Logo ran interactively.

It seems even stranger that NetLogo continued along this path. After all, in his 1978 Turing award lecture, Backus [3] used Lisp-like notation to argue for functional programming. By the time NetLogo was born, the Haskell 98 report [14] had consolidated much of the work on functional programming.

The remainder of Section 3 discusses aspects of the NetLogo language as specified in the *NetLogo User Manual* (version 6.1.1, September 2019). Links point to relevant sections of the Programming Guide or Dictionary.

We wish to acknowledge the clarity and ease of use of the NetLogo documentation.

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 as well.

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, functions such as *if*, *ifelse*, *ifelse-value*, and *while* appear in code as if they were keywords. The Programming Guide includes the following.

> 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.

Functions such as *let* and *set*, infix operators such as *of* and *with*, and identifiers such as *self* and *myself* also look to readers of NetLogo code very much 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.

To deny that identifiers defined via "special forms" are not keywords adds confusion to the language definition.

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; breeds do not have methods that are syntactically marked as restricted to breed instances; even though patches are agents, one can't create a breed of patches; and breed instances can be converted from one breed to anther. In the latter case, when an instance of *breed-1* becomes an instance of *breed-2*, it loses its *breed-1* instance variables and gains those of *breed-2*.

3.4 Sets and lists

Sets and *lists* are treated as unrelated data structures. This raises a couple of issues.

- Why limit sets to agents? Just as one can have lists of agents as well as lists of 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 halve the

number of functions and reduce the memory burden on users. This is straightforward in Python since sets and lists are both types of collections, and one can iterate over Python collections.

Following are some simple illustrative examples.

3.4.1 ask and foreach

Suppose our world consists of five turtles, i.e., *crt 5*, 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 associated number of degrees and then to move forward the associated distance. This would seem 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 ] ] )
```

Since *turtles* is an agentset and *foreach* requires lists, we used

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

to construct an ordered list of turtles to match the lists. Then, since we are requiring the turtles to perform turtle methods, we embed an *ask* in the *foreach* anonymous procedure. The surrounding parentheses are required. Not very pretty code.

A Python version is much simpler. (We assume that *turtle(n)* retrieves the turtle with *who*-number *n* and that *rt* and *fd* are turtle methods.)

```
for (who, (deg, dist)) in enumerate( zip( [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., this GitHub Python Style Guide, generally recommend list comprehensions over *map* and *filter*.

NetLogo does not have a list comprehension form—although it's possible to create a simple list comprehension using *of* and *with*. Based on examples from of and with,

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

has the following Python equivalent, assuming *t.who* retrieves turtle *t*'s *who*-number,

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

3.5 Reporters and simple list comprehensions

NetLogo uses the term *reporter* in multiple ways.

- **Procedures created with** *to-report*. These are the standard user-defined reporters. Most primitive reporters, such as *turtle* and *list*, behave similarly.

- ***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, *who* referred to the *who*-number of the then-active agent. This is done implicitly.
- **anonymous reporters**. These may or may work as one would expect.

To explore these areas, consider the following example.

```
turtle ([who] of a-turtle + 1) ;; Correct parsing requires the parentheses.
```

This reports the turtle with *who*-number one greater than that of *a-turtle*.
One can define a regular reporter, named, say, *next-turtle*, that does the same thing.

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

When applied to some turtle, *next-turtle* produces the correct answer. But attempting to run

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

produces the error message: *NEXT-TURTLE expected 1 input.* (See Section 3.8 for why.)

Nor would wrapping *next-turtle* in an anonymous reporter work: *agent primitives such as* of *and* with *don't accept anonymous procedures.* (See limitations).

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

```
0. next-turtle some-turtle

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 it's first argument plays the same role as the *of* reporter, syntactically, the two reporters are very different.
3. This illustrates how limited anonymous reporters can be. (Recall (a) that anonymous procedures have square brackets as part of their definition and (b) that the surrounding parentheses are required. See Section 3.11.) In most languages with anonymous procedures one can apply anonymous procedures to their arguments in the same way one applies standard procedures to its arguments. That's not the case with NetLogo.

   If one attempts to apply an anonymous procedure to its arguments directly:

   ```
   [ t -> next-turtle t ] some-turtle
   ```

   some-turtle is highlighted with the error: Expected command.

   runresult is required. (One must use *run* for non-reporter anonymous procedures.)

3.6 The if and while constructs require different condition forms

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

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

produces 4.

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

functions properly—and produces nothing. Why require different syntactic forms?

3.7 Non-standard terminology

NetLogo uses *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, concatenates those strings (if there are multiple arguments), and returns the resulting string.
- The function *sentence* combines the functions 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 this attempted definition of *call-first-last*. It expects either of the functions *first* or *last* as its first argument and a list as its second. *call-first-last* applies its first argument to its second to return either the first or last element of the second argument.

```
to-report call-first-last [first-last a-list]
    report first-last a-list
end
```

The preceding fails to compile: *a-list* is highlighted; the error is: Expected command.

One can use *map* to trick NetLogo into compiling.

```
to-report call-first-last [first-last a-list]
    report first map first-last (list a-list)
end
```

But attempted execution

```
call-first-last first a-list
```

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

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

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

Using traditional function-call notation, NetLogo apparently parsed

```
call-first-last first a-list
```

as

```
call-first-last(first(a-list))
```

This problem seems to derive from the decision not to use parentheses to indicate procedure calls. So (almost) any time a procedure name appears, the NetLogo parser takes it as a procedure call.

The only apparent way around this is to wrap the procedure name in an anonymous procedure. For example, to put the procedures fd and rt into a list—[fd rt]—requires list [n -> fd n] [d -> rt d], which produces a list of those two anonymous commands.

3.9 Scoping

NetLogo has quite rigid scoping rules. From the *Programming Guide*:

> All primitives, global and agent variable names, and procedure names share a single global case-insensitive namespace; local names (let variables and the names of procedure inputs) may not shadow global names or each other.

This precludes the parameters *x* and *dx* and the local variables *y* and *z*.

```
globals [x y]

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

3.10 Square brackets and parentheses are syntactically overloaded

The anonymous procedures in the *foreach* examples above illustrate another problem: square brackets are confusingly overloaded. In particular, the square brackets that surround anonymous procedures are a required part of their syntax. They are not used for grouping, like parentheses, and they are not used to indicate lists, as square brackets often are. The second arguments in each of the *foreach* examples are anonymous procedures, not lists or code blocks containing anonymous procedures.

Similarly the surrounding parentheses in the final *foreach* example above are required even though they do not perform a grouping function. See Section 3.11.

3.11 List construction

In NetLogo, the list [1 2] is ok, but the would-be list [(turtle 0) (turtle 1)] is not. The error message says that lists formed with square brackets must consist entirely of literal values such as numbers, strings, and lists of numbers and strings.

Furthermore, list turtle 0 and list (turtle 0) are also invalid, whereas (list turtle 0) is ok. The rule is that the *list* function requires bounding parentheses even when there is no ambiguity about the syntactic structure—unless one is forming a list of two elements, in which case parentheses are optional.

More generally, it seems that surrounding parentheses are required around any function—such as *foreach*, *ifelse*, *ifelse-value*, *list*, *map*, *patch-set*, *run*, *runresult*, *sentence*, *turtle-set*, and *word*—that takes a variable number of arguments. That *list* does not require surrounding parentheses when given two arguments seems to be an exception. See list construction in the *Programming Guide* for additional details.

3.12 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?

3.13 Dictionaries

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

3.14 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 some-turtle rather than some-turtle.who. (See Section 3.5). Dot notation is simpler and cleaner.

4 PyLogo design

This section discusses the PyLogo design. The first subsection provides an overview. The second discusses individual modules and other lower-level concerns.

4.1 PyLogo Overview

This overview is divided into three sections. (a) the structure of a standard PyLogo model, (b) PyLogo's tiered architecture, and (c) the control flow of running a model.

4.1.1 Model structure

This section presents the structure of a standard PyLogo model. Annotated versions of *Game of Life* and *Starburst*, our two smallest models, are in Appendices H and M. (In *Starburst*, agents respond to the repulsive forces of other agents.)

All PyLogo models have the following skeleton.

```
import ...

class <model>-Agent(Agent): # # A subclass of Agent is optional.
    ...

class <model>-Link(Link): # # # A subclass of Link is optional.
    ...

class <model>-Patch(Patch): # # A subclass of Patch is optional.
    ...

class <model>-World(World): # # A subclass of World is required..
    def setup(self):
        ...

    def step(self):
        ...

# # # # # # # # Define <model>-specific GUI elements # # # # # # # #
gui_left_upper = ...

if __name__ == "__main__":
    PyLogo(<model>-World, '<model-name>', ... )
```

- **Imports.** Like most Python programs, PyLogo models import from other modules.
- ***<model>-Agent* class**. If included, *<model>-Agent* would be an *Agent* subclass. (See Section 4.2.6 for more about the *Agent*, *Link*, *Patch*, and *World*, classes)
- ***<model>-Link* class**. If included, *<model>-Link* would be a *Link* subclass.
- ***<model>-Patch* class**. If included, *<model>-Patch* would be a *Patch* subclass.
- ***<model>-World* class**. *<model>-World* is a *World* subclass. Virtually all *<model>-World* classes define the functions, *setup* and *step*, which override the abstract versions in the *World* class.
  - *setup* is the PyLogo analog to NetLogo's *setup* function. It runs when the user clicks the *setup* button—or when the model is loaded if *auto_setup* had been set to *True*. *setup* creates the initial complement of agents; it initializes their properties; and it situates them on the grid.
  - *step* is the PyLogo analog to NetLogo's *go* function. It modifies the *<model>-World* to reflect the passage of one model *tick*. Clicking the *go* button starts PyLogo's model event-loop (Section 4.1.2). The model event-loop calls the *step* function once per cycle—thereby implementing NetLogo's *go "forever"* button.
- **GUI.** Virtually every PyLogo model includes a GUI. Although a bit bulky, *PySimpleGUI* (Section 4.1.2) makes it quite easy to specify a desired GUI.
- **Call the *PyLogo* function**. Models may be executed by running their files. To run, a model calls the *PyLogo* function, which takes a number of model-level parameters. The *PyLogo* code and its parameters are shown in Appendix O.

4.1.2 Tiered architecture: levels of abstraction

The PyLogo code has two relevant top-level directories: *core* and *models* (The *assignments* directory contains partial models to be completed by students.) Table 1 shows the *core* modules and their levels of abstraction. Modules may import from

lower-level modules. They may also import from modules on the same level as long as circular imports are avoided. Following are more details.

- **Advanced features**. Although in the *core* directory, these modules are organized like models: all subclass the *World* class. Derived models access them via inheritance.
  - *ga.py*. Includes support for genetic algorithms. See Section 4.2.1. Models in *models/ga_and_aco_examples* inherit from *ga.py*.
  - *graph_framework.py*. Defines a subclass of *Agent*. Includes functions for the construction, manipulation, and analysis of graphs. See Section 4.2.3. The model *graph_algorithms* inherits from *graph_framework.py*.
  - *on_off.py*. Defines subclasses of *Patch* and *World*. Includes functions to change patch colors and to turn patch display on and off. See Section 4.2.4. Used by the *Cellular Automata* models.
- **User-level classes**. These are quasi-abstract versions of classes which users extend to build models. See Section 4.2.6.
  - *agent.py*. Defines the top-level *Agent* class.
  - *link.py*. Defines the top-level *Link* class.
  - *world_patch_block.py*. Defines the top-level *Patch* and *World* classes.
- **2-D elements**. Points and other support for operating in 2D space.
  - *pairs.py*. See Section 4.2.5.
- **Event loops**. The two event-loops that run the system.
  - *sim_engine.py*. Includes a *SimEngine* **class**, which runs the two event loops:
    (a) *top_loop* runs when a model is loaded but not running. (See Appendix A.)
    (b) *model_loop* runs when a model is running. It calls the model's *step* function.
- **GUI and utils**.
  - *gui.py*. Defines GUI elements, including: a few of the NetLogo figures, the default patches array size (51 *x* 51), and the default patch size (11 pixels). Patches are drawn with one-pixel borders. (NetLogo draws patches without borders.)
    gui.py also calls *draw* functions from *Pygame*. It also includes a *SimpleGUI* class, which includes *PySimpleGUI* code to integrate *PyGame* with *tkinter*. When instantiated *SimpleGUI* creates the *PySimpleGUI* window.
  - *utils.py*. NetLogo-style trigonometric functions (Section 4.2.6) and other utilities.
- **Libraries**. The imported libraries.
  - NumPy. Used minimally—only for dealing with the array of patches.
  - Pygame. Supports drawing, color functionality, and frame-rate control.
  - PySimpleGUI. Used for the layout and operation of interaction widgets. Provides the event mechanism for the event loops. Overlays Python's tkinter package.

  PyLogo also uses Python libraries such as *itertools*, *math*, *random*, and *typing*.

Advanced features	ga.py, graph_framework.py, on_off.py
User-level classes	agent.py, link.py, world_patch_block.py
2-D elements	pairs.py
Event loops	sim_engine.py
GUI and utils	gui.py, utils.py
Libraries	NumPy, Pygame, PySimpleGUI

Table 1 PyLogo tiered architecture: levels of abstractions

4.1.3 Control flow

This section describes the flow of control both while a model is loading and running. A model is run by running its file. Running a file calls the *PyLogo* function, which instantiates the system and starts the *top_loop* event loop (see Appendix A). That loop runs until the user clicks the *setup* button, the *go* button, or the *go_once* button.

- The *setup* button runs the *World*'s *setup* function. The *top_loop* continues afterwards.
- The *go* button runs the *model_loop*. On each cycle, it reads values and senses events from the GIU and calls the *World*'s *step* function.
- The *go_once* button runs the *World*'s *step* function once but stays in the *top_loop*.

The model stops when the user clicks *Exit* or the model indicates it's *done*.

4.2 Details

This section discuss a number of more narrowly focused design issues.

4.2.1 ga.py

The *ga.py* module defines a genetic algorithm framework. *ga.py* includes a *GA_World* class, which developers may extend in their models. It also includes an *Individual* class and a *Chromosome* class.

- *GA_World* runs the genetic algorithm. It stores the *population* of *Individuals* and runs the evolutionary process. On each call to its *step* function, it generates the next population generation.

  GA_World uses a steady-state approach when generating the next generation. In particular, the *step* function consists of a loop that generates *self.pop_size//2* pairs of new *Individuals*: each "breeding" step, produces two children.

  The breeding step uses tournament selection to choose two parents. The tournament size can be set through a widget. Parents mate (or with some probability are simply duplicated). The children then mutate (again with some probability).

  The *mating* function uses a user-selected mating strategy to produce children. Two mating strategies are pre-defined, although users may define their own. The pre-defined strategies are uniform cross-over and cross-over that preserves gene uniqueness. The latter is typically used for problems like the Traveling Salesman

Problem (TSP) in which a chromosome is a sequence of genes, each of which is one of the cities to be included in the tour. All chromosomes have the same genes.

Once generated, the children are placed back into the population using reverse tournament selection to determine which population elements to replace.

A number of mutation operators are pre-defined—although again, users may write their own. These include: *flip-a-gene* (which can be used when the genes are binary), *move-a-gene*, which moves a gene from its current location in the chromosome to a randomly selected alternative location, and *2-opt* (see Figure 1), which reverses a random subsection of the chromosome.

If the problem is such that *Chromosome* rotation does not change a *Chromosome*'s meaning—as is the case with TSP and quite a few other problems—*GA_World* sorts the population after each *step* and removes duplicate *Individuals*, replacing them with randomly generated new *Individuals*. This strategy helps to maintain population diversity and helps to minimize premature convergence.

- *Individual*. The population consists of *Individuals*, each of which is a *Chromosome* along with additional information such as fitness. The mating and mutation operations are methods in the *Individual* class.
- A *Chromosome* is a sequence of genes, where a *gene* is any entity type. Genes may be agents. In TSP, a *Chromosome* is a sequence of *Agents* representing cities to be visited. (See Section 5.2.)

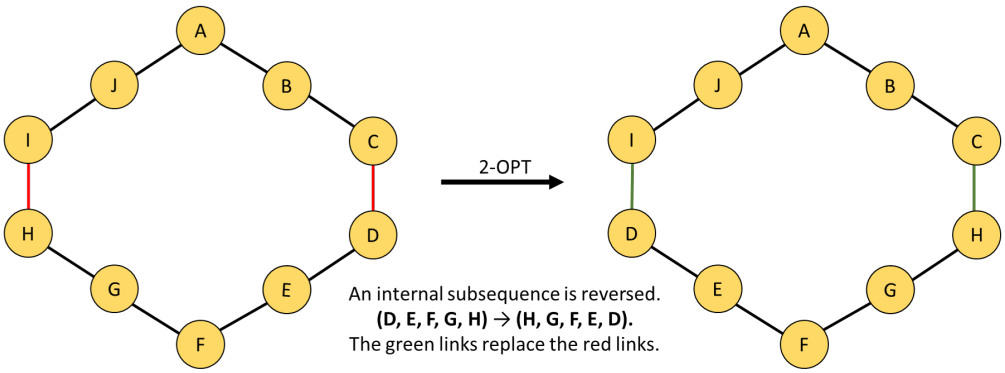


Figure 1 The 2-opt mutation

The *Chromosome* class is defined as a subclass of *tuple*; each *Chromosome* is simply a *tuple* with additional methods and instance variables. Since tuples are immutable, *Chromosomes* are also immutable and cannot be changed accidentally during mating or mutation. The breeding and mutation operators defined at the *Individual* level are implemented at the *Chromosome* level.

This abstract GA model includes a number of parameters, which the user may set—e.g., *Population size*, *Tournament size*, etc. These parameters appear in all models that use this framework. See Section 5.2 for models that use this framework.

4.2.2 The Grab anywhere checkbox

When checked, the *Grab anywhere* checkbox, which appears in all PyLogo model GUIs, allows the user to move the model window by clicking and dragging anywhere in the window. This is useful when the window is larger than the user's computer screen, and the top frame border is off the screen. This checkbox is unchecked by default: when checked it interferes with the operation of the other widgets.

4.2.3 graph_framework.py

The *graph_framework.py* module provides a framework for constructing graphs, i.e., nodes connected by edges. Model developers may subclass its classes. For an example, see *graph_algorithms.py*, Section 5.5.

4.2.4 on_off.py

The *OnOffPatch* and *OnOffWorld* classes support models that use patch color to convey information. The two Cellular Automata models (Section 5.3), *1D CAs* and *Game of Life*, extend this model. The *set_on_off* method marks a *Patch* as "on" or "off," each with an associated color. *OnOffWorld* methods enable users to set the "on" and "off" colors and to click and change the color of individual patches. (To change a color, one must click the labelled buttons rather than the colored panels to their right.)

4.2.5 pairs.py

The *pairs* module defines four classes for pairs of elements. (See Figure 2.) All classes subclass *tuple* and add additional methods.

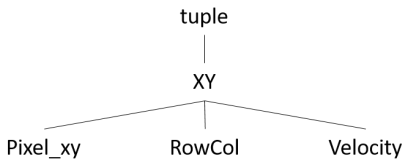


Figure 2 Pairs classes

- *XY* is a generic pairs class. It collects methods common to its subclasses. Most notably, it defines the *__str__* function and the arithmetic infix operators differently from how they are defined on tuples. See Figure 3.

```
tuple_3_4 = (3, 4)
# The argument to XY is a tuple, not two elements.
xy_3_4 = XY( (3, 4) )
print(f'a. {tuple_3_4} == {xy_3_4}: {tuple_3_4 == xy_3_4}')
print(f'b. {tuple_3_4} + {tuple_3_4}: {tuple_3_4 + tuple_3_4}')
print(f'c. {xy_3_4} + {xy_3_4}: {xy_3_4 + xy_3_4}')
print(f'd. {tuple_3_4} * 5: {tuple_3_4 * 5}')
print(f'e. {xy_3_4} * 5: {xy_3_4 * 5}')
```

```
a. (3, 4) == XY(3, 4): True
b. (3, 4) + (3, 4): (3, 4, 3, 4)
c. XY(3, 4) + XY(3, 4): XY(6, 8)
d. (3, 4) * 5: (3, 4, 3, 4, 3, 4, 3, 4, 3, 4)
e. XY(3, 4) * 5: XY(15, 20)
```

Figure 3 The code on the left produces the output on the right.

- *Pixel_xy*. A pair of integers to refer to a pixel. Pixel-based operations are defined.

- *RowCol*. A pair of integers to refer to a patch in the grid of patches. Includes: *patch_to_center_pixel*, which returns the center pixel of the *Patch* at this *RowCol*.
- *Velocity*. A pair of numbers to indicate an agent's velocity in pixels/*step*.
- *force_as_dxdy*, defined at the module level, *force_as_dxdy* returns a delta-*Velocity* representing the effect of the force between two *Pixels*. It is used by the *graph_framework* (Section 4.2.3) and the *starburst* (Section 5.8) models.

4.2.6 User-level classes

PyLogo defines four major user-level classes: *Agent*, *Link*, *Patch*, and *World*. These plays the roles in PyLogo as the corresponding elements do in NetLogo.

- *Agent*. The *Agent* class provides functions for moving around the grid.

  Agents have a *velocity* instance variable. (Section 4.2.5.) To support velocities, the *Agent* class includes (a) two *static* methods, *update_agent_positions* and *update_agent_velocities*, and (b) two instance-level methods, *move_by_velocity* and the overrideable *update_velocity*. See the *Starburst* model (Appendix M).

  Agents have a *heading*: due North, is a heading of 0 degrees; due East is 90 degrees; etc. PyLogo restricts headings to integer values—which makes caching calls to trigonometric functions worth the trouble. (See Section 4.2.7.)

  Agents have optional and optionally displayable labels. The *Minority game* (Appendix K) and the *Traveling Salesman Problem* (Section 5.2) use this capability.

  The *Agent.id* instance variable corresponds to NetLogo's *who* number.

  Agents support simple key_frame animation. See the *spring_paradox* (Section 5.1).

  Perhaps half of the models, including *Starburst*, extend the *Agent* class.
- *Link*. PyLogo *Link*s may be directed or undirected. *Link*s offer functions to identity the agents at the link ends and to help when doing a graph search. *Graph algorithms* (Appendix J) uses but doesn't extend *Link*s. The *Spring paradox* (Section 5.1), and the models discussed in Section 5.2.1 extend the *Link* class.
- *Patch*. Every *Patch* includes a set of the *Agents* on that *Patch*. (*Agents* keep this updated.) The *Segregation* model (Section 5.7) depends on this. *Patch* color functions play important roles in the *Cellular Automata* models (Section 5.3). The *Game of Life* model (Appendix H) extends the *Patch* class.
- *World*. The *World* class includes abstract versions of the *setup* and *step* functions and other support for initializing models. All models extend the *World* class.
  The *World* class also includes the following abstract functions.
  - *final_thoughts*. Useful for logging final statistics when a model is finished.
  - *handle_event*. Called by the event loops when a user creates a model-level event, such as clicking a model-specific button.
  - *mouse_click*. Called when the user clicks in the grid.

4.2.7 utils.py

The most important *utils* functions are *heading*-oriented trigonometric functions. They are normalized to either *range(360)* or *range(-180, 180)*. It's often awkward to explain *heading* arithmetic; comments may be longer than executable code.

5 Example models

This section discusses the models in the *PyLogo/models* directory. Many have parallels in NetLogo's *Models Library*.

5.1 Braess Paradox (Figure 4)

Braess paradox is often illustrated by the phenomenon that adding a road to a network may make travel time worse. [7] At nearly 450 lines of code, the NetLogo model (Social Science → Economics) is one of the more complex. A perhaps superficial but striking feature is the model's attractiveness. The roads are nicely colored and laid out; the grassy areas are made up of randomly selected shades of green.

Students were asked first to read and understand the NetLogo model and then to write their own PyLogo models. The student models (400 to 600 lines of code) replicated the NetLogo computations—and were equally attractive.

Penchina and Penchina [21] explain how springs can exhibit Braess-like counter-intuitive behavior. We developed an illustrative PyLogo spring model, which combines agent-based simulation with a simple key-frame animation capability.

5.2 ga_and_aco_examples

These are models that use the *ga.py* framework (Section 4.2.1).

5.2.1 Models related to the Traveling Salesman Problem (TSP)

- *ga_closed_paths* (Figure 5). This is *not* a TSP model. In this model a number of points are randomly distributed across the screen. The user sets a *fitness target* and a *cycle length*. The goal is to find a closed path (with *cycle length* steps) through a subset of these points whose total length is closest to the *fitness target*.

  This model is engaging in that the user may select to have the points move slowly but randomly. Note that the default *Max generations* is set to *inf*—i.e., the GA never stops. As the points move, tour lengths change, and the GA searches for other point subsets and tours that more closely approximate the desired *fitness target*.
- *ga_tsp* (Figure 6). This *is* a TSP model. The user selects a number of points, which are placed randomly on the screen. The model operates in two step.
  - *Generate the population* (**setup**). Individuals are generated using one of three randomly selected algorithms: random path, greedy path, and spanning-tree path. When *Animate construction* is checked, the construction steps for the greedy path and the spanning-tree path are shown. The spanning tree is shown in green.
  - *Run the GA* (**go**). As in the *ga_closed_paths* model, the GA runs continually. Besides optionally moving points, users may add or remove points dynamically.
- *aco_tsp* (Figure 7). This is an *Ant-Colony Optimization* approach to TSP. (The result seems less successful than using a GA. For an effective display we selected only seven cities.) The display shows both the currently selected tour (thicker light blue lines) along with other highly rated links (thinner green, yellow, and red lines).

The link labels show the "pheromone levels," normalized to a maximum of 100. Only links with levels of at least *Min pheromone* (user selectable) are shown.

5.2.2 Other ga-related models

- *ga_knapsack* (Figure 8). This model solves very easy knapsack problems. A number of example are built into the model. The output appears in text form.

  In Figure 8, the system solved *Problem 4* in six generations. The output shows the entire population (5 elements) for each generation. An individual (e.g., the first one in Generation 0) is shown as 96/49<-0111000. This individual produces a value of 96 with a weight of 49 by selecting items 1, 2, and 3. After 6 generations (i.e., in Generation 5) the system found an individual that selects items 0 (value: 70, weight: 31) and 3 (value: 37, weight: 19) for the best possible result.
- *ga_segregation* (Figures 9 and 10) This is a one-dimensional segregation model. The world consists of a circular sequence of cells—so rotation is irrelevant. Each cell has four neighbors, two on each side. An individual is happy if *more* of its neighbors are its color than the other color.
  Figures 9 and 10 show a complete run. The lines with green and yellow patches show the best individual of that generation. (The pale blue patches are unoccupied.) A red patch beneath an agent indicates an unhappy agent.

  For example, the first (yellow) patch on the first line in Figure 9 has a red patch beneath it because it has two cells to its right (one yellow and one green) and two cells to its (wrapped-around) left (also yellow and green). It's unhappy because it is not the case that a majority of its neighbors are yellow, as it is.

  As the model runs, the GA moves agents around, and fewer and fewer red cells appear. (Note that the final two green and yellow lines at the bottom of Figure 9 are repeated at the top of Figure 10.) Finally, at the bottom of Figure 10, all agents are happy. Given the rules, the only way for all agents to be happy is for them to form homogeneous blocks of size 2 or greater separated by empty cells.
  There are a couple of interesting technical issues.

  1. As in TSP, all *Individuals* must have exactly the same agents. Using a *Named tuple*, each agent is defined by an *id* and a *value* (color). Mating must produce children which also have the same agents. The *cx_all_diff* mating function, defined in the *ga.py* framework, guarantees that result.
  2. *Segregation_World* has no *step* function. It uses *step* defined in *GA_World*.
- *ga_parentheses* (Figure 11). This model accepts a list of left and right parentheses—the same number of each—but in random order. The goal is to reorder them so that the parentheses balance. This model inherits from *ga_segregation*. Although this and *ga_segregation* seem quite different, the main differences are the fitness function and the choice of mutation operators. The example shown in Figure 11 requires 5 generations to balance a string with 6 parentheses initially out of place.

5.3 Cellular Automata

Two Cellular Automata models extend the *OnOff World* model (Section 4.2.4).

- *One-dimensional Cellular Automata (1D-CA)* (Figure 12) are the kind Wolfram popularized [33]. (See a mathematical perspective on Wolfram's work [9].)
  The PyLogo 1D-CA model has a fairly sophisticated GUI. Users can specify:
  - the rule that drives the CA either by clicking (binary) digits or via a slider;
  - whether to center the expansion or produce it flush left or right; (Figure 12 shows the initial part of a run in which the user selected centering.)
  - the initial string to be expanded; and
  - the colors for the *on* and *off* cells, which may be changed dynamically.
- *Game of Life* (Appendix H). Like the *1D-CA* model, users may change the colors of the *on* and *off* dynamically. Users may also toggle cells when the model is paused.
  The *Game of Life* model is the smallest PyLogo model. It's short primarily because *onOffPatch* and *onOffWorld* (Section 4.2.4) do much of the work.
  A question that often arises in *Game of Life* models is how to handle the transition from one state to the next: we don't want cells from the upcoming state effecting transitions for cells from the current state.
  We make two passes over the cells.
  1. Use a cell instance variable to store the number of live neighbors for each cell.
  2. Convert those stored values to whether a cell is live or dead.

  Appendix H includes the entire model, well annotated.

5.4 Flocking (Figures 14, 15, and 16)

Reynolds' [23] classic flocking model illustrates how three relatively straightforward rules can give rise to flocks—and other groups such schools of fish and herds of cattle.

The model includes the option of showing what NetLogo calls *flockmates* with pale blue links. The model also gives the user the option of turning *bounce* on and off.

- The first image shows three small flocks shortly after the model starts running.
- The second image, taken a few *steps* later, shows that two members of the five-element flock at the bottom right have gone off the bottom of the screen—and reappeared at the top. Since the world is toroidal (bounce is unchecked), the five elements are still a single flock; the links stretch across the screen. Notice that the two flocks to the left in the first image have merged into a single larger flock.
- Then *bounce* is checked. The third image shows that the five-element flock on the right is now two flocks: the world is no longer toroidal.

5.5 Graph Algorithms (Figures 17, 18, 19, 20, and 21)

The *Graph Algorithms* model uses the *graph_framework* module. It finds shortest paths within a graph as nodes and links are added and removed.

- **Figure 17.** Initially a wheel graph is shown. Nodes repel each other. Edges stretch, but like rubber bands, the longer they become, the more they resist stretching.
- **Figure 18.** Clicking on or near a node selects the node. (The *click* functionality is a bit finicky.) Clicking two nodes produces a shortest path between them (if any).
- **Figure 19.** Clicking *Delete a shortest-path link*, results in an alternate path.

- **Figure 20.** Deleting a few more shortest-path links stretches out the graph.
- **Figure 21.** *Creating* a few random links results (in this example) in this formation.

5.6 Minority Game (Figures 22 and 23)

In the Minority Game, each agent selects either 0 or 1. An agent wins if its selection is in the minority. This model has 25 agents. Agents 1 - 23 use the strategy in the NetLogo model: *Models Library: Sample Models → Social Science*. Agents 0 and 24 select randomly; one of these almost always wins! (See Appendix K for an explanation.)

Figure 22 shows the "horse race." The agents farther to the right have more points. Figure 23 shows a partial log. In this case, it shows the state of the race from steps 19 through 54, when the screen shot was taken. Each step shows:

- the step number,
- the five most recent winning guesses (history marches to the left),
- the history as an index into the agents' strategies (history as a binary number),
- the winning guess this turn,
- the leading agent(s): shown as: <agent id>: <agent points>/<number required to win>

In this case, we have just completed step 54; the history is [1, 0, 1, 0, 0], which is taken as index 20 into agent strategies; 0 was the winning guess; agent 24 was the only leader, with 28 points out of the 50 required to win.

5.7 Segregation (Figures 24a and 24b)

This is Schelling's famous *Segregation* model. The model has two features of interest.

- Require each agent to have 100% similar neighbors—not Schelling's original point. The NetLogo model never manages to make more than 15% of the agents happy—even with density at 50%. That's striking because with half the patches unoccupied it shouldn't be too difficult to make all agents happy. The PyLogo model with 100% similarity and a density of 90% can make all the agents happy. (See Figure 24.)

  The results are so different because the two models move unhappy agents in different ways. When the NetLogo model moves an unhappy agent, it has the agent take a random walk until it lands on an empty patch. When the PyLogo Model moves an unhappy agent, it searches a list of empty patches for a patch where it would be happy. If it finds one, it puts the agent there. Otherwise, it moves the agent to a random empty patch.

  The PyLogo approach more closely resembles how people actually move. They don't randomly look for a vacant house near their current house. They look for a vacant house that will make them happy, even if it's not close to their current location.
- The PyLogo model displays the world by changing *Patch* colors. A *Patch* is given the color of the agent that occupies it, if any. Agents are not displayed.

  To do this, the model overrides the *draw* function. The standard *draw* function, defined in the *World* class, draws the *Patch*es, *Agent*s, and *Link*s. The *SegregationWorld draw* function draws only the *Patch*es. PyLogo makes it easy to override features even as fundamental as the *draw* function.

5.8 Starburst (Figure 25)

Agents respond to repulsive forces of other agents. (The beginning is especially fun to watch.) Appendix M contains an annotated version of the full model.

5.9 Synchronized Agents (Figure 26)

The *Synchronized Agents* model illustrates how even random motions create attractive patterns if all agents make *the same* random moves.

6 Possible next steps

- **Reinforcement learning and ant-colony optimization.** Ant-colony optimization bears a close resemblance to reinforcement learning using *Q-Learning* [6]. We are considering developing frameworks for both—similar to the frameworks we developed for genetic algorithms—along with models that will illustrate their similarities and differences.
- **A two-phase *step* framework.** A number of our models—*One-Dimensional-Cellular Automata*, *Game of Life*, *Minority game*, *Starburst*, and perhaps others—use a two-phase *step* process: (i) based on the current state of the world, decide for each agent how it will act; (ii) carry out those actions for all agents. This approach suits models in which an agent's actions depends on the current state of the world but not on the (concurrent) actions of other agents. Two-phase *step* doesn't work for models in which agents may take incompatible actions. In Schelling's segregation model, for example, multiple agents may decide to move to the same vacant cell.

  Two-phase *step* may be common enough to justify a supporting framework. Parallelization would be very effective for models that can use two-phase *step*.

  NetLogo randomizes the order in which agents act. This may be important for some models but not for others. If we develop a support framework for two-phase *step* it makes sense also to provide the option of randominizing agent activation order for other models. The *random.sample* function would do the job nicely.

7 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, the models discussed in this paper run quite well.)

  We welcome the participation of anyone knowledgeable about Python optimization.
- PyLogo doesn't offer graphing. We welcome the participation of anyone knowledgeable about any of the Python visualization libraries. See, for example, [26] and [17] for two recent (overlapping) lists of plotting options.

Notwithstanding these limitations, PyLogo was more than adequate for teaching about agent-based modeling.

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

Yet 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 after a month of development; additional features plus a broad range of models developed while using the system for a class—demonstrates that Python enables the rapid development of a fairly sophisticated system.

When we embarked, we were unsure how well Python—along with PyCharm, our IDE—would support small-to-medium-scale development. We were pleased with our experience and leave it to readers to judge the result.

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

References

[1] Russ Abbott. "What Makes Complex Systems Complex?" In: *Journal on Policy and Complex Systems* 4.2 (2018), pages 77–113.

[2] Hal Abelson, Nat Goodman, and Lee Rudolph. "Logo manual". In: (1974).

[3] John Backus. "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs". In: *Communications of the ACM* 21.8 (1978), pages 613–641.

[4] Jennifer Badham. "Review of *An Introduction to Agent-Based Modeling: Modeling Natural, Social, and Engineered Complex Systems with NETLogo*". In: (2015).

[5] E Bakshy and U Wilensky. "NetLogo-Mathematica Link". In: *Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL.* (2007). URL: http://ccl.northwestern.edu/netlogo/mathematica.html.

[6] Reinaldo AC Bianchi, Carlos HC Ribeiro, and Anna HR Costa. "On the relation between ant colony optimization and heuristically accelerated reinforcement learning". In: *1st international workshop on hybrid control of autonomous system*. AAAI Palo Alto, CA. 2009, pages 49–55.

[7] Dietrich Braess, Anna Nagurney, and Tina Wakolbinger. "On a paradox of traffic planning". In: *Transportation science* 39.4 (2005), pages 446–450.

[8] Wallace Feurzeig and Seymour Papert. "The logo programming language". In: *ODP-Open Directory Project* (1967).

[9] Lawrence Gray, A New, et al. "A mathematician looks at Wolfram's new kind of science". In: *Notices of the American Mathematical Society 50 (2)(2003) 200–211, URL http://www. ams. org/notices/200302/fea-gray. pdf. URL http://www. ams. org/notices/200302/fea-gray. pdf*. Citeseer. 2003.

[10] Ilya Grigoryev. "AnyLogic 7 in three days". In: *A quick course in simulation modeling* 2 (2015).

[11] Chathika Gunaratne and Ivan Garibay. "NL4Py: Agent-Based Modeling in Python with Parallelizable NetLogo Workspaces". In: *arXiv preprint arXiv:1808.03292* (2018).

[12] Brian Harvey. "Why Logo". In: *Byte* 7.8 (1982), pages 163–193.

[13] Marc Jaxa-Rozen and Jan H Kwakkel. "Pynetlogo: Linking netlogo with python". In: *Journal of Artificial Societies and Social Simulation* 21.2 (2018).

[14] PEYTON JONES et al. "The Haskell 98 Report". In: *http://www. haskell. org/definition/* (1999).

[15] Sean Luke, Claudio Cioffi-Revilla, Liviu Panait, Keith Sullivan, and Gabriel Balan. "Mason: A multiagent simulation environment". In: *Simulation* 81.7 (2005), pages 517–527.

[16] David Masad and Jacqueline Kazil. "MESA: an agent-based modeling framework". In: *14th PYTHON in Science Conference*. 2015, pages 53–60.

[17] Josh Miramant. *Visualization Libraries for Machine Learning with Python*. Dec. 2019. URL: https://blueorange.digital/visualization-libraries-for-machine-learning-with-python/.

[18] Abhinav Nagpal and Goldie Gabrani. "Python for data analytics, scientific and technical applications". In: *2019 Amity International Conference on Artificial Intelligence (AICAI)*. IEEE. 2019, pages 140–145.

[19] Michael J North, Nicholson T Collier, Jonathan Ozik, Eric R Tatara, Charles M Macal, Mark Bragen, and Pam Sydelko. "Complex adaptive systems modeling with Repast Simphony". In: *Complex adaptive systems modeling* 1.1 (2013), page 3.

[20] Jonathan Ozik, Nicholson T Collier, John T Murphy, and Michael J North. "The ReLogo agent-based modeling language". In: *2013 Winter Simulations Conference (WSC)*. IEEE. 2013, pages 1560–1568.

[21] Claude M Penchina and Leora J Penchina. "The Braess paradox in mechanical, traffic, and other networks". In: *American Journal of Physics* 71.5 (2003), pages 479–482.

[22] Steven Railsback, Daniel Ayllón, Uta Berger, Volker Grimm, Steven Lytinen, Colin Sheppard, and Jan Thiele. "Improving execution speed of models implemented in NetLogo". In: *Journal of Artificial Societies and Social Simulation* 20.1 (2017).

[23] Craig W Reynolds. "Flocks, herds and schools: A distributed behavioral model". In: *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*. 1987, pages 25–34.

[24] Davoud Taghawi-Nejad, Rudy H Tanin, R Maria Del Rio Chanona, Adrián Carro, J Doyne Farmer, Torsten Heinrich, Juan Sabuco, and Mika J Straka. "ABCE: A Python Library for Economic Agent-based Modeling". In: *International Conference on Social Informatics*. Springer. 2017, pages 17–30.

[25] Patrick Taillandier, Benoit Gaudou, Arnaud Grignard, Quang-Nghi Huynh, Nicolas Marilleau, Philippe Caillou, Damien Philippon, and Alexis Drogoul. "Building, composing and experimenting complex spatial models with the GAMA platform". In: *GeoInformatica* 23.2 (2019), pages 299–322.

[26] Shaun Taylor-Morgan. *The 7 most popular ways to plot data in Python*. Apr. 2020. URL: https://opensource.com/article/20/4/plot-data-python.

[27] Jan C Thiele. "R marries NetLogo: introduction to the RNetLogo package". In: *Journal of Statistical Software* 58.2 (2014), pages 1–41.

[28] Seth Tisue and Uri Wilensky. "Netlogo: A simple environment for modeling complexity". In: *International conference on complex systems*. Volume 21. Boston, MA. 2004, pages 16–21.

[29] Seth Tisue and Uri Wilensky. "NetLogo: Design and implementation of a multi-agent modeling environment". In: *Proceedings of agent*. Volume 2004. 2004, pages 7–9.

[30] Ali R Vahdati. "Agents.jl: agent-based modeling framework in Julia". In: *Journal of Open Source Software* 4.42 (2019), page 1611.

[31] Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, et al. "SciPy 1.0: fundamental algorithms for scientific computing in Python". In: *Nature methods* 17.3 (2020), pages 261–272.

[32] Uri Wilensky. Private communication. July 2020.

[33] Stephen Wolfram. *A new kind of science*. Volume 5. Wolfram media Champaign, IL, 2002.

A Event loop

This is a somewhat simplified version of the *top_loop*. It runs when a model is loaded but not running.

```
while SimEngine.event not in [self.ESCAPE, self.q, self.Q, self.CTRL_D, self.CTRL_d]:

        # Read the current PySimpleGUI values.
        (SimEngine.event, SimEngine.values) = gui.WINDOW.read(timeout=10)

        ...

        # The normal case is a timeout, i.e., no events occurred. Go around the loop again.
        if SimEngine.event == '__TIMEOUT__':
            continue

        # Every model has a GOSTOP button. Its default value is "Go".
        # When clicked, the model loop is run, and the GOSTOP button become a "Stop" button.
        # When the model run terminates, the GOSTOP button becomes "Go" again.
        elif SimEngine.event == GOSTOP:
            # Run the model loop
            ...

        ...

        else:
            # If an event occurs that this loop doesn't recognize, such as a model-specific button,
            # pass it on to the model to handle.
            SimEngine.world.handle_event(SimEngine.event)
```

B Braess Paradox

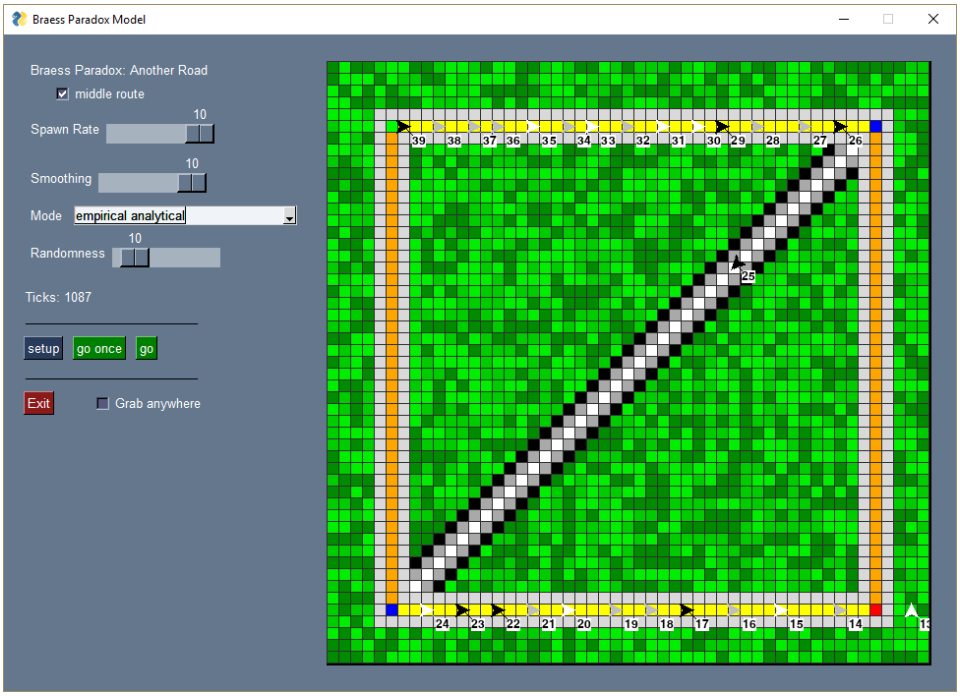


Figure 4 Braess Road

C ga_and_aco_examples

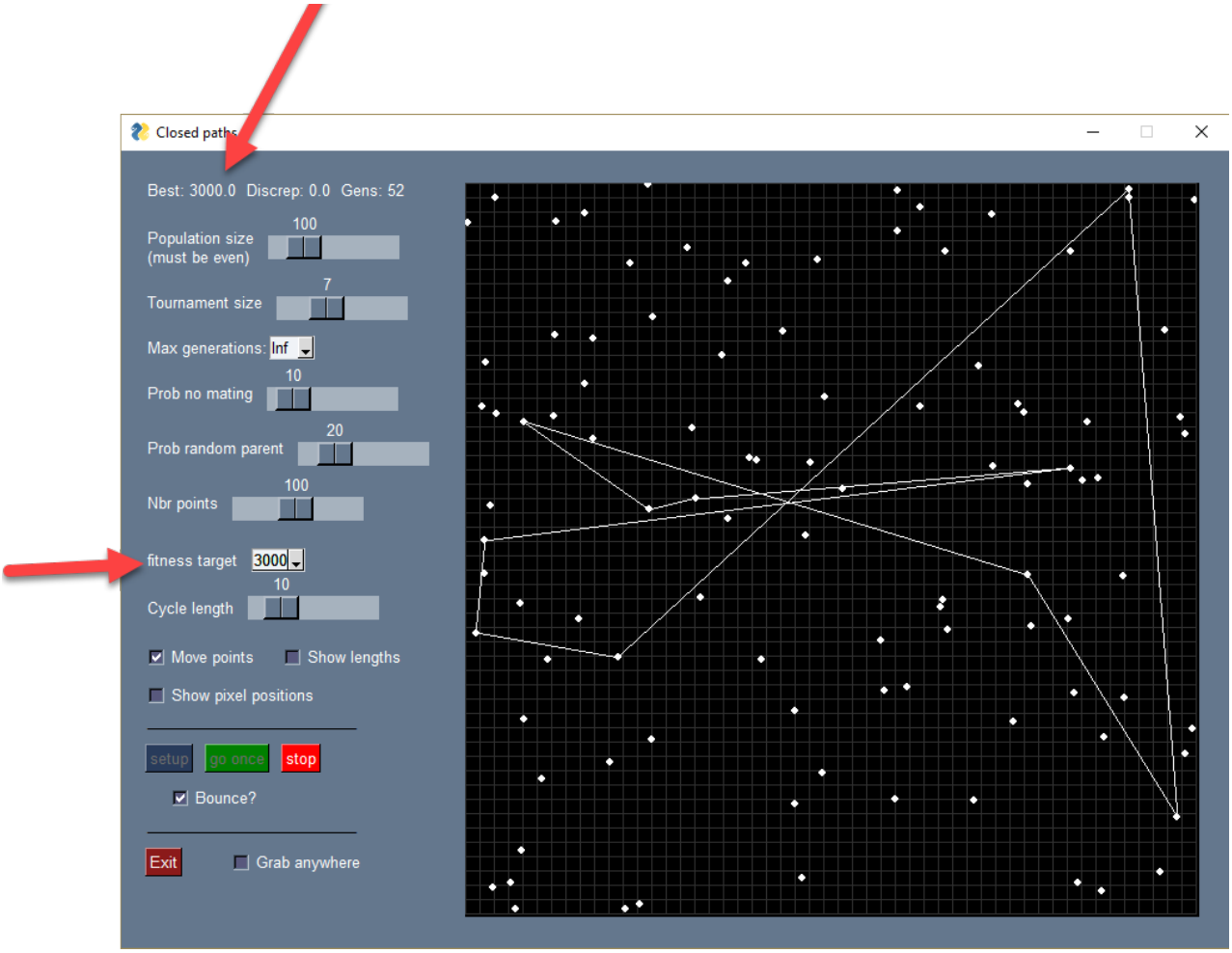


Figure 5 Closed paths

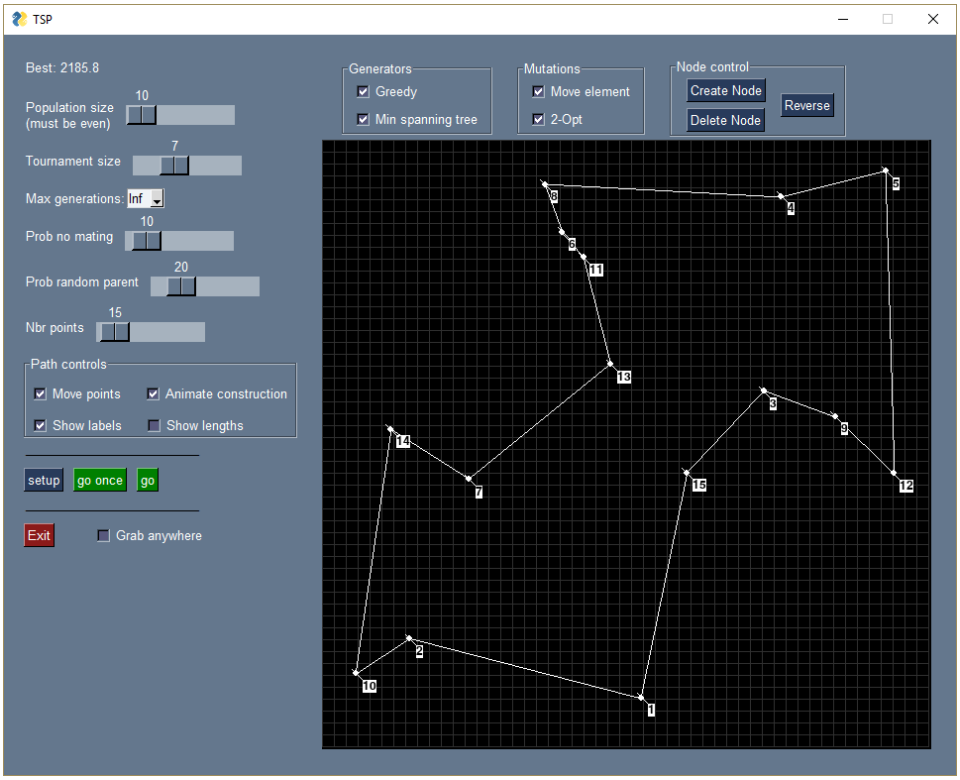


Figure 6 GA TSP

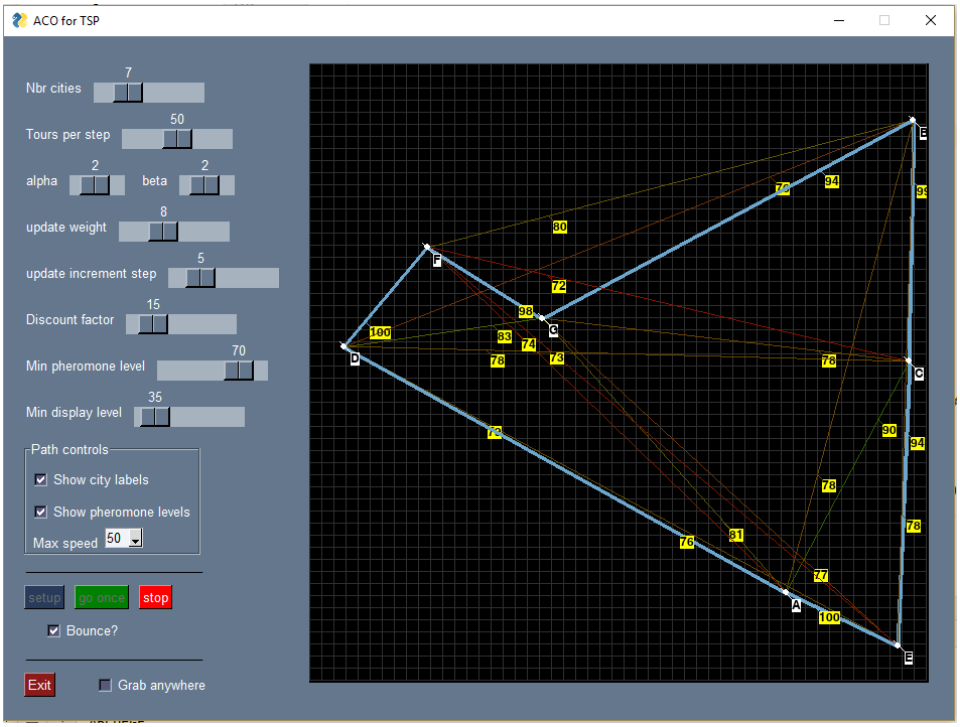


Figure 7 ACO-TSP

D Knapsack

```
Problem 4:
  Capacity: 50
  Items in density order (value/weight): 70/31, 20/10, 39/20, 37/19, 7/4, 5/3, 10/6
  Solution: 107/50<-1001000
0) Pop (value/weight<-selection):  96/49<-0111000,  88/46<-0011110,  59/30<-0110000,  75/34<-1000010,  66/34<-0110100
   Best (value/weight<-selection):  96/49<-0111000
1) Pop (value/weight<-selection):  96/49<-0111000,  88/46<-0011110,  96/49<-0111000,  30/16<-0100001,  76/39<-0011000
   Best (value/weight<-selection):  96/49<-0111000
2) Pop (value/weight<-selection):  96/49<-0111000,  74/39<-0110011,  96/49<-0111000,  105/50<-1100011,  105/50<-1100011
   Best (value/weight<-selection):  105/50<-1100011
3) Pop (value/weight<-selection):  96/49<-0111000,  105/50<-1100011,  96/49<-0111000,  105/50<-1100011,  105/50<-1100011
   Best (value/weight<-selection):  105/50<-1100011
4) Pop (value/weight<-selection):  96/49<-0111000,  105/50<-1100011,  100/47<-1100001,  105/50<-1100011,  105/50<-1100011
   Best (value/weight<-selection):  105/50<-1100011
5) Pop (value/weight<-selection):  107/50<-1001000,  105/50<-1100011,  37/19<-0001000,  105/50<-1100011,  105/50<-1100011
   Best (value/weight<-selection):  107/50<-1001000
```

Figure 8 Knapsack

E 1D Segregation

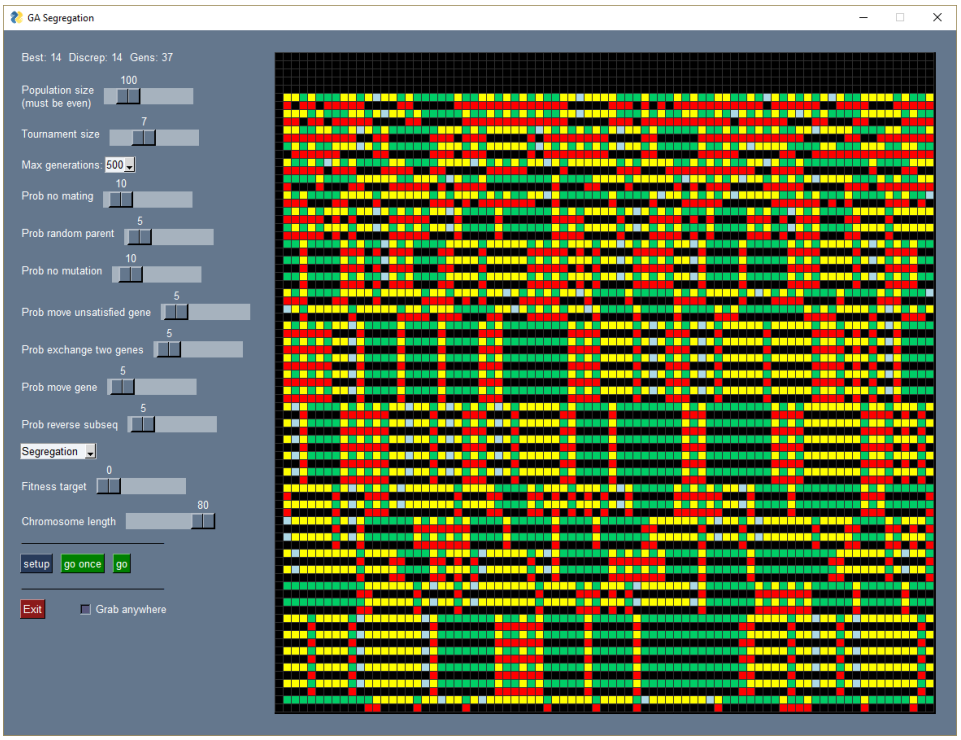


Figure 9 1D Segregation

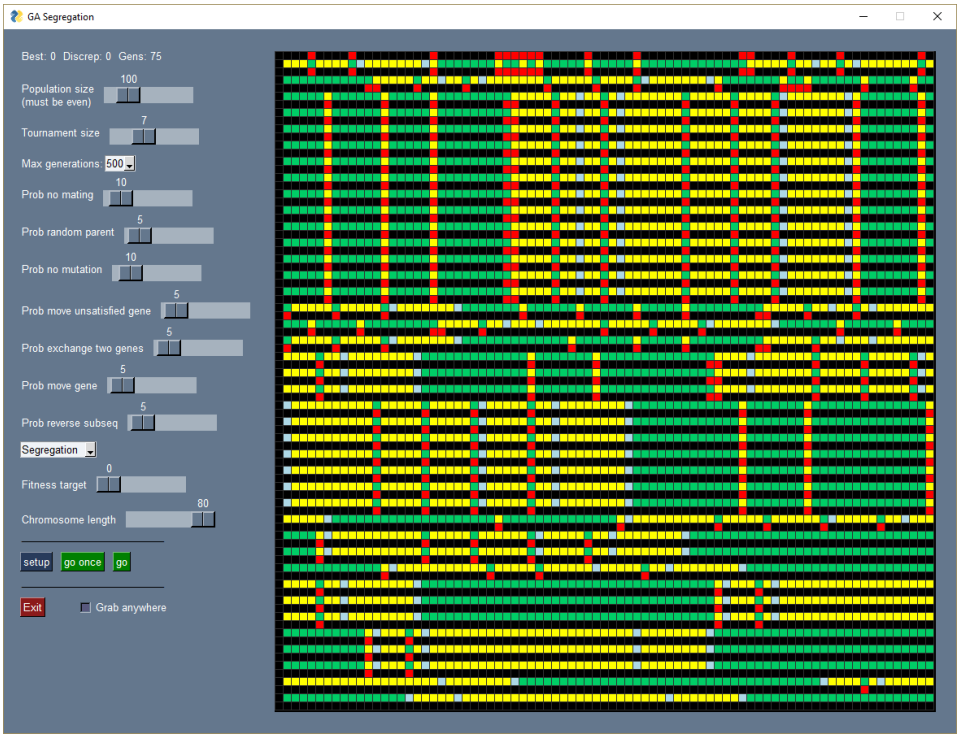


Figure 10 1D Segregation (continued)

F Parentheses

```
Gen 0.  6: ( ) ( ( ) ( ( ) ) ) ( ) ) ( ( ) ( ( ( ) ( ) ( ) ) ) ) ( ) ) ( ) ( ) ( ( ( ( ( ) ) ( ) ) ( ) ( )
                          ^                             ^   ^       ^ ^ ^
Gen 1.  4: ( ) ( ( ) ( ( ) ) ) ( ) ( ( ) ( ( ( ) ( ) ( ) ) ) ) ( ) ) ( ) ( ) ( ( ( ( ( ) ( ) ) ( ) ) ( ) ( )
                                                          ^   ^       ^ ^
Gen 2.  4: ( ) ( ( ) ( ( ) ) ) ( ) ( ( ) ( ( ( ) ( ) ( ) ) ) ) ( ) ) ( ) ( ) ( ( ( ( ( ) ( ) ) ( ) ) ( ) ( )
                                                          ^   ^       ^ ^
Gen 3.  4: ( ) ( ( ) ( ( ) ) ) ( ) ( ( ) ( ( ( ) ( ) ( ) ) ) ) ( ) ) ( ) ( ) ( ( ( ( ( ) ( ) ) ( ) ) ( ) ( )
                                                          ^   ^       ^ ^
Gen 4.  0: ( ) ( ( ) ( ( ) ) ) ( ) ( ( ) ( ( ( ( ( ) ( ( ) ) ) ) ) ) ( ) ( ( ( ( ) ) ( ) ) ) ) ( ( ) ) ( ) ( )
```

Figure 11 Parentheses

G 1D Cellular Automata

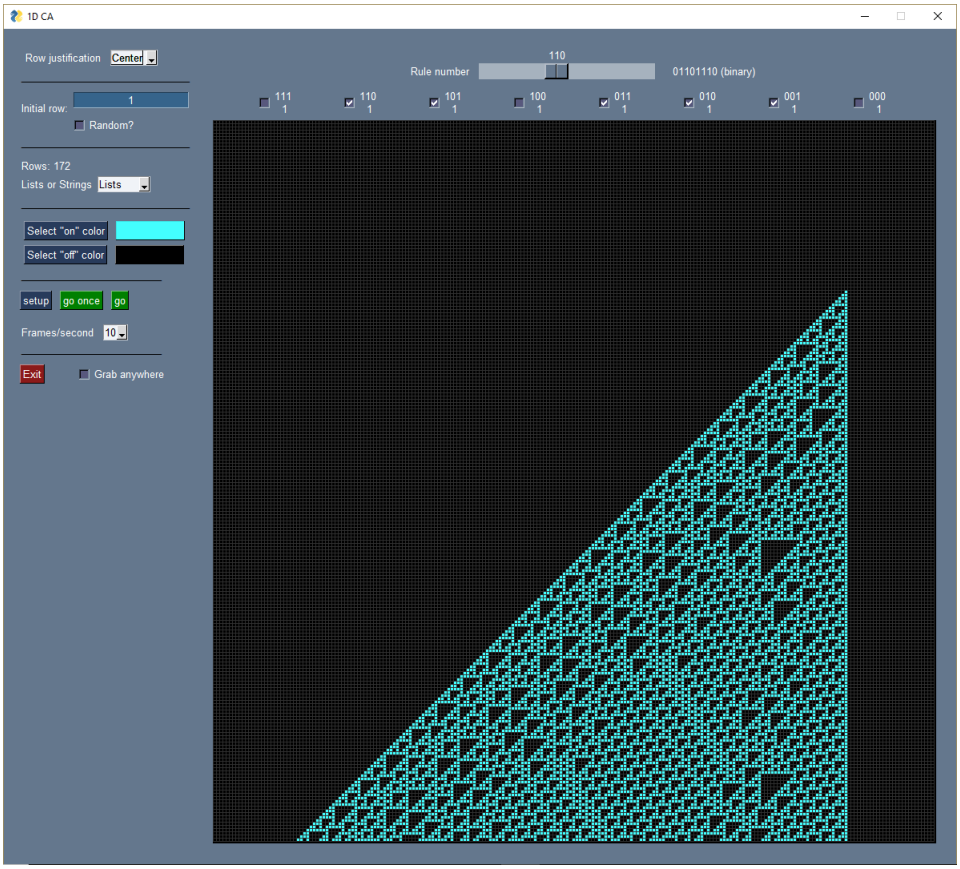


Figure 12 1D Cellular Automata

H Game of Life

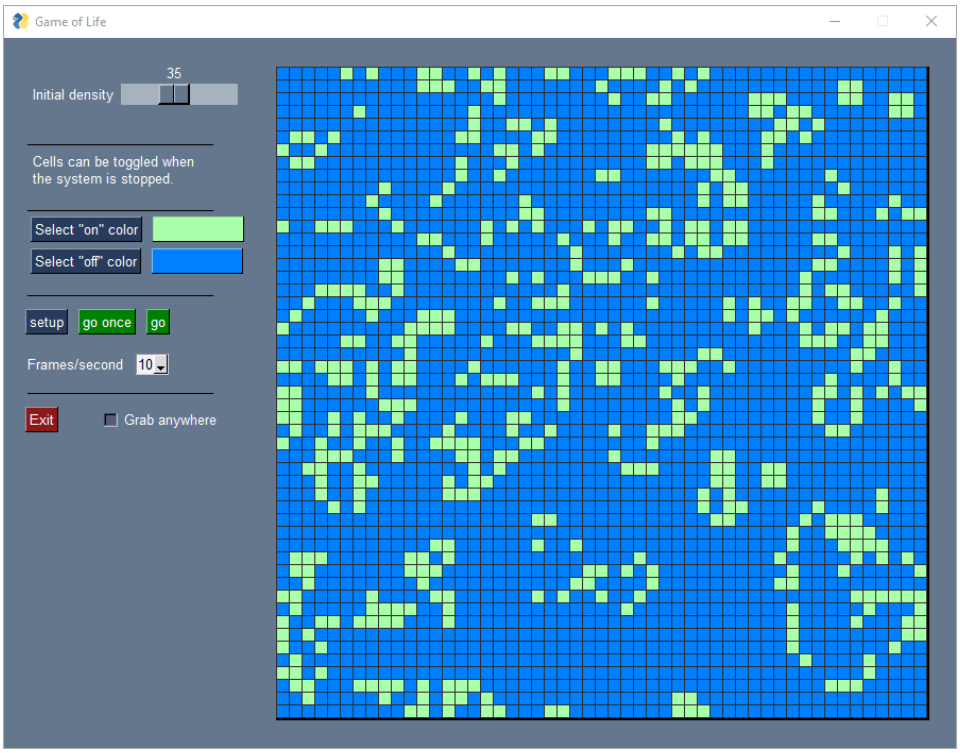


Figure 13 Game of Life

The *Game of Life*, the smallest PyLogo model, extends the *OnOffPatch* and *OnOffWorld* classes (defined in *on_of.py*, Section 4.2.4). *Game of Life* can be a small model because the *OnOffPatch* and *OnOffWorld* classes do much of the work it needs. In particular, *OnOffPatch* defines whether a *Patch* is *on* or *off*, which the *Game of Life* model uses to represent whether a cell is *alive* or *dead*.

The following notes are linked to line numbers in Listing 1 below.

- 1-5. Imports. In particular, *Game of Life* does not import or subclass an *Agent* class. It imports *OnOffPatch*, and *OnOffWorld* both of which it subclasses.
- 8-21. The *Life_Patch* class is a subclass of *Patch*.
- 10-12. The *__init__* method defines the *live_neighbors* instance variable, which is used to keep track of live neighbors.
- 14-15. The *count_live_neighbors* method counts the number of live neighbors and stores the result in the *live_neighbors* instance variable.
- 17-18. The *is_alive* method returns *True* or *False* depending on the value of *self.is_on*, which is defined in *OnOffPatch*.
- 20-21. The *set_alive_or_dead* method uses the method *self.set_on_off*, which is defined in *OnOffPatch* (Section 4.2.4), to store whether this *Patch* is alive or dead.
- 24-41. The *Life_World* class, a subclass of *OnOffWorld*. Defines *setup* and *step*.
  - 26-31. The *setup* method initializes the model.

    - 27. It calls the *setup* in *OnOffWorld*, its superclass.
    - 28. It reads the value of *density* from the GUI. *density* is taken to be the probability that any *Patch* will initially be alive.
    - 29-31. It sets each *Patch* to be alive or dead probabilistically based on *density*.
  - 33-41. The *step* method initializes the model.
    - 35-36. It counts and stores the number of each *Patch*'s live neighbors.
    - 39-41. It determines whether each *Patch* is alive based on the count of its live neighbors.
- 45-55. Import *PySimpleGUI* and define the model-specific GUI. Widgets are specified as a list of lists. Each internal list is a line of widgets. There are four lines.
  1. A labelled widget for density.
  2. A horizontal separator.
  3. Text displayed as information to the user.
  4. Another horizontal separator.
- 54-55. All that is prepended to the left-upper widgits defined in *core.on_off.py*.
- 58. Run this file if it is executed directly.
- 59. Import *PyLogo* from *core.agent.py*, where it is defined.
- 59-60. Call *PyLogo*, pass the relevant parameters, and run the model.

```
from random import randint

from core.gui import HOR_SEP
from core.on_off import OnOffPatch, OnOffWorld, on_off_left_upper
from core.sim_engine import gui_get


class Life_Patch(OnOffPatch):

    def __init__(self, *args, **kw_args):
        super().__init__(*args, **kw_args)
        self.live_neighbors = 0

    def count_live_neighbors(self):
        self.live_neighbors = sum([1 for p in self.neighbors_8() if p.is_alive()])

    def is_alive(self):
        return self.is_on

    def set_alive_or_dead(self, alive_or_dead: bool):
        self.set_on_off(alive_or_dead)


class Life_World(OnOffWorld):

    def setup(self):
        super().setup()
        density = gui_get('density')
        for patch in self.patches:
            is_alive = randint(0, 100) < density
            patch.set_alive_or_dead(is_alive)

    def step(self):
        # Count the live neighbors in the current state.
        for patch in self.patches:
            patch.count_live_neighbors()

        # Determine and set whether each patch is_alive in the next state.
        for patch in self.patches:
            is_alive = patch.live_neighbors == 3 or patch.is_alive() and patch.live_neighbors == 2
            patch.set_alive_or_dead(is_alive)


# # # # # # # # # # # # # Define Game-of-Life GUI # # # # # # # # # # # # #
import PySimpleGUI as sg

gol_left_upper = [[sg.Text('Initial density'),
                   sg.Slider(key='density', range=(0, 80), resolution=5, size=(10, 20),
                             default_value=35, orientation='horizontal', pad=((0, 0), (0, 20)),
                             tooltip='The ratio of alive cells to all cells')],
                  HOR_SEP(pad=((0, 0), (0, 0))),
                  [sg.Text('Cells can be toggled when\nthe system is stopped.')],
                  HOR_SEP(pad=((0, 0), (0, 0))),
                  ] + \
                  on_off_left_upper


if __name__ == "__main__":
    from core.agent import PyLogo
    PyLogo(Life_World, 'Game of Life', gol_left_upper, patch_class=Life_Patch, fps=10)
```

Listing 1 *Starburst* model

I Flocking

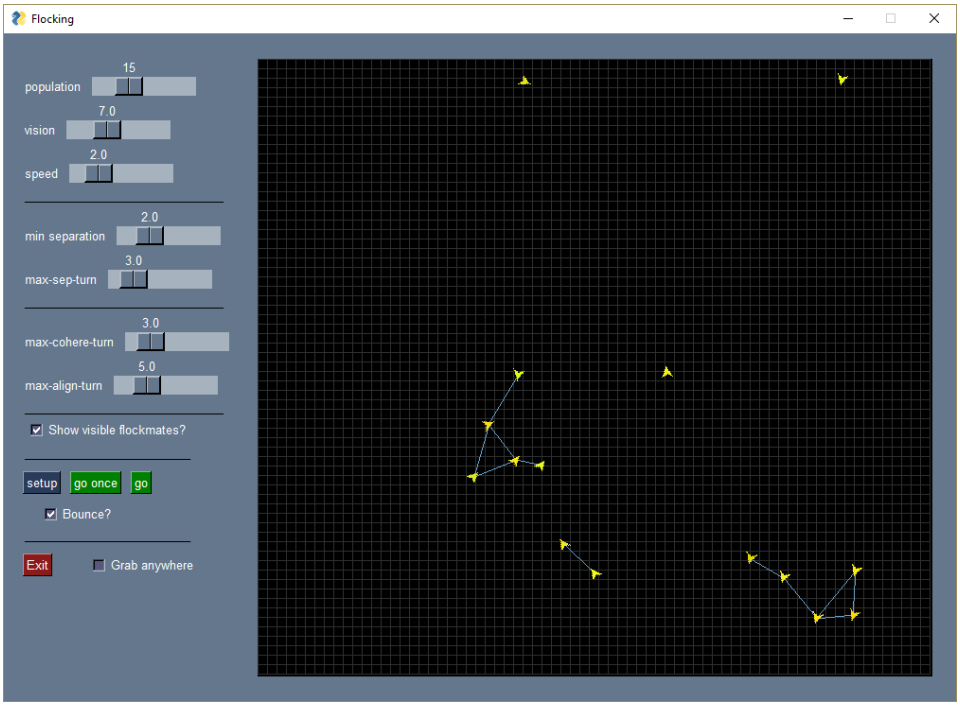


Figure 14 Flocking (bounce on)

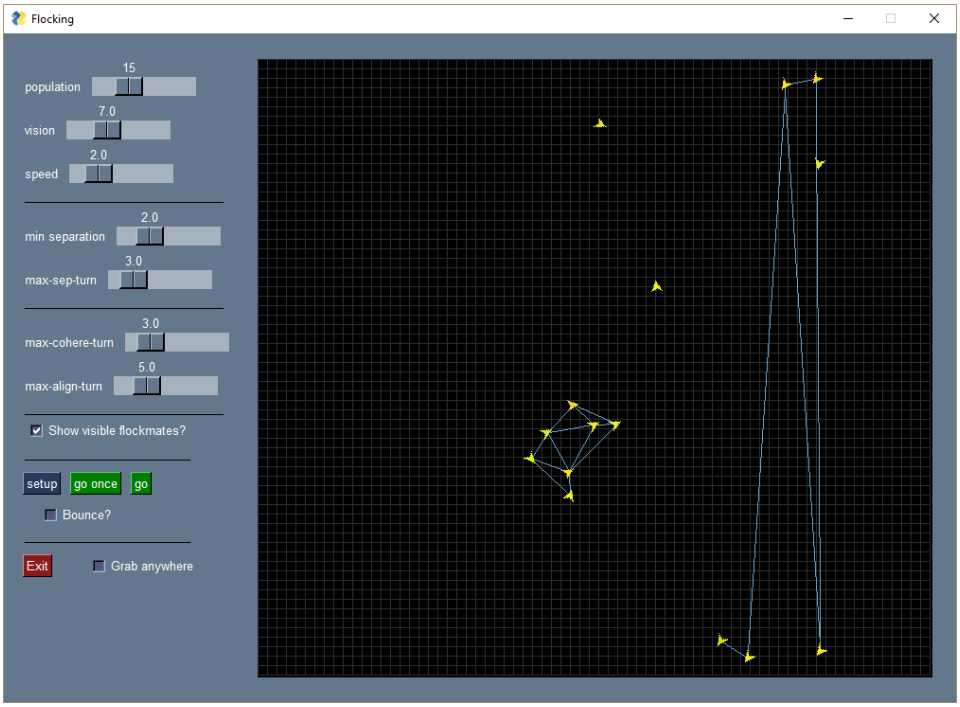


Figure 15 Flocking (bounce off)

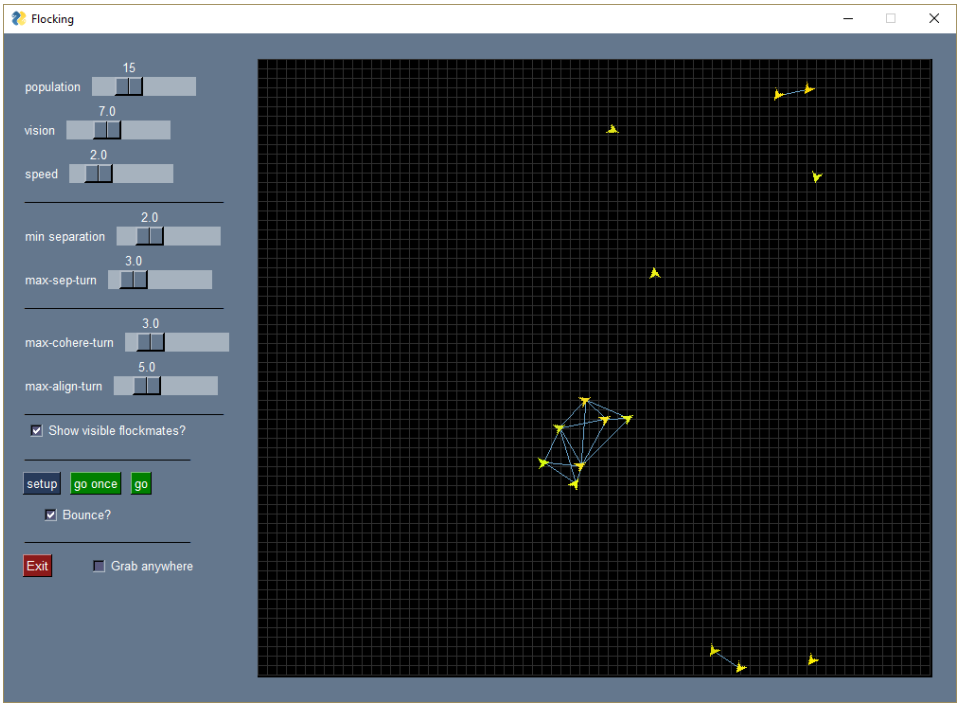


Figure 16 Flocking (bounce back on)

J Graph algorithms

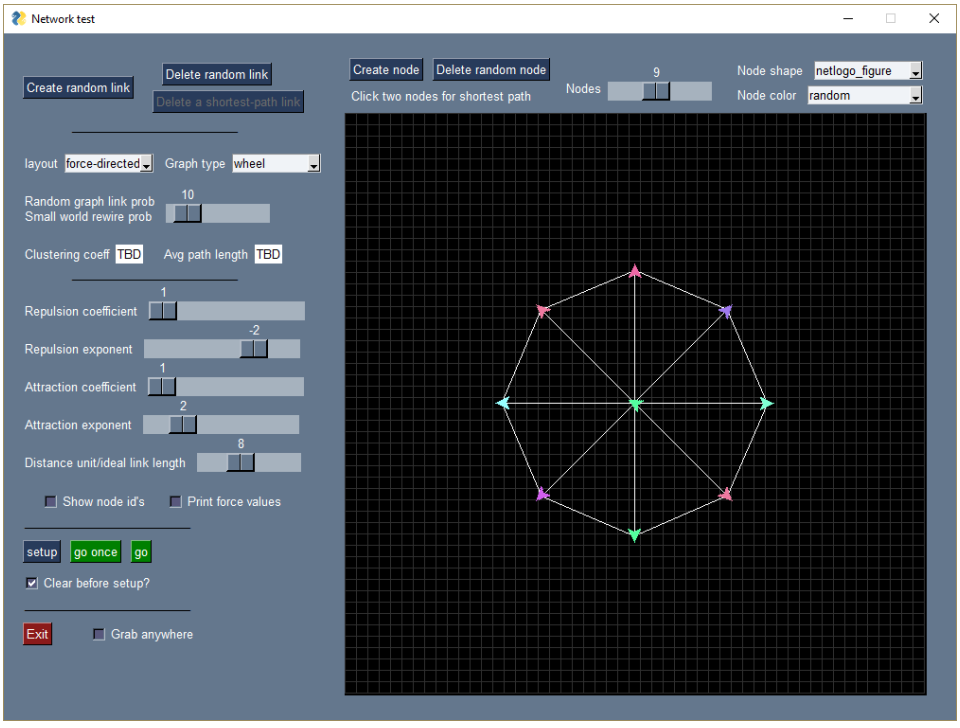


Figure 17 A wheel graph

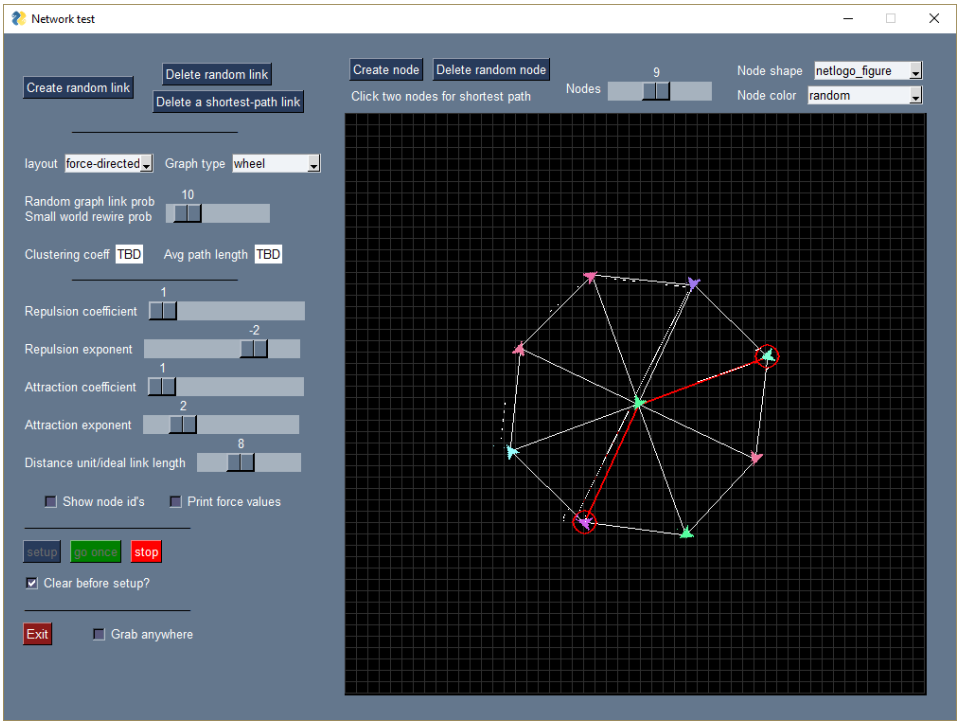


Figure 18 A wheel graph with two selected nodes and a shortest path between them

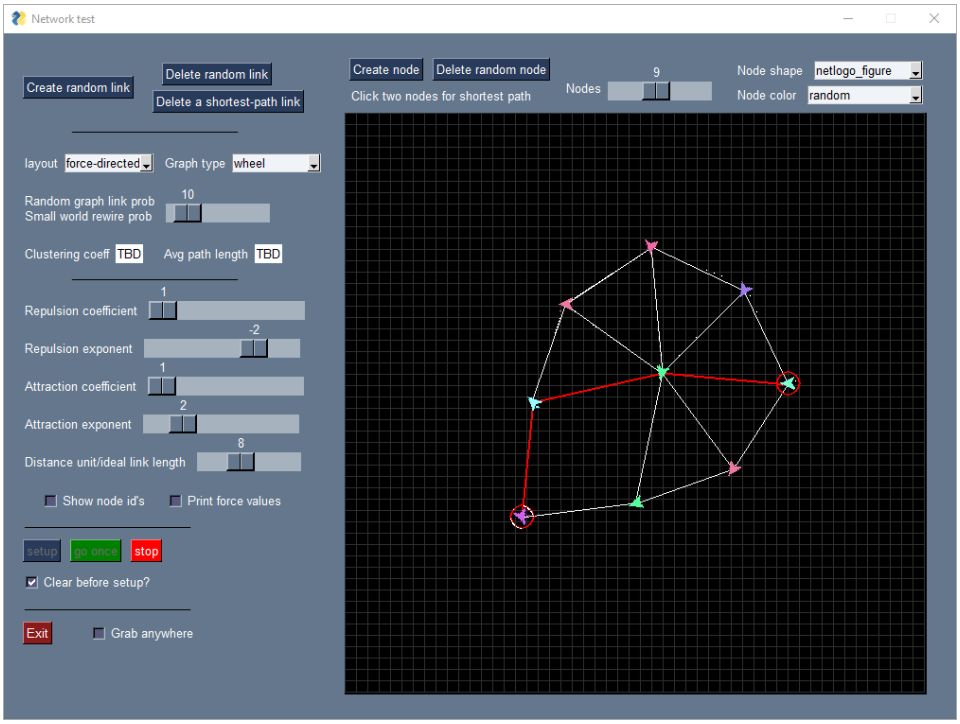


Figure 19 The previous graph with a shortest path link removed

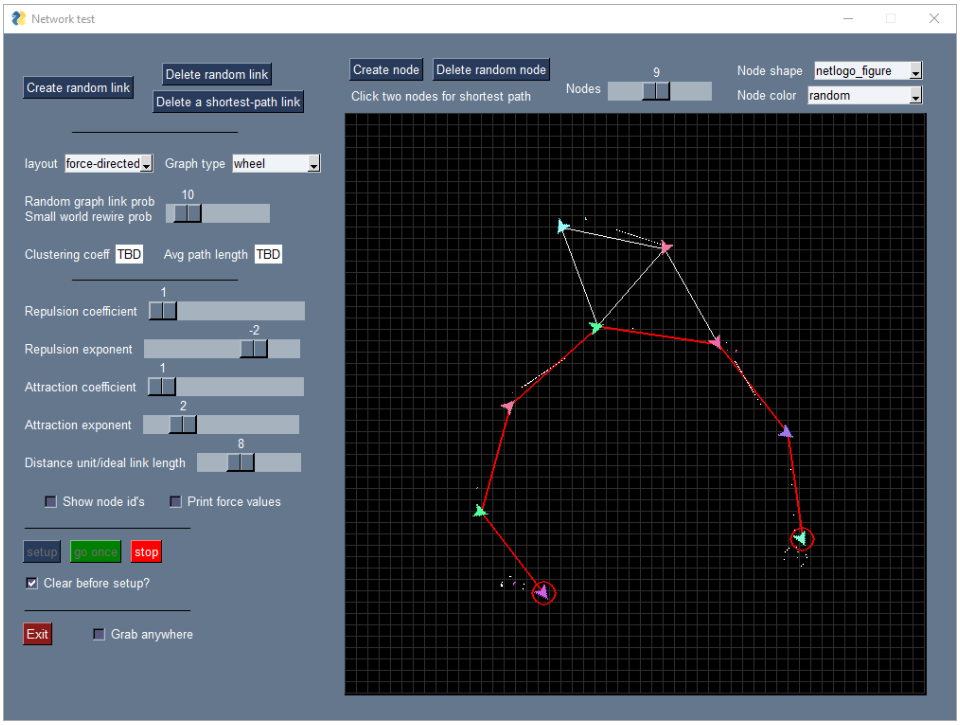


Figure 20 The previous graph with additional shortest path links removed

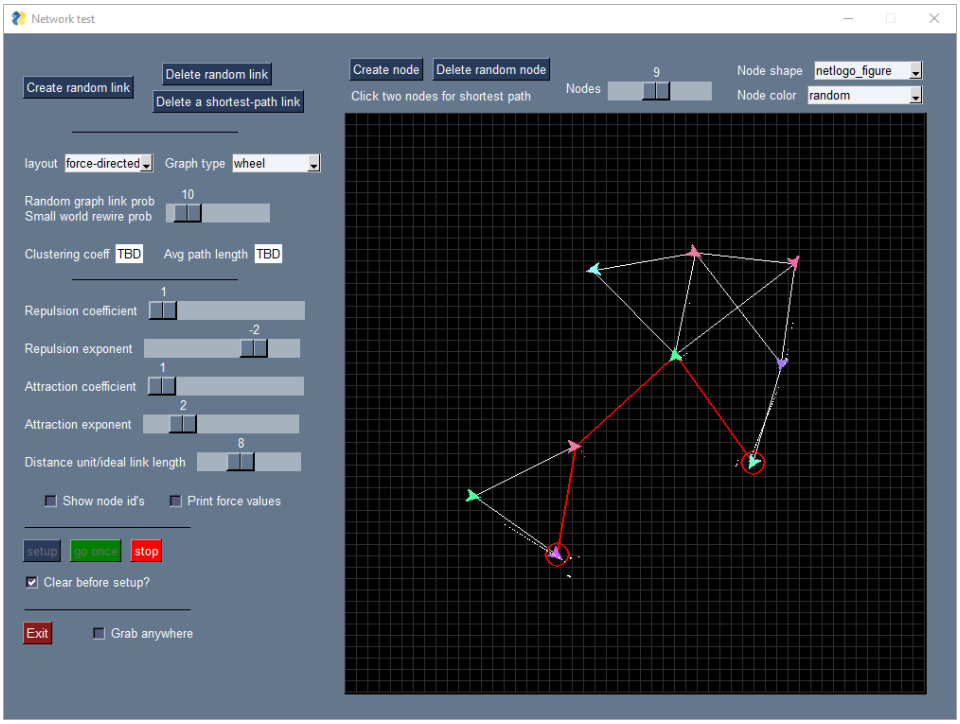


Figure 21 The previous graph with some random links added

K Minority Game

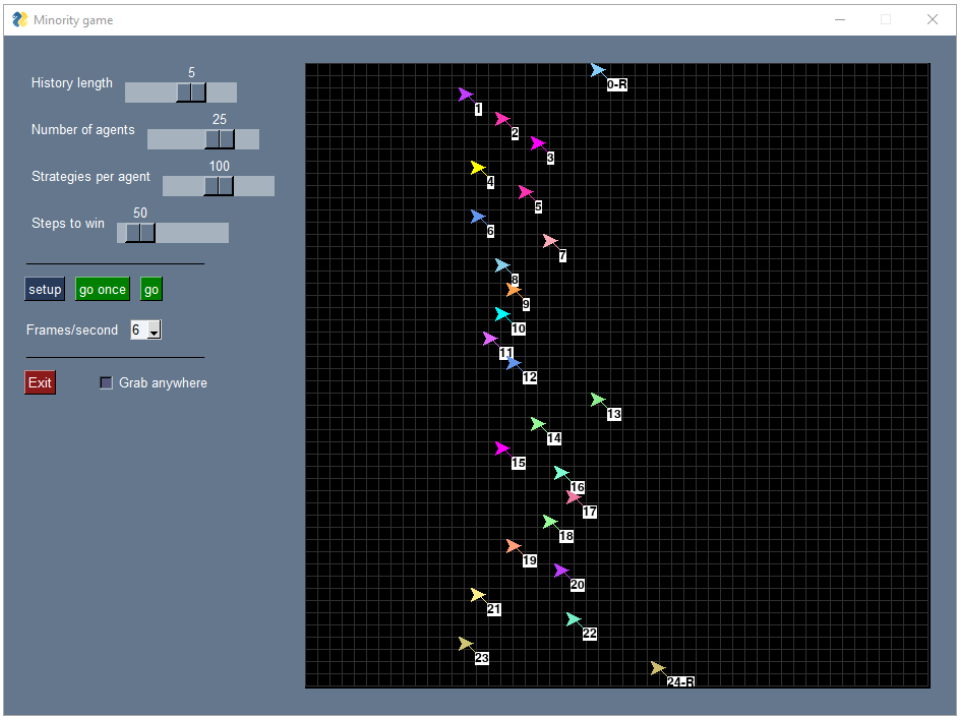


Figure 22 The Minority Game "horse race"

```
19. [1, 0, 0, 1, 1], 19, 0, {17: 12/50}
20. [0, 0, 1, 1, 0],  6, 0, {17: 12/50}
21. [0, 1, 1, 0, 0], 12, 0, {17: 13/50}
22. [1, 1, 0, 0, 0], 24, 0, {17: 13/50}
23. [1, 0, 0, 0, 0], 16, 0, {17: 13/50}
24. [0, 0, 0, 0, 0],  0, 0, {17: 13/50, 24: 13/50}
25. [0, 0, 0, 0, 0],  0, 1, {17: 14/50, 24: 14/50}
26. [0, 0, 0, 0, 1],  1, 0, {17: 15/50}
27. [0, 0, 0, 1, 0],  2, 0, {17: 15/50, 24: 15/50}
28. [0, 0, 1, 0, 0],  4, 0, {24: 16/50}
29. [0, 1, 0, 0, 0],  8, 1, {24: 16/50}
30. [1, 0, 0, 0, 1], 17, 0, {24: 17/50}
31. [0, 0, 0, 1, 0],  2, 1, {24: 18/50}
32. [0, 0, 1, 0, 1],  5, 0, {24: 19/50}
33. [0, 1, 0, 1, 0], 10, 0, {24: 19/50}
34. [1, 0, 1, 0, 0], 20, 0, {24: 19/50}
35. [0, 1, 0, 0, 0],  8, 0, {24: 20/50}
36. [1, 0, 0, 0, 0], 16, 1, {24: 20/50}
37. [0, 0, 0, 0, 1],  1, 1, {24: 21/50}
38. [0, 0, 0, 1, 1],  3, 0, {24: 21/50}
39. [0, 0, 1, 1, 0],  6, 1, {24: 22/50}
40. [0, 1, 1, 0, 1], 13, 0, {24: 22/50}
41. [1, 1, 0, 1, 0], 26, 1, {24: 23/50}
42. [1, 0, 1, 0, 1], 21, 0, {24: 24/50}
43. [0, 1, 0, 1, 0], 10, 1, {24: 25/50}
44. [1, 0, 1, 0, 1], 21, 1, {24: 25/50}
45. [0, 1, 0, 1, 1], 11, 0, {24: 26/50}
46. [1, 0, 1, 1, 0], 22, 0, {24: 26/50}
47. [0, 1, 1, 0, 0], 12, 1, {24: 26/50}
48. [1, 1, 0, 0, 1], 25, 1, {24: 27/50}
49. [1, 0, 0, 1, 1], 19, 1, {24: 27/50}
50. [0, 0, 1, 1, 1],  7, 0, {24: 27/50}
51. [0, 1, 1, 1, 0], 14, 1, {24: 27/50}
52. [1, 1, 1, 0, 1], 29, 0, {24: 28/50}
53. [1, 1, 0, 1, 0], 26, 0, {24: 28/50}
54. [1, 0, 1, 0, 0], 20, 0, {24: 28/50}
```

Figure 23 The Minority Game partial log

Why agents that select randomly are better than agents with strategies

In our experiments, agents that selected *0* or *1* randomly almost always beat the agents with strategies. Here is a simple explanation. (The following differs from the actual *Minority Game* model in which agents select among their strategies based on which have done better.)

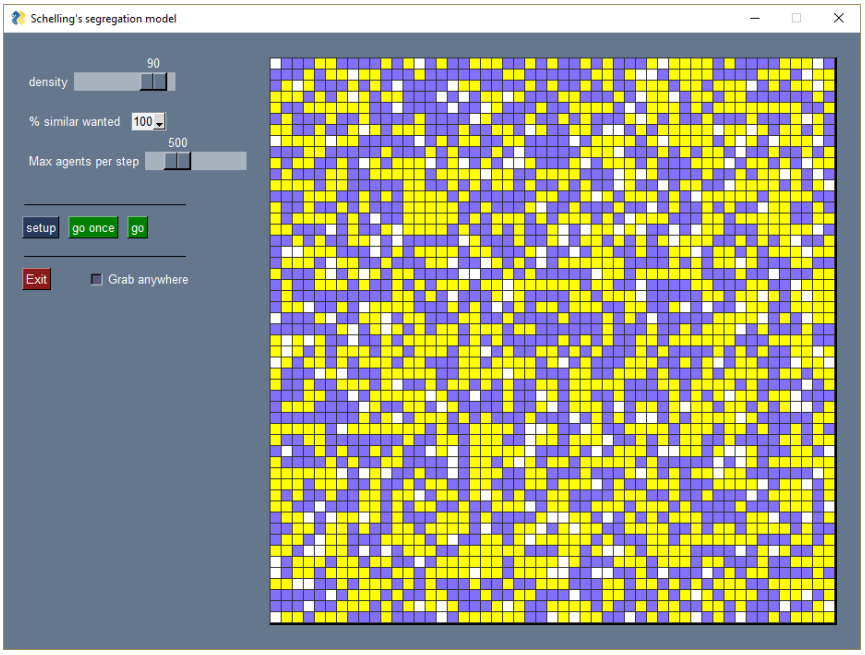
Suppose that each agent has only a single strategy—or selects randomly. Then for any history index, we know what the agents with strategies will select, and we know which ones will win and lose for that history index. The same thing will happen each time that history index occurs.

On average, agents will lose more often than they will win—because this is a minority game. For some history indices, because the selections are unbalanced, many more agents will lose than will win. So the average agent will lose significantly more often than it will win. With enough agents and a long enough game, all agents will fit that pattern.

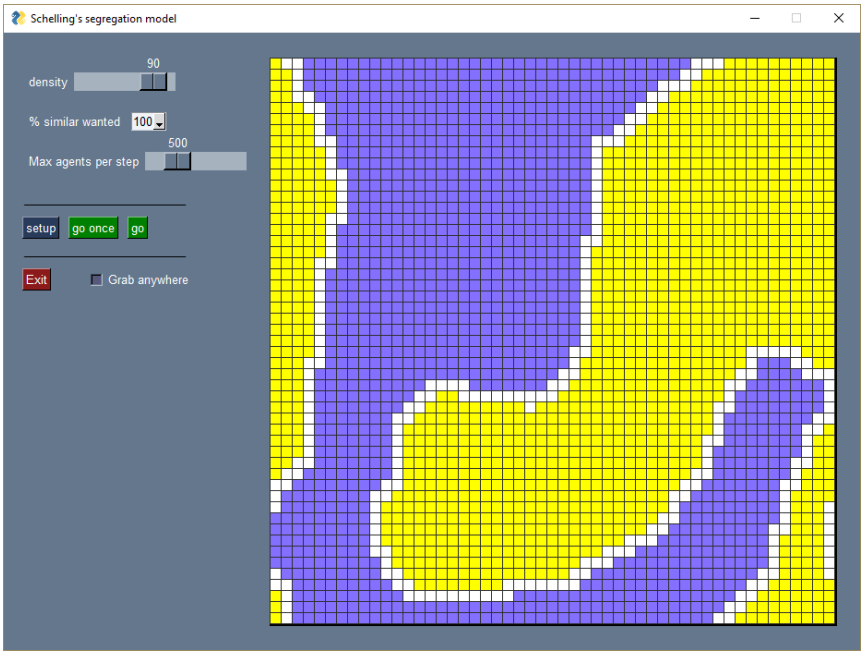
What about agents that select randomly? Each of those agents has only slightly less than a 50% chance to win for any history index. And that will happen each time that index occurs—it is not determined. These agents have less than a 50% chance to win because if the other agents are exactly balanced, whatever choice the random agent makes will become the majority. (Remember that we insisted on an odd number of agents.) But an even split among the other agents will be relatively rare, especially if we have a lot of agents. So the random agents will win almost 50% of the time—and certainly more than agents with fixed strategies.

This is admittedly a simplification of the actual game, but an elaboration of the same argument should hold in that case as well.

L Schelling segregation



(a) Start state



(b) End state

Figure 24 Segregation start and end states with similarity wanted 100% and density 90%

M Starburst

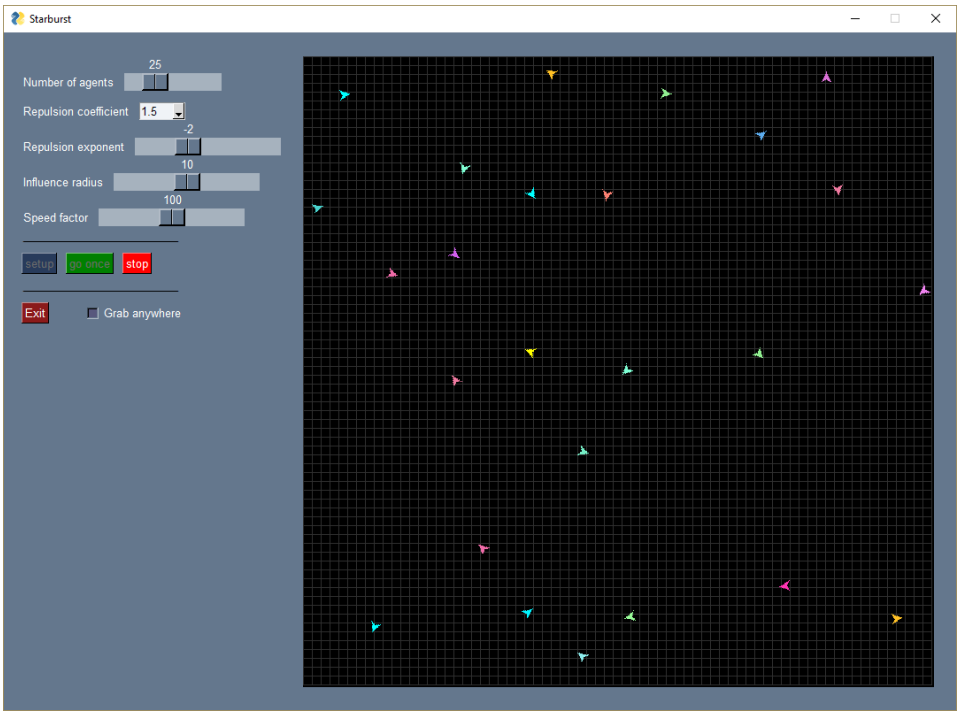


Figure 25 Starburst

The following notes are linked to line numbers in Listing 2 below. (Note that the listing continues on a second page.)

- 1-8.Imports. In particular, *Starburst* imports *Agent* and *World*, both of which it subclasses. It uses the system-defined *Patch* class.
- 11-24. The *Starburst_Agent* class is a subclass of *Agent*. It has no *__init__* method since *Agents* have no model-specific instance variables.
- 13-24. The *update_velocity* function modifies the agent's velocity to include the repulsive forces of its neighbors.
  - 15-16. Consider only neighbors within *Influence radius*, a slider in the GUI.
  - 17-22. For each neighbor:
    - 18-22. If *forces_cache* does not already contain the repulsive force from this neighbor, Call *force_as_dxdy* (Section 4.2.5) to compute the force. (Note that if the distance between two agents is 0, *force_as_dxdy* chooses a random vector with coordinates in [-1, 0, 1].)
    - 21. Cache the inverse of the force so that we don't have to look of a pair twice. (Agents repel each other equally.)
    - 24. Add the force to the current velocity.
  - 23-24. Normalize the force and update the agent's velocity. (*PySimpleGUI Sliders* are limited to integers. Hence dividing by 100. An alternative is to use a *ComboBox* (a drop-down selector), which supports all values.)

- 35-43. The *Starburst_World* class, a subclass of *World*. Defines *setup* and *step*.
  - 29-36. The *setup* method initializes the model.
    - 30. Read the GUI widget with key *nbr_agents* (defined on lines 55-57).
    - 31-32. Create the indicated number of agents. Note that an agent is created by calling *self.agent_class*, which is bound to *Starburst_Agent*. That occurs on line 78 in the call to *PyLogo*.
    - 34. Define five initial *Velocities*.
    - 35-36. The *Agents*, which had been stored in a *set* of *Agents* (a class level variable in *World*), are paired with the *Velocities*. The *cycle* function, imported on line 1, repeats them as needed.
  - 38-43. The *step* function.
    - 39-43. If we have reached "Burst ticks," determine each agent's new velocity based on its current position and near neighbors.
    - 41. Update all agents' current positions based on their new velocities.
- 47-74 (on the next page). Import *PySimpleGUI* and define the model-specific GUI. Widgets are specified as a list of lists. Each internal list is a line of widgets. In this case, each line contains a *Text* widget as a label and either a *Slider* widget or a *ComboBox* widget for a value.
- 77. Run this file if it is executed directly.
- 78-79. Call *PyLogo*, pass the relevant parameters, and run the model.

```
from itertools import cycle

from core.agent import Agent, PyLogo
from core.gui import BLOCK_SPACING
from core.pairs import REP_COEFF, REP_EXPONENT, Velocity, force_as_dxdy
from core.sim_engine import gui_get
from core.utils import normalize_dxdy
from core.world_patch_block import World


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()


# # # # # # # # # # # # # Define Starburst GUI # # # # # # # # # # # # #
import PySimpleGUI as sg
gui_left_upper = [ [sg.Text('Number of agents', pad=((0, 5), (20, 0))),
                    sg.Slider(key='nbr_agents', range=(1, 101), resolution=25, default_value=25,
                              orientation='horizontal', size=(10, 20))],

                   [sg.Text('Burst tick', pad=((0, 10), (20, 0)), tooltip='The burst tick'),
                    sg.Slider(range=(0, 900), default_value=710, resolution=10, size=(15, 20),
                              orientation='horizontal', key='Burst tick', tooltip='The burst tick')],

                   [sg.Text('Repulsion coefficient', pad=((0, 10), (15, 0)), tooltip='Larger is stronger.'),
                    sg.Combo((0.5, 1.0, 1.5, 2.0, 2.5, 3.0), key=REP_COEFF, default_value=1.5,
                             size=(5, 20), tooltip='Larger is stronger.', pad=((0, 0), (15, 0)))],

                   [sg.Text('Repulsion exponent', pad=((0, 10), (20, 0)),
                            tooltip='Negative means raise to the power and divide (like gravity).\n'
                                    'Larger magnitude means distince reduces repulsive force more.'),
                    sg.Slider(range=(-3, 0), default_value=-2, orientation='horizontal',
                              key=REP_EXPONENT, size=(15, 20),
                              tooltip='Negative means raise to the power and divide (like gravity).\n'
                                      'Larger magnitude means distince reduces repulsive force more.')],
```

```
67
68                      [sg.Text('Influence radius', pad=((0, 10), (20, 0)), tooltip='Influence radius'),
69                       sg.Slider(range=(0, 20), default_value=10, orientation='horizontal',
70                                 key='Influence radius', size=(15, 20), tooltip='Influence radius')],
71
72                      [sg.Text('Speed factor', pad=((0, 10), (20, 0)), tooltip='Relative speed'),
73                       sg.Slider(range=(0, 200), default_value=100, orientation='horizontal',
74                                 key='Speed factor', size=(15, 20), tooltip='Relative speed')] ]
75
76
77 if __name__ == "__main__":
78     PyLogo(Starburst_World, 'Starburst', gui_left_upper, agent_class=Starburst_Agent,
79            bounce=(True, False), patch_size=9, board_rows_cols=(71, 71))
```

Listing 2 *Starburst* model

N Synchronized Agents

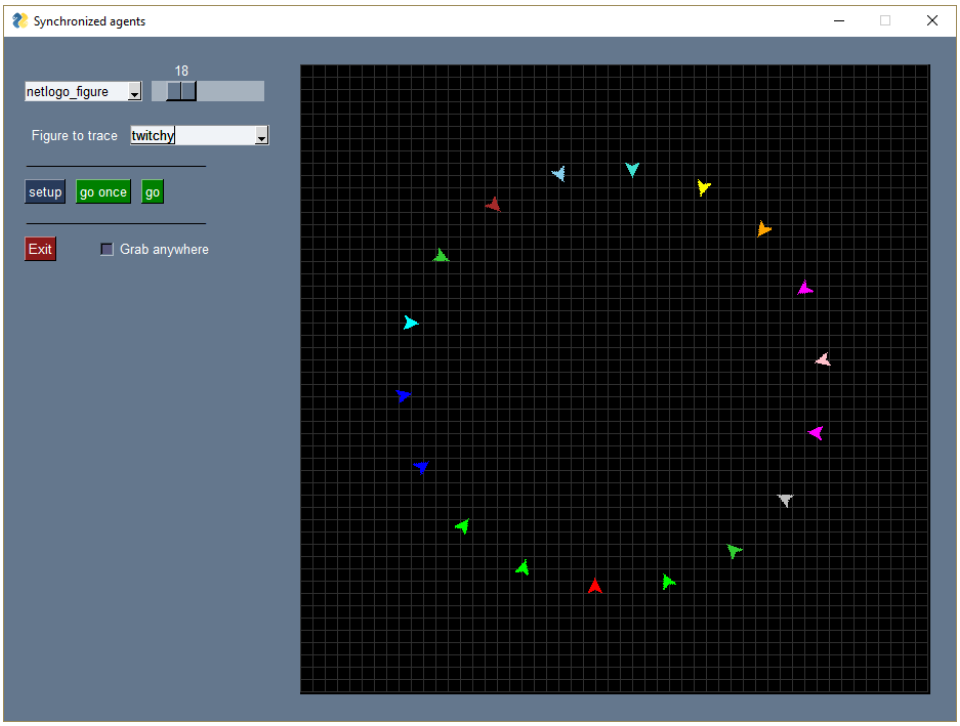


Figure 26 Synchronized Agents

O PyLogo

This appendix presents the *PyLogo* function and the *PyLogo* function parameters.

O.1 The PyLogo function

```
def PyLogo(world_class=World, ... # See the PyLogo appendix for parameter descriptions ):
    if gui_left_upper is None:
        gui_left_upper = []
    if caption is None:
        caption = utils.extract_class_name(world_class)

    # Create sim_engine, an instance of the SimEngine class in sim_engine.py.
    sim_engine = SimEngine( # Pass the relevant PyLogo parameters to SimEngine )

    # Instantiating SimEngine triggers the creation of the GUI elements, including the
    # main window: gui.WINDOW. We can now read initial widget values from gui.WINDOW.
    gui.WINDOW.read(timeout=10)

    # Instantiate the world_class and store the patch_class and agent_class
    # as values of its instance variable.
    the_world = world_class(patch_class, agent_class)

    # Read events and values from gui.WINDOW (again).
    gui.WINDOW.read(timeout=10)

    # Start the top-loop event loop in sim_engine.
    sim_engine.top_loop(the_world, auto_setup=auto_setup)
```

O.2 PyLogo parameters

The *PyLogo parameters*.

- *world_class*. The *World* class to use. **Default**: *World*.
- *caption*. The caption to put at the top-left of the window frame. **Default**: *None*.
- *gui_left_upper*. *PySimpleGUI* code for the upper left side. **Default**: *None*.
- *agent_class*. The *Agent* class to use. **Default**: *Agent*.
- *patch_class*. The *Patch* class to use. **Default**: *Patch*.
- *gui_right_upper*. *PySimpleGUI* code for above the grid. **Default**: *None*.
- *auto_setup*. Run *setup* automatically on loading the model. **Default**: *True*.
- *patch_size*. The patch size in pixels. **Default**: *11*.
- *board_rows_cols*. The *row x column* grid dimensions. **Default**: *(51, 51)*.
- *clear*. Whether to include a widget to allow the user to specify whether to clear the world before *setup*. **Default**: *None*. (No checkbox; clear the world.)
- *fps*. The maximum frames-per-second at which to run the model. **Default**: *60*.
- *bounce*. A system-defined *bounce* checkbox may appear below the *setup*, *go once*, and *go* buttons. The *bounce* argument is a tuple with two boolean values: *(bounce-value, checkbox-is-visible)*. Whether the checkbox is initially checked depends on *bounce-value*. Whether the checkbox is visible depends on *checkbox-is-visible*.
  - If one supplies a single boolean value (rather than a tuple) that value is taken as the *bounce-value*, and *checkbox-is-visible* is taken to be *True*. The checkbox is initially checked or not according to the value provided, and the checkbox is accessible for the user change.
  - If no value is provided, both values are treated as *False*. There is no bounce; i.e., the world is toroidal. The user has no means to change that condition.

– If the user supplies a tuple whose second element is *False*, the checkbox is initially checked or not depending the first value, but the user has no means to change that condition. (An argument of *(False, False)* is equivalent to providing no argument.)

Default: *None*. Equivalent to *(False, False)*. No bounce; checkbox not accessible.

About the authors

Russ Abbott is the author of this paper.
Contact him at rabbott@calstatela.edu.

Jungsoo Lim is a co-author of this paper.
Contact her at jlim34@calstatela.edu.

Ricardo Medina is a co-author of this paper.
Contact him at rmedina0531@gmail.com.